Natural Language Processing Using R

Pocket Primer

O. Campesato

Natural Language Processing Using R

Pocket Primer

LICENSE, DISCLAIMER OF LIABILITY, AND LIMITED WARRANTY

By purchasing or using this book and companion files (the "Work"), you agree that this license grants permission to use the contents contained herein, including the disc, but does not give you the right of ownership to any of the textual content in the book/disc or ownership to any of the information or products contained in it. *This license does not permit uploading of the Work onto the Internet or on a network (of any kind) without the written consent of the Publisher.* Duplication or dissemination of any text, code, simulations, images, etc. contained herein is limited to and subject to licensing terms for the respective products, and permission must be obtained from the Publisher or the owner of the content, etc., in order to reproduce or network any portion of the textual material (in any media) that is contained in the Work.

Mercury Learning and Information ("MLI" or "the Publisher") and anyone involved in the creation, writing, or production of the companion disc, accompanying algorithms, code, or computer programs ("the software"), and any accompanying Web site or software of the Work, cannot and do not warrant the performance or results that might be obtained by using the contents of the Work. The author, developers, and the Publisher have used their best efforts to ensure the accuracy and functionality of the textual material and/or programs contained in this package; we, however, make no warranty of any kind, express or implied, regarding the performance of these contents or programs. The Work is sold "as is" without warranty (except for defective materials used in manufacturing the book or due to faulty workmanship).

The author, developers, and the publisher of any accompanying content, and anyone involved in the composition, production, and manufacturing of this work will not be liable for damages of any kind arising out of the use of (or the inability to use) the algorithms, source code, computer programs, or textual material contained in this publication. This includes, but is not limited to, loss of revenue or profit, or other incidental, physical, or consequential damages arising out of the use of this Work.

The sole remedy in the event of a claim of any kind is expressly limited to replacement of the book and/or disc, and only at the discretion of the Publisher. The use of "implied warranty" and certain "exclusions" vary from state to state, and might not apply to the purchaser of this product.

Companion files for this title are available by writing to the publisher at info@merclearning.com.

Natural Language Processing Using R

Pocket Primer

Oswald Campesato

Mercury Learning and Information
Dulles, Virginia
Boston, Massachusetts
New Delhi

Copyright ©2022 by Mercury Learning and Information LLC. All rights reserved.

This publication, portions of it, or any accompanying software may not be reproduced in any way, stored in a retrieval system of any type, or transmitted by any means, media, electronic display or mechanical display, including, but not limited to, photocopy, recording, Internet postings, or scanning, without prior permission in writing from the publisher.

Publisher: David Pallai

Mercury Learning and Information
22841 Quicksilver Drive
Dulles, VA 20166
info@merclearning.com
www.merclearning.com
800-232-0223

O. Campesato. *Natural Language Processing Using R Pocket Primer*.
ISBN: 978-1-68392-730-3

The publisher recognizes and respects all marks used by companies, manufacturers, and developers as a means to distinguish their products. All brand names and product names mentioned in this book are trademarks or service marks of their respective companies. Any omission or misuse (of any kind) of service marks or trademarks, etc. is not an attempt to infringe on the property of others.

Library of Congress Control Number: 2021950959
222324321 This book is printed on acid-free paper in the United States of America.

Our titles are available for adoption, license, or bulk purchase by institutions, corporations, etc. For additional information, please contact the Customer Service Dept. at 800-232-0223(toll free).

All of our titles are available in digital format at *academiccourseware.com* and other digital vendors. Companion files (figures and code listings) for this title are available by contacting info@merclearning.com. The sole obligation of Mercury Learning and Information to the purchaser is to replace the disc, based on defective materials or faulty workmanship, but not based on the operation or functionality of the product.

I'd like to dedicate this book to my parents –
may this bring joy and happiness into their lives.

CONTENTS

Preface *vii*

Chapter 1: Introduction to R **1**

What is R? 1
Features of R 2
Installing R, RStudio, and RStudio Cloud 2
Variable Names, Operators, and Data Types in R 2
Assigning Values to Variables in R 3
Operators in R 3
Data Types in R 3
Working with Strings in R 4
Uppercase and Lowercase Strings 4
Other String-Related Tasks 5
Working with Vectors in R 6
Finding NULL Values in a Vector in R 9
Updating NA Values in a Vector in R 10
Sorting a Vector of Elements in R 10
Working with the Built-in Letters Variable in R 11
Working with Lists in R 12
Useful Vector-Related Functions in R 15
Working with Matrices in R (1) 16
Working with Matrices in R (2) 19
Working with Matrices in R (3) 21
Working with Matrices in R (3) 22
Working with Matrices in R (4) 23
Updating Matrix Elements 24
Logical Constraints and Matrices 25
Assigning Values to Matrix Elements 25
Working with Matrices in R (5) 26

Working with Dates in R 27
The seq Function in R 28
Summary 30

Chapter 2: Loops, Conditional Logic, and Dataframes 31

Working with Simple Loops in R 31
Working with Other Types of Loops in R 32
Working with Nested Loops in R 32
Working with While Loops in R 32
Working with Conditional Logic in R 33
Compound Conditional Logic 34
Check if a Number is Prime in R 34
Check if Numbers in an Array are Prime in R 36
Check for Leap Years in R 37
Well-formed Triangle Values in R 37
What are Factors in R? 38
What are Data Frames in R? 39
Working with Dataframes in R (1) 41
Working with Data Frames in R (2) 42
Working with Data Frames in R (3) 43
Working with Data Frames in R (4) 45
Working with Data Frames in R (5) 46
Reading Excel Files in R 47
Reading SQLITE Tables in R 48
Reading Text Files in R 49
Saving and Restoring Objects in R 50
Data Visualization in R 51
Working with Bar Charts in R (1) 52
Working with Bar Charts in R (2) 53
Working with Line Graphs in R (1) 53
Working with Line Graphs in R (2) 54
Working with Multi-Line Graphs in R 55
Working with Histograms in R 56
Working with Scatter Plots in R (1) 57
Working with Scatter Plots in R (2) 58
Working with Box Plots in R 59
Working with Pie Charts in R (1) 60
Working with Pie Charts in R (2) 61
Summary 62

Chapter 3: Working with Functions in R 63

NaN and Functions in R 63
Math-Related Functions in R 65
String-Related Functions in R 66
The gsub() Function in R 67

Miscellaneous Built-in Functions 68
Set Functions in R 69
The "Apply" Family of Built-in Functions 70
The "Must Learn" dplyr Package in R 72
Other Useful R Packages 74
The Pipe Operator %>% 75
Working with CSV Files in R 77
Working with XML in R 78
Reading an XML Document into an R Dataframe 80
Working with JSON in R 81
Reading a JSON File into an R Dataframe 82
Statistical Functions in R 83
Summary Functions in R 84
Defining a Custom Function in R 85
Recursion in R 86
Calculating Factorial Values in R (non-recursive) 87
Calculating Factorial Values in R (recursive) 87
Calculating Fibonacci Numbers in R (non-recursive) 88
Calculating Fibonacci Numbers in R (recursive) 89
Convert a Decimal Integer to a Binary Integer in R 90
Calculating the GCD of Two Integers in R 91
Calculating the LCM of Two Integers in R 92
Summary 92

Chapter 4: NLP Concepts (I) 93

What is NLP? 94
    The Evolution of NLP 95
A Wide-Angle View of NLP 96
    NLP Applications and Use Cases 96
    NLU and NLG 97
    What is Text Classification? 98
Information Extraction and Retrieval 99
Word Sense Disambiguation 99
NLP Techniques in ML 100
    NLP Steps for Training a Model 101
Text Normalization and Tokenization 101
    Word Tokenization in Japanese 102
    Text Tokenization with Unix Commands 104
Handling Stop Words 104
What is Stemming? 105
    Singular vs. Plural Word Endings 105
    Common Stemmers 105
    Stemmers and Word Prefixes 106
    Over Stemming and Under Stemming 106
What is Lemmatization? 107

Stemming/Lemmatization Caveats 107
Limitations of Stemming and Lemmatization 107
Working with Text: POS 108
POS Tagging 108
POS Tagging Techniques 109
Working with Text: NER 109
Abbreviations and Acronyms 110
NER Techniques 110
What is Topic Modeling? 111
Keyword Extraction, Sentiment Analysis, and Text Summarization 112
Summary 113

Chapter 5: NLP Concepts (II) 115

What is Word Relevance? 115
What is Text Similarity? 116
Sentence Similarity 117
Sentence Encoders 117
Working with Documents 117
Document Classification 117
Document Similarity (doc2vec) 118
Techniques for Text Similarity 118
Similarity Queries 119
What is Text Encoding? 119
Text Encoding Techniques 120
Document Vectorization 120
One-Hot Encoding (OHE) 121
Index-Based Encoding 122
Additional Encoders 122
The BoW Algorithm 123
What are N-grams? 124
Calculating Probabilities with n-grams 125
Calculating tf, idf, and tf-idf 127
What is Term Frequency (TF)? 127
What is Inverse Document Frequency (IDF)? 128
What is tf-idf? 129
Limitations of tf-idf 130
What is BM25? 131
Pointwise Mutual Information (PMI) 132
The Context of Words in a Document 132
What is Semantic Context? 132
Textual Entailment 133
Discrete, Distributed, and Contextual Word Representations 133
What is Cosine Similarity? 133
Text Vectorization (aka Word Embeddings) 135
Overview of Word Embeddings and Algorithms 136

Word Embeddings 137
Word Embedding Algorithms 137
What is word2vec? 138
The Intuition for word2vec 139
The word2vec Architecture 140
Limitations of word2vec 140
The CBoW Architecture 140
What are Skip-grams? 141
An Example of Skip-grams 142
The Skip-gram Architecture 142
Neural Network Reduction 144
What is GloVe? 144
Working with GloVe 145
What is fastText? 146
Comparison of Word Embeddings 146
What is Topic Modeling? 147
Topic Modeling Algorithms 147
LDA and Topic Modeling 147
Text Classification vs Topic Modeling 149
Language Models and NLP 149
How to Create a Language Model 149
Vector Space Models 150
Term-Document Matrix 151
Tradeoffs of the VSM 151
NLP and Text Mining 152
Text Extraction Preprocessing and N-Grams 152
Relation Extraction and Information Extraction 152
What is a BLEU Score? 153
ROUGE Score: An Alternative to BLEU 153
Summary 154

Chapter 6: NLP in R 155

Launch R Scripts from the Command Line 156
Installing RStudio Packages 158
NLP Packages in R 159
Common Tasks for Cleaning NLP Datasets 160
Does the Language Make a Difference? 160
Cleaning NLP Data in R 161
Tokenization 161
Remove Punctuation in Strings 161
Convert Strings to Lowercase and Uppercase 162
Convert File Data to Lowercase and Uppercase 163
Stop Words 164
Stemming in R 165
Lemmatization 165

POS (Parts Of Speech) with SpaCy in R 167
POS in R 168
NER in R 170
The tf-idf Algorithm 171
Working with N-Grams 174
Topic Modeling in R 176
Working With word2vec in R 177
Summary 178

Chapter 7: Transformer, BERT, and GPT 179
What is Attention? 180
Types of Word Embeddings 180
Types of Attention and Algorithms 181
An Overview of the Transformer Architecture 182
The Transformers Library from HuggingFace 182
Transformer and NER Tasks 183
Transformer and QnA Tasks 184
Transformer and Sentiment Analysis Tasks 185
Transformer and Mask Filling Tasks 185
What is T5? 186
What is BERT? 187
BERT Features 187
How is BERT Trained? 187
How BERT Differs from Earlier NLP Models 188
The Inner Workings of BERT 188
What is MLM? 188
What is NSP? 188
Special Tokens 189
BERT Encoding: Sequence of Steps 190
Subword Tokenization 192
Sentence Similarity in BERT 194
Word Context in BERT 194
Generating BERT Tokens (1) 196
Generating BERT Tokens (2) 197
The BERT Family 198
Surpassing Human Accuracy: deBERTa 200
What is Google Smith? 200
Introduction to GPT 200
Installing the Transformers Package 201
Working with GPT-2 201
GPT-2 versus BERT 207
What is GPT-3? 207
GPT-3 Task Strengths and Mistakes 208
GPT-3 Architecture 208
The GPT-3 Playground 208

Accessing the GPT-3 Playground 209
What is the Goal of GPT-3? 209
Zero-Shot, One-Shot, and Few Shot Learners 210
GPT-3 Task Performance 210
The Switch Transformer: One Trillion Parameters 211
Looking Ahead 211
Summary 212

Appendix: Intro to Probability and Statistics 213

Index 241

PREFACE

What Is the Value Proposition for This Book?

This book contains a fast-paced introduction to as much relevant information about NLP using R that can be reasonably included in a book of this size. Some chapters contain topics that are discussed in great detail with many code samples, whereas other chapters contain theoretical foundations of NLP concepts (such as Chapter 4).

This book helps developers who have a wide range of technical backgrounds, which is the rationale for the inclusion of a plethora of topics. Regardless of your background, please remember the following point: *this book is essentially a stepping stone for your study of NLP.*

You will be exposed to various NLP and machine learning topics in this book, some of which are presented in a cursory manner for two reasons. First, it's important that you be exposed to these concepts. In some cases, you will find topics that might pique your interest, and hence motivate you to learn more about them through self-study; in other cases, you will probably be satisfied with a brief introduction. Hence, you can decide whether to delve into more detail regarding the topics in this book.

Second, a full treatment of all the topics that are covered in this book would probably triple its page count, and few people are interested in reading long technical books. Hence, this book provides a decent view of the NLP and machine learning landscape, based on the belief that this approach will be more beneficial for readers who are experienced developers who want to learn about NLP and machine learning.

The Target Audience

This book is intended primarily for people who have a solid background as software developers. Specifically, this book is for developers who are

accustomed to searching online for more detailed information about technical topics. If you are a beginner, there are other books that are more suitable for you, and you can find them by performing an online search.

This book is also intended to reach an international audience of readers with highly diverse backgrounds in various age groups. This book uses standard English rather than colloquial expressions that might be confusing to those readers. People learn in different ways, which includes reading, writing, or hearing new material. This book tries to take these approaches into consideration to provide a comfortable and meaningful learning experience for the intended readers.

Do I Need to Learn the Theory Portions of This Book?

Once again, the answer depends on the extent to which you plan to become involved in NLP and machine learning. In addition to creating a model, you will use algorithms to see which ones provide the level of accuracy (or some other metric) that you need for your project. The theoretical aspects of machine learning can help you perform a forensic analysis of your model and your data, and ideally assist in determining how to improve your model.

Why is a Python-based Chapter in This Book?

Chapter 7 is devoted to the Transformer architecture, the BERT model, and GPT-related models. The reason for the inclusion of Python-based code samples in this chapter is simple: there is a plethora of Python-based code available to illustrate how to use the NLP-related APIs of these models, whereas R-based code samples are typically unavailable. Most of the code samples in Chapter 7 require Python 3.7.

In addition, many of the R-based code samples in Chapter 6 are wrappers around Python-based code, which will necessitate installing Python 3 and other Python-based NLP libraries. The installation details are provided in Chapter 6.

Getting the Most From This Book

Some programmers learn well from prose and others learn well from sample code (and lots of it), which means that there's no single style that can be used for everyone.

Moreover, some programmers want to run the code first, see what it does, and then return to the code to delve into the details (and others use the opposite approach).

Consequently, there are various types of code samples in this book: some are short, some are long, and other code samples build on earlier code samples.

What Do I need to Know for This Book?

Although this book is introductory in nature, some knowledge of R for the first three chapters is helpful. In addition, some knowledge of Python 3.x for the code samples in Chapter 7 would also be helpful. Knowledge of other programming languages (such as Java) can also be helpful because of the exposure to programming concepts and constructs. The less technical knowledge that you have, the more diligence will be required to understand the various topics that are covered.

If you want to be sure that you can grasp the material in this book, glance through some of the code samples to get an idea of how much is familiar to you and how much is new for you.

Does This Book Contain Production-Level Code Samples?

The code samples in this book are for basic NLP tasks. The primary purpose of the code samples is to show you how to solve various NLP-related tasks, some of which are performed in conjunction with machine learning. Moreover, clarity has a higher priority than writing more compact code that is more difficult to understand (and possibly more prone to bugs). If you decide to use any of the code in this book in a production website, you should subject that code to the same rigorous analysis as the other parts of your code base.

What Are the Non-Technical Prerequisites for This Book?

Although the answer to this question is more difficult to quantify, it's important to have strong desire to learn about NLP, along with the motivation and discipline to read and understand the code samples. As a reminder, even simple machine language APIs can be a challenge to understand the first time you encounter them, so be prepared to read the code samples several times.

How Do I Set up a Command Shell?

If you are a Mac user, there are three ways to do so. The first method is to use `Finder` to navigate to `Applications > Utilities` and then double click on the `Utilities` application. Next, if you already have a command shell available, you can launch a new command shell by typing the following command:

```
open /Applications/Utilities/Terminal.app
```

A second method for Mac users is to open a new command shell on a Macbook from a command shell that is already visible simply by clicking `command+n` in that command shell, and your Mac will launch another command shell.

If you are a PC user, you can install Cygwin (open source at *https://cygwin.com/*) that simulates bash commands or use another toolkit such as MKS (a commercial product). Please read the online documentation that describes the download and installation process. Note that custom aliases are not automatically set if they are defined in a file other than the main start-up file (such as `.bash_login`).

Companion Files

All the code samples and figures in this book may be obtained by writing to the publisher at *info@merclearning.com*.

What Are the "Next Steps" After Finishing This Book?

The answer to this question varies widely, mainly because the answer depends heavily on your objectives. If you are interested primarily in NLP, you can learn more advanced concepts, such as attention, transformers, and the BERT-related models.

If you are primarily interested in machine learning, there are some subfields of machine learning, such as deep learning and reinforcement learning (and deep reinforcement learning) that might appeal to you. Fortunately, there are many resources available, and you can perform an Internet search for those resources. One other point: the aspects of machine learning for you to learn depend on who you are. The needs of a machine learning engineer, data scientist, manager, student, or software developer are all different.

O. Campesato
January 2022

CHAPTER 1

INTRODUCTION TO R

This chapter provides a quick introduction to `R` programming, with code samples that illustrate some basic features of `R`. If you are already familiar with `R`, you can probably skim this chapter, just to be sure that you're acquainted with the code and concepts in this chapter.

The first section starts with a brief description of some features of `R`, followed by a description of valid variable names, operators, data types. You will also learn how to perform various simple string-related tasks.

The second section discusses vectors, with an assortment of code samples, followed by a section that discusses lists in `R`. The third section discusses matrices and how to manipulate them, and this is followed by an explanation of the `R seq` data type.

This chapter contains an assortment of code samples that show you the flexibility of `R` with respect to defining variables with heterogeneous data (i.e., mixed data types). Although you might not need to use all of the features that are illustrated in the code samples, read the code samples to become aware of those features.

WHAT IS R?

R is a popular programming language. You can create R scripts with R commands that you can launch from the command line, or you can launch R commands inside RStudio. R provides a convenient way to perform statistical analysis and reporting operations. In general, if you can think of a feature that you need, it probably already exists in R, and if not, you can go to CPAN (which contains more than 14,000 modules for R) to install a library that supports that feature.

Features of R

R supports Boolean logic and programmatic constructs, such as loops and functions, along with support for recursion. R also supports various data types, such as arrays, lists, vectors, and matrices.

One highly useful data type is a *data frame*, which is comparable to data frames in Pandas (a powerful Python module), both of which are analogous to spreadsheets.

In addition, R provides support for many statistical distributions, as well as support for charts and graphs.

Installing R, RStudio, and RStudio Cloud

The download links for R depend on your platform. For example, you can download R for MacBooks from the following link:

https://cran.r-project.org/bin/macosx/

Fortunately, the following link enables you to download and install RStudio for your platform:

https://www.rstudio.com/products/rstudio/download/

After downloading the appropriate distributions, follow the prompts for the installation of R and RStudio for your platform.

One detail to keep in mind is that the R code samples in this book are executed from the command line via the `rscript` utility. However, if you are more comfortable working in an IDE, then RStudio is an excellent alternative to the command line.

An optional installation is RStudio Cloud, which has a free tier as well as various paid tiers, and you can register for a free online account here:

https://rstudio.cloud

RStudio Cloud provides "Studio Primers", which is a collection of online tutorials that show you how to visualize data, perform table-related tasks, perform data visualization with `ggplot2`, work with `shiny`, and various other tasks.

VARIABLE NAMES, OPERATORS, AND DATA TYPES IN R

A valid variable name in R is a combination of letter, numbers, a period (.), or an underscore (_). In addition, a valid variable name starts with a letter or period, and must not be followed by a number. For example, the following names are valid in R:

```
var_name2.
.var_name
var.name
```

However, the following names are not valid in R:

```
var_name%  (contains a % symbol)
2var_name  (starts with a number)
.2var_name (dot followed by a num)
_var_name  (starts with "_")
```

Assigning Values to Variables in R

There are three ways to assign values to variables:

- using the equals operator: `var.1 = c(0,1,2,3)`
- using the leftward operator: `var.2 <- c('learn', ,coding)`
- using the rightward operator: `(c(TRUE, 1) -> var.3`

Don't worry if some of the preceding assignments aren't clear right now. Later you will see examples of all three assignments.

Operators in R

R supports arithmetic, relational, and logical operators, along with some "miscellaneous" operators that you will learn later in this book. The `R` operators are shown below:

- arithmetic operators in R: `+,-,/,*,%%,%/%,^`
- relational operators in R: `<, <=, >, >=, ==, !=`
- logical operators in R: `&, |, !, &&, ||`
- miscellaneous operators in R: `%in%, %>%, %*%`

Data Types in R

Although R does not have a set of data types that are as extensive as some other programming languages, the data types that it does support enable you to solve a wide variety of tasks:

- Vectors (homogeneous)
- Lists (heterogeneous)
- Matrices (two-dimensional)
- Arrays (multi-dimensional)
- Factors (similar to `enum`)
- Data Frames
- Series

The following interactive session shows you some examples of working with numbers and strings in R:

```
#1: sqrt(500)
#2:
a <- 500
#3:
a <- as.character(a) print() (a)

a <- Hello, b <- , How,
c <- "are you? "
print(paste(a,b,c))
print(paste(a,b,c, sep = "-"))
print(paste(a,b,c, sep = "", collapse = ""))
```

R supports formatting of numbers and strings, as shown below:

```
# Formatting Numbers/Strings
# last digit rounded off:
result <- format(23.123456789, digits = 9) print(result)
# scientific notation:
result <- format(c(6, 13.14521), scientific = TRUE)
print(result)
# minimum # of digits to the right of the decimal point:
result <- format(23.47, nsmall = 5)
print(result)
```

WORKING WITH STRINGS IN R

Listing 1.1 displays the content of `strings1.R` that illustrates how to initialize simple variables as strings and how to print them in R.

LISTING 1.1 strings1.R

```
a <-  "Hello"
b <-  "How"
c <-  "are you? "

print(paste(a,b,c))
print(paste(a,b,c, sep = "-"))
print(paste(a,b,c, sep = "", collapse = ""))
```

Listing 1.1 initializes the variables `a`, `b`, and `c` with the strings `Hello`, `How`, and `are you?`, respectively. Next, a `print()` statement prints the result of "pasting" or concatenating the values of `a`, `b`, and `c`.

The second `print()` statement is similar, but with a hyphen (-) as a separator. The third `print()` statement is similar to the second, except that no character is used as a separator. Launch the code in Listing 1.1 to see the following output:

```
[1] "Hello How are you? "
[1] "Hello-How-are you? "
[1] "HelloHoware you? "
```

Uppercase and Lowercase Strings

Listing 1.2 shows the content of `UpperLower.R` that illustrates how to convert text strings to uppercase and lowercase letters, respectively, with the `uppercase()` and `lowercase()` functions in R.

LISTING 1.2 UpperLower.R

```
result <- nchar("Count the number of characters")
print(result)

# Upper case:
result <- toupper("Changing To Upper")
print(result)
```

```
# Lower case:
result <- tolower("Changing To Lower")
print(result)

# Extract 5th to 7th positions:
result <- substring("HelloWorld", 5, 7)
print(result)
```

Listing 1.2 starts by initializing the variable `result` with the number of characters in a text string, calculating with the built-in R function `nchar()`. Next, `result` is initialized with a character string that is converted to uppercase, and then initialized again with a character string that has been converted to lowercase.

Finally, `result` is initialized with the substring from positions 5 through 7 inclusive (which corresponds to the index values 4 through 6 inclusive) of the string `HelloWorld`. A `print()` statement displays the value of `result` after each initialization. Launch the code in Listing 1.2 to see the following output:

```
[1] 30
[1] "CHANGING TO UPPER"
[1] "changing to lower"
[1] "oWo"
```

Other String-Related Tasks

The previous sections showed you various string-related functions for detecting uppercase letters and lowercase letters, and how to perform case-based conversions. There are other string-related tasks that are easy to perform with R built-in functions, some of which are listed here:

- Given a string, find the number of blanks
- Given a string, find the number of non-blanks
- Given a string, find the number of characters
- Given a string, find the number of digits
- Reverse a string (a vector of strings)

Listing 1.3 displays the content of `string_tasks.R` that illustrates how to perform the tasks in the preceding list. This code sample is intended to pique your interest: it's a preview of several useful R functions (shown in bold), some of which are discussed further in Chapter 3.

LISTING 1.3: string_tasks.R

```
#Given a string find the number of blanks
#Given a string find the number of non-blanks
#Given a string find the number of characters
#Given a string find the number of digits

str <- "I love deep dish pizza 2day and 3morrow!"
blank_count  = length(gregexpr(" ", str)[[1]])
```

```
str_length1  = length(str)
str_length2  = nchar(str) # also works with numbers
non_blanks   = str_length2 - blank_count
digit_count1 = nchar(gsub("[^0-9]+", "", str))
digit_count2 = nchar(gsub("\\D", "", str))

print(paste0("Original string:  ",str))
print(paste0("count of blanks:  ",blank_count))
print(paste0("Non-blanks:       ",non_blanks))
print(paste0("String length #1: ",str_length1))
print(paste0("String length #2: ",str_length2))
print(paste0("digit count #1:   ",digit_count1))
print(paste0("digit count #2:   ",digit_count2))

tokens1 = strsplit(str, " ")[[1]]
tokens2 = strsplit(str, " ")
print(paste0("Tokens #1:        ",tokens1))
print(paste0("Tokens #2:        ",tokens2))
```

Listing 1.3 starts by initializing the variable `str` as a text string, followed by initializing `blank_count` with the number of characters in `str`, based on a combination of the built-in `length()` function and the `gregexpr()` function.

The next code snippet initializes `str_length2` with the number of characters in the `str` variable, and then sets `non_blank` equal to the number of non-blank characters in `str`.

The next pair of code snippets calculates the number of digits and non-digits in `str`. The final section in Listing 1.3 displays the values of the preceding variables and performs calculations using the built-in R function `nchar()`. Launch the code in Listing 1.3 to see the following output:

```
[1] "Original string:  I love deep dish pizza 2day and 3morrow!"
[1] "count of blanks:  7"
[1] "Non-blanks:       33"
[1] "String length #1: 1"
[1] "String length #2: 40"
[1] "digit count #1:   2"
[1] "digit count #2:   2"
[1] "Tokens #1:        I"       "Tokens #1:       love"
[3] "Tokens #1:        deep"    "Tokens #1:       dish"
[5] "Tokens #1:        pizza"   "Tokens #1:       2day"
[7] "Tokens #1:        and"     "Tokens #1:       3morrow!"
[1] "Tokens #2:        c(\"I\", \"love\", \"deep\", \"dish\",
\"pizza\", \"2day\", \"and\", \"3morrow!\")"
```

The next section introduces you to vectors in R and how to initialize them with values and display their contents in R.

WORKING WITH VECTORS IN R

A *vector* in R is a one-dimensional variable. For example, [3] is a 1 × 1 vector with a single integer value, and [2 –4 8 15] is a 1 × 4 vector of integers.

A simple way to create a vector is with the built-in `c` ("concatenate") function. Listing 1.4 displays the content of `VectorStuff.R`, which uses simple operations with vectors.

LISTING 1.4: VectorStuff.R

```
y =  c(10,20,30,40,50)
print("y:")
print(y)

y =  c(1,2,3)
print("y:")
print(y)

x <- c(10,20,30,40,50)
print("x:")
print(x)

print(paste0("x[2]:       ",x[2]))
print(paste0("length:     ",length(x)))
print(paste0("typeof(x): ",typeof(x)))
print(paste0("x:          ",x))
```

Listing 1.4 invokes the built-in R function `c()` to initialize the vector `y` as a vector of 5 integers (`c` for *concatenate*), and the `print()` statement displays the contents of the vector `y`. Next, the vector `y` is initialized to a vector containing three integers, and then its values are displayed.

Notice that the variable `x` is initialized as a vector of five integers via the built-in `c()` function, this time with a `<-` symbol instead of an equals (=) symbol. Although the `<-` symbol is preferred among R aficionados, you can also use an equals (=) symbol. The white space is important, as shown in the following code snippets in which the first is an assignment and the second is a comparison:

```
x<-7
x < -7
```

The next portion of Listing 1.4 displays the third element (index 2) of `x`, the length of `x`, and the type of `x`. Launch the code in Listing 1.4 to see the following output (notice the last output line):

```
[1] "y:"
[1] 10 20 30 40 50
[1] "y:"
[1] 1 2 3
[1] "x:"
[1] 10 20 30 40 50
[1] "x[2]:      20"
[1] "length:    5"
[1] "typeof(x): double"
[1] "x:         10" "x:       20" "x:       30" "x:       40"
[5] "x:         50"
```

Listing 1.5 displays the content of `VectorStuff2.R` that shows you additional simple operations with vectors.

LISTING 1.5: VectorStuff2.R

```
v <- c(3,8,4,5,0,11, -9, 304)
v <- c(1,2,3,4,0,-1,-2)

# Sort the elements of the vector:
sort.result <- sort(v)
print(paste0("v: ",v))
print(paste0("sorted v: ",sort.result))

#mixed1 <- c(6, a, 7, b, 8)
#print(paste0("mixed1: ",mixed1))
#print(paste0("class:  ",class(mixed1)))

ul_chars <- character(4)
print(paste0("ul_chars: ",ul_chars))
ul_chars[1] <- "A"
print(paste0("ul_chars: ",ul_chars))

names <- c("dave", "stella", "ralph", "john")
print(paste0("names:      ",names))
print(paste0("length:     ",length(names)))
print(paste0("names[1:2]: ",names[1:2]))
print(paste0("3,4,1,2:    ",names[3:4], names[1:2]))

x <- c(1,2,3,4,5,6)
print(paste0("x[2]:    ",x[2]))
print(paste0("x[8]:    ",x[8]))
print(paste0("x[-3]:   ",x[-3]))
print(paste0("x[2:4]:  ",x[2:4]))

x1 <- c(1,2,3,4)
y1 <- c(4,5,6,7)
print(paste0("x1+y1:   ",x1+y1))

x2 <- c(1,2,3,4)
y2 <- c(4,5)
print(paste0("x2+y2:   ",x2+y2))
print(paste0("x2-y2:   ",x2-y2))
print(paste0("x2*y2:   ",x2*y2))
```

Listing 1.5 initializes `v` as a vector of four integers and displays `v`, then sorts the vector `v` and displays the sorted result. Next, the variable `ul_chars` is initialized as a string of length four, and then the first character is initialized with the letter `A`.

Next, the variable `names` is initialized with four names (i.e., strings), and various operations are performed to find its length, display the names in positions 1 and 2, and then change the initial ordering to 3, 4, 1, 2 (and display the new ordering of `names`).

The next portion of Listing 1.5 initializes `x` as a vector of integers and shows you various operations you can perform on `x`, such as the elements of `x` that are in position 2, 8, –3, and in the range from 2 to 4.

Launch the code in Listing 1.5 to see the following output:

```
[1] "v: 1"  "v: 2"  "v: 3"  "v: 4"  "v: 0"  "v: -1" "v: -2"
[1] "sorted v: -2" "sorted v: -1" "sorted v: 0"  "sorted v: 1"  "sorted v: 2"
[6] "sorted v: 3"  "sorted v: 4"
[1] "ul_chars: " "ul_chars: " "ul_chars: " "ul_chars: "
[1] "ul_chars: A" "ul_chars: "  "ul_chars: "  "ul_chars: "
[1] "names:      dave"   "names:      stella" "names:      ralph"
[4] "names:      john"
[1] "length:     4"
[1] "names[1:2]: dave"   "names[1:2]: stella"
[1] "3,4,1,2:    ralphdave"  "3,4,1,2:    johnstella"
[1] "x[2]:    2"
[1] "x[8]:    NA"
[1] "x[-3]:   1" "x[-3]:   2" "x[-3]:   4" "x[-3]:   5" "x[-3]:   6"
[1] "x[2:4]:  2" "x[2:4]:  3" "x[2:4]:  4"
[1] "x1+y1:   5"  "x1+y1:   7"  "x1+y1:   9"  "x1+y1:   11"
[1] "x2+y2:   5" "x2+y2:   7" "x2+y2:   7" "x2+y2:   9"
[1] "x2-y2:   -3" "x2-y2:   -3" "x2-y2:   -1" "x2-y2:   -1"
[1] "x2*y2:   4"  "x2*y2:   10" "x2*y2:   12" "x2*y2:   20"
```

Finding NULL Values in a Vector in R

Listing 1.6 shows the content of `simple_vector.R` that illustrates how to initialize a vector with numbers and an `NA` value in R.

LISTING 1.6: simple_vector1.R

```
v <- c(1,2,NA,4)

print("v:")
print(v)

print("length of v:")
print(length(v))

print("null values in v:")
print(is.na(v))

print("numeric values in v:")
print(is.numeric(v))
```

Listing 1.6 defines vector `v`, which contains three integers and an `NA` value. Launch the code in Listing 1.6 to see the following output:

```
[1] "v:"
[1]  1  2 NA  4
[1] "length of v:"
[1] 4
[1] "null values in v:"
[1] FALSE FALSE  TRUE FALSE
[1] "numeric values in v:"
[1] TRUE
```

Updating NA Values in a Vector in R

Listing 1.7 shows the content of `missing_mean.R` that illustrates how to replace `NA` values with the mean of the non-null values of a vector in R.

LISTING 1.7: missing_mean.R

```
print("Initial contents of v1:")
v1 <- c(1,2,NA,4)
print(v1)

print("Updated v2:")
v2 <- replace(v1, is.na(v1), mean(v1, na.rm = TRUE))
print(v2)

print("-----------------------")
print("Initial contents of v3:")
v3 <- c(1,2,NA,4,NA,5,6)
print(v3)

print("Updated v4:")
v4 <- replace(v3, is.na(v3), mean(v3, na.rm = TRUE))
print(v4)
```

Listing 1.7 defines vector `v` that contains three integers and an `NA` value, after which the `NA` value in `v` is replaced with the mean value of the numbers in `v`. Launch the code in Listing 1.7 to see the following output:

```
[1] "Initial contents of v1:"
[1]  1  2 NA  4
[1] "Updated v2:"
[1] 1.000000 2.000000 2.333333 4.000000
[1] "-----------------------"
[1] "Initial contents of v3:"
[1]  1  2 NA  4 NA  5  6
[1] "Updated v4:"
[1] 1.0 2.0 3.6 4.0 3.6 5.0 6.0
```

Sorting a Vector of Elements in R

Listing 1.8 shows the content of `sorting1.R` that illustrates how to sort a vector of numbers and a vector of strings in R.

LISTING 1.8: sorting1.R

```
v <- c(13,8,44,5,0,-1,-3,-2)

# Sort the elements of the vector:
sort.result <- sort(v)
print(sort.result)

# Sort in reverse order:
revsort.result <- sort(v, decreasing = TRUE)
print(revsort.result)
```

```
# Sorting character vectors:
v <- c("Red","Blue","yellow","violet")
sort.result <- sort(v)
print(sort.result)
```

Listing 1.8 defines the vector `v` that contains 8 integer values and then sorts these values. The next code snippet sorts the numbers in `v` in decreasing order (i.e., from the largest to smallest values).

The final code snippet initializes the vector `v` with a set of strings and then sorts them in alphabetical order. Launch the code in Listing 1.8 to see the following output:

```
[1] -3 -2 -1  0  5  8 13 44
[1] 44 13  8  5  0 -1 -2 -3
[1] "Blue"   "Red"    "violet" "yellow"
```

Working with the Built-in Letters Variable in R

Listing 1.9 shows the content of `alphabet.R` that illustrates the built-in variable `letters` in R.

LISTING 1.9: alphabet.R

```
# The "letters" vector is a built-in vector in R
print(paste0("letters: ",letters))

# displays the letters in a consecutive fashion:
print(letters)

# extract first 5 letters (comma-separated):
first5 <- paste0(letters[1:5], collapse=",")
print(first5)
```

Listing 1.9 shows the content of the built-in variable `letters`, which contains the lowercase letters of the English alphabet. The second `print()` statement displays the letters of the alphabet, separated by a white space. Finally, the variable `first5` is initialized with the first five letters in alphabet. Launch the code in Listing 1.9 to see the following output:

```
 [1] "letters: a" "letters: b" "letters: c" "letters: d" "letters: e"
 [6] "letters: f" "letters: g" "letters: h" "letters: i" "letters: j"
[11] "letters: k" "letters: l" "letters: m" "letters: n" "letters: o"
[16] "letters: p" "letters: q" "letters: r" "letters: s" "letters: t"
[21] "letters: u" "letters: v" "letters: w" "letters: x" "letters: y"
[26] "letters: z"
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
[20] "t" "u" "v" "w" "x" "y" "z"
 [1] "a,b,c,d,e"
```

WORKING WITH LISTS IN R

A *list* in R can contain a heterogeneous set of values whereas vectors in R must contain values of the same data type (which might be converted implicitly). Moreover, a vector data type is one-dimensional, whereas a list data type is a multidimensional object.

Listing 1.10 shows the content of `ListOperations1.R` that illustrates an assortment of list-related operations in R.

LISTING 1.10: ListOperations1.R

```
a <- "abc"
b <- "zzz"
list1 <- c(a, seq(1,3)) # seq() is discussed later
list1[2]
list2 <- c(b, seq(1,10, by=3))
list2[2:3]

list3 <- list2[!is.na(list2)]
list3[1]
list3[!is.na(list3)]
samples1 <- sample(1:50, replace=TRUE)
class(samples1)
list2 <- c(b, seq(1,10), by=3)

#Naming List Elements
# Create a list of a vector, a matrix and a list:
list_data <- list(c("Jan","Feb","Mar"),
matrix(c(3,9,5,1,-2,8), nrow = 2), list("green",12.3))

# Name the elements of the list:
names(list_data) <- c("1st Quarter", "A_Matrix", "An Inner list")

# display the list:
print(list_data)
```

Listing 1.10 initializes the variables `a` and `b`, followed by the variable `list1` that consists of the contents of the variable `a`, followed by the integers 1, 2, and 3. Next, the expression `list1[2]` displays the contents of the second element of `list1`, which is the value 1.

The next portion of Listing 1.10 initializes the variable `list2` that consists of the contents of the variable `b`, followed by the integers 1, 4, 7, and 10. Next, the expression `list2[2:3]` displays the contents of the second and third elements of `list2`, which are the values 1 and 4.

The next portion of Listing 1.10 initializes the variable `list3` that consists of the elements of the variable `list2` that are not integers, which is the value `zzz`. Next, the variable `samples1` is initialized with the first 50 integers, and its data type is displayed, which is `integer`.

The next portion of Listing 1.10 initializes the variable `list_data` with three components (a vector, a matrix, and a list), as shown here:

```
list_data <- list(c("Jan","Feb","Mar"),
matrix(c(3,9,5,1,-2,8), nrow = 2), list("green",12.3))
```

The final portion of Listing 1.10 initializes the names of the elements of the variable `list_data` with three strings, as shown here:

```
names(list_data) <- c("1st Quarter", "A_Matrix", "An Inner
list")
```

Launch the code in Listing 1.10 to see the following output:

```
[1] "1"
[1] "1" "4"
[1] "zzz"
[1] "zzz" "1"   "4"   "7"   "10"
[1] "integer"
$'1st Quarter'
[1] "Jan" "Feb" "Mar"

$A_Matrix
     [,1] [,2] [,3]
[1,]    3    5   -2
[2,]    9    1    8

$'A Inner list'
$'A Inner list'[[1]]
[1] "green"

$'A Inner list'[[2]]
[1] 12.3
```

Listing 1.11 shows the content of `ListOperations2.R` that illustrates an assortment of list-related operations in R.

LISTING 1.11: ListOperations2.R

```
#Accessing List Elements:
# Create a list of a vector, a matrix and a list:
list_data <- list(c("Jan","Feb","Mar"),
matrix(c(3,9,5,1,-2,8), nrow = 2), list("green",12.3))

# Name the elements in the list:
names(list_data) <- c("1st Quarter", "A_Matrix", "An Inner
list")

#Accessing List Elements
# Access the first element of the list:
print(list_data[1])

# Access the 3rd element (which is also a list):
print(list_data[3])
```

```
# Access the list element using the name of the element:
print(list_data$A_Matrix)

# Merging Two Lists
# Create two lists and merge them:
list1 <- list(1,2,3)
list2 <- list("Sun","Mon","Tue")
merged.list <- c(list1,list2)

# Print() the merged list:
print(merged.list)
```

Listing 1.11 starts with the initialization of the variable `list_data`, which similar to Listing 1.10, followed by the display of the first and third elements of `list_data`. The next portion of Listing 1.11 initializes the variables `list1` and `list2` and then merges their contents. Launch the code in Listing 1.11 to see the following output:

```
$'1st Quarter'
[1] "Jan" "Feb" "Mar"

$'An Inner list'
$'An Inner list'[[1]]
[1] "green"

$'An Inner list'[[2]]
[1] 12.3

     [,1] [,2] [,3]
[1,]    3    5   -2
[2,]    9    1    8
[[1]]
[1] 1

[[2]]
[1] 2

[[3]]
[1] 3

[[4]]
[1] "Sun"

[[5]]
[1] "Mon"

[[6]]
[1] "Tue"
```

Listing 1.12 shows the content of `ListOperations3.R` that illustrates an assortment of list-related operations in R.

LISTING 1.12: ListOperations3.R

```
#Convert Lists To Vectors
list1 <- list(1:5)
print(list1)
list2 <- list(10:14)
print(list2)

# Convert the lists to vectors:
v1 <- unlist(list1)
v2 <- unlist(list2)
print(v1)
print(v2)

# Add the vectors:
result <- v1 + v2
print(result)
```

Listing 1.12 initializes the variables `list1` and `list2` with the integers from 1 to 5 and the integers from 10 to 14, respectively, and then prints their contents. Next, the variables `v1` and `v2` are initialized with the vector-based counterparts to `list1` and `list2`, respectively. The final code snippet initializes the variable result with the sum of `v1` and `v2`. Launch the code in Listing 1.12 to see the following output:

```
[[1]]
[1] 1 2 3 4 5

[[1]]
[1] 10 11 12 13 14

[1] 1 2 3 4 5
[1] 10 11 12 13 14
[1] 11 13 15 17 19
```

Useful Vector-Related Functions in R

R provides various useful vector related functions, some of which are displayed here:

- `append()`: add elements to a vector
- `cbind()`: combine vectors by row/column
- `sort(x)`: sort the vector x
- `unique(x)`: remove duplicate entries from vector

Listing 1.13 shows the content of `vector_functions.R` that illustrates how to use several of the preceding vector-related functions in R.

LISTING 1.13: vector_functions.R

```
# initialize an empty vector:
vect <- c()
print(paste0("vect: ",vect))
vect <- c(vect, 1*1)
print(paste0("vect: ",vect))
vect <- c(vect, 3*3)
print(paste0("vect: ",vect))
vect <- c(vect, 2*2)
print(paste0("vect: ",vect))
vect <- sort(vect)
print(paste0("sort: ",vect))
vect <- append(vect, 100)
print(paste0("vect: ",vect))
vect <- append(vect, 4)
print(paste0("vect: ",vect))
vect <- unique(vect)
print(paste0("vect: ",vect))
```

Listing 1.13 starts by initializing the variable `vect` as an empty vector, followed by appending the squares of the numbers 1, 3, and 2. The next code snippet sorts the elements in `vect`, and then appends the number 4. The last code snippet updates `vect` with the unique elements in `vect`. Launch the code in Listing 1.13 to see the following output:

```
[1] "vect: "
[1] "vect: 1"
[1] "vect: 1" "vect: 9"
[1] "vect: 1" "vect: 9" "vect: 4"
[1] "sort: 1" "sort: 4" "sort: 9"
[1] "vect: 1"   "vect: 4"   "vect: 9"   "vect: 100"
[1] "vect: 1"   "vect: 4"   "vect: 9"   "vect: 100" "vect: 4"
[1] "vect: 1"   "vect: 4"   "vect: 9"   "vect: 100"
```

WORKING WITH MATRICES IN R (1)

A matrix in R is a 2D rectangular dataset. Listing 1.14 shows the content of `MatrixOperations1.R` that illustrates how to use more matrix-related functions in R.

LISTING 1.14: MatrixOperations1.R

```
M = matrix(c(1,2,3,4,5,6), nrow=2,ncol=3,byrow=TRUE)
M
M[,1]
M[2:3]
W <- cbind(c(0.5,0.3),c(0.3,0.5))
W
class(W)

#Arrays multi-dimensional rectangular data sets
dim(as.array(letters))
```

```
U <- array(0, dim=c(2,2,2))
U
V <- array(1, dim=c(2,2,2,2))
V
```

Listing 1.14 initializes the 2 × 3 matrix `M` with the integers from 1 to 6 inclusive, and then displays the contents of `M`, the first column of `M`. The next code snippet initializes the array `W` with two one-dimensional vectors and displays the contents of `W` as well as the class type of `W`.

The final portion of Listing 1.14 initializes the arrays `U` and `V` as a vectors of the number 2 and then displays their contents. Launch the code in Listing 1.14 to see the following output:

```
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6

[1] 1 4
[1] 4 2
     [,1] [,2]
[1,]  0.5  0.3
[2,]  0.3  0.5
[1] "matrix"
[1] 26
, , 1

     [,1] [,2]
[1,]    0    0
[2,]    0    0

, , 2

     [,1] [,2]
[1,]    0    0
[2,]    0    0
```

Listing 1.15 shows the content of `MatrixOperations2.R` that illustrates how to use more matrix-related functions in R.

LISTING 1.15: MatrixOperations2.R

```
arr <- array(rep(1:4, each=4), dim=c(2,2,2,2))
arr
dim(arr)
vector1 <- c(5,9,3)
vector2 <- c(10,11,12,13,14,15)
array1 <- array(c(vector1,vector2),dim = c(3,3,2))

vector1
vector2
array1
```

Listing 1.15 initializes the variable `arr` as a four-dimensional array, where each of the four “slices” is a 2 × 2 array that contains the values 1, 2, 3, and 4.

The next code snippet initializes the variable `vector1` with the values 5, 9, and 3 and then initializes the variable `vector2` with the numbers from 10 to 15 inclusive. The next code snippet initializes the variable `array1` with the contents of `vector1` and `vector2`, constructed as a $3 \times 3 \times 2$ array. Launch the code in Listing 1.15 to see the following output:

```
, , 1, 1

     [,1] [,2]
[1,]    1    1
[2,]    1    1

, , 2, 1

     [,1] [,2]
[1,]    2    2
[2,]    2    2

, , 1, 2

     [,1] [,2]
[1,]    3    3
[2,]    3    3

, , 2, 2

     [,1] [,2]
[1,]    4    4
[2,]    4    4

[1] 2 2 2 2
[1] 5 9 3
[1] 10 11 12 13 14 15
, , 1

     [,1] [,2] [,3]
[1,]    5   10   13
[2,]    9   11   14
[3,]    3   12   15

, , 2

     [,1] [,2] [,3]
[1,]    5   10   13
[2,]    9   11   14
[3,]    3   12   15
```

Listing 1.16 shows the content of `MatrixOperations3.R` that illustrates how to use more matrix-related functions in R.

LISTING 1.16: MatrixOperations3.R

```
vector1 <- c(5,9,3)
vector2 <- c(10,11,12,13,14,15)
```

```
# use these vectors as input to the array:
result <- array(c(vector1,vector2),dim = c(3,3,2))
print(result)

# third row of the second matrix:
print(result[3,,2])

# element in the (1st row, 3rd col) of 1st matrix:
print(result[1,3,1])

# print() the 2nd matrix:
print(result[,,2])
```

Listing 1.16 initializes the variable `vector1` with the values 5, 9, and 3, and then initializes the variable `vector2` with the numbers from 10 to 15 inclusive. The next code snippet initializes the variable `result` with the contents of `vector1` and `vector2`, constructed as a $3 \times 3 \times 2$ array.

The next code snippet prints the contents of the third row of the second matrix, followed by the element in the third column of the first row of the first matrix. The final snippet displays the contents of the second matrix. Launch the code in Listing 1.16 to see the following output:

```
, , 1

     [,1] [,2] [,3]
[1,]    5   10   13
[2,]    9   11   14
[3,]    3   12   15
, , 2

     [,1] [,2] [,3]
[1,]    5   10   13
[2,]    9   11   14
[3,]    3   12   15

[1]  3 12 15
[1] 13
     [,1] [,2] [,3]
[1,]    5   10   13
[2,]    9   11   14
[3,]    3   12   15
```

WORKING WITH MATRICES IN R (2)

The matrix `M` is an $m \times n$ matrix if it has m rows and n columns. In addition, matrix `M` is a *square* matrix if $m = n$.

As an example, the following code snippet creates a 2×3 matrix `X` whose elements are 0:

```
X <- matrix(0, nrow = 2, ncol = 3)
```

The contents of the matrix X are shown here:

```
>X

      [,1] [,2] [,3]
[1,]     0    0    0
[2,]     0    0    0
```

The API dim(X) returns the dimensionality of a matrix, which equals the number of rows and the number of columns. In this example, the dimensionality of X is 3×4:

```
> dim(X)
[1] 3 4
```

Listing 1.17 shows the content of matrices1.R that illustrates more examples of matrices in R.

LISTING 1.17: matrices1.R

```
# Elements are arranged sequentially by row:
M <- matrix(c(3:14), nrow = 4, byrow = TRUE)
print("Matrix M:")
print(M)

sqrtm = sqrt(M)
print("sqrtm:")
print(sqrtm)

# Elements are arranged sequentially by column.
N <- matrix(c(3:14), nrow = 4, byrow = FALSE)
print("Matrix N:")
print(N)

# Define the column and row names.
rownames = c("row1", "row2", "row3", "row4")
colnames = c("col1", "col2", "col3")

P <- matrix(c(3:14),nrow = 4,byrow = TRUE,
dimnames = list(rownames,colnames))
print("Matrix P:")
print(P)
```

Listing 1.17 initializes the matrix M as a 4×3 matrix that contains the integers from 3 to 14 inclusive, where the integers populate the *rows* of M. Next, the variable sqrtm is initialized as the square root of the elements in the array M.

The matrix N is similar to the matrix M, except that N is populated via *columns* instead of rows. Finally, the matrix P is populated with the same values as matrix M but with a different syntax. Launch the code in Listing 1.17 to see the following output:

```
[1] "Matrix M:"
     [,1] [,2] [,3]
[1,]    3    4    5
[2,]    6    7    8
[3,]    9   10   11
[4,]   12   13   14
[1] "sqrtm:"
         [,1]     [,2]     [,3]
[1,] 1.732051 2.000000 2.236068
[2,] 2.449490 2.645751 2.828427
[3,] 3.000000 3.162278 3.316625
[4,] 3.464102 3.605551 3.741657
[1] "Matrix N:"
     [,1] [,2] [,3]
[1,]    3    7   11
[2,]    4    8   12
[3,]    5    9   13
[4,]    6   10   14
[1] "Matrix P:"
     col1 col2 col3
row1    3    4    5
row2    6    7    8
row3    9   10   11
row4   12   13   14
```

WORKING WITH MATRICES IN R (3)

Listing 1.18 shows the content of `matrices2.R` that illustrates how to display the contents of elements of matrices in R.

LISTING 1.18: matrices2.R

```
# Define the column and row names:
rownames = c("row1", "row2", "row3", "row4")
colnames = c("col1", "col2", "col3")

# Create the matrix:
P <- matrix(c(3:14),nrow = 4,byrow = TRUE, dimnames = list(rownames,colnames))
P

# Access the element at 3rd column and 1st row:
print("P[1,3]:")
print(P[1,3])

# Access the element at 2nd column and 4th row:
print("P[4,2]:")
print(P[4,2])

# Access only the 2nd row:
print("P[2,1]:")
print(P[2,])

# Access only the 3rd column:
print("P[,3]:")
print(P[,3])
```

Listing 1.18 initializes the variables `rownames` and `colnames` as vectors of strings, followed by the 4 × 3 matrix `P` that is populated "row wise" with the integers from 3 to 14 inclusive.

The next four blocks of code display the contents of various "cells" in `P`, starting with the element in the 3rd column and the 1st row. See the comments in the code that specify the location of the elements that are displayed. Launch the code in Listing 1.18 to see the following output:

```
     col1 col2 col3
row1    3    4    5
row2    6    7    8
row3    9   10   11
row4   12   13   14
[1] "P[1,3]:"
[1] 5
[1] "P[4,2]:"
[1] 13
[1] "P[2,1]:"
col1 col2 col3
   6    7    8
[1] "P[,3]:"
row1 row2 row3 row4
   5    8   11   14
```

WORKING WITH MATRICES IN R (3)

Listing 1.19 shows the content of `matrices3.R` that illustrates additional operations involving matrices in R.

LISTING 1.19: matrices3.R

```
# Create two 2x3 matrices.
matrix1 <- matrix(c(3, 9, -1, 4, 2, 6), nrow = 2)
print("matrix1:")
print(matrix1)

matrix2 <- matrix(c(5, 2, 0, 9, 3, 4), nrow = 2)
print("matrix2:")
print(matrix2)

# Add the matrices.
result <- matrix1 + matrix2
cat("Result of addition","\n")
print(result)

# Subtract the matrices
result <- matrix1 - matrix2
cat("Result of subtraction","\n")
print(result)
```

Listing 1.19 initializes the variables `matrix1` and `matrix2` and displays their values. Next, the variable `result` is initialized as the *sum* of `matrix1`

and `matrix2`, and then initialized again as the *difference* of `matrix1` and `matrix2`, and the result is displayed in both cases. Launch the code in Listing 1.19 to see the following output:

```
[1] "matrix1:"
     [,1] [,2] [,3]
[1,]    3   -1    2
[2,]    9    4    6
[1] "matrix2:"
     [,1] [,2] [,3]
[1,]    5    0    3
[2,]    2    9    4
Result of addition
     [,1] [,2] [,3]
[1,]    8   -1    5
[2,]   11   13   10
Result of subtraction
     [,1] [,2] [,3]
[1,]   -2   -1   -1
[2,]    7   -5    2
```

WORKING WITH MATRICES IN R (4)

Listing 1.20 shows the content of `matrices4.R` that illustrates how to multiply and divide matrices in R.

LISTING 1.20: matrices4.R

```
# Create two 2x3 matrices.
matrix1 <- matrix(c(3, 9, -1, 4, 2, 6), nrow = 2)
print("matrix1:")
print(matrix1)

matrix2 <- matrix(c(5, 2, 0, 9, 3, 4), nrow = 2)
print("matrix2:")
print(matrix2)

# Multiply the matrices.
result <- matrix1 * matrix2
cat("Result of multiplication","\n")
print(result)

# Divide the matrices
result <- matrix1 / matrix2
cat("Result of division","\n")
print(result)
```

Listing 1.20 initializes the variables `matrix1` and `matrix2` and displays their values. Next, the variable `result` is initialized as the *product* of `matrix1` and `matrix2`, and then initialized again as the *quotient* of `matrix1` and `matrix2`, and the result is displayed in both cases. Launch the code in Listing 1.20 to see the following output:

```
[1] "matrix1:"
     [,1] [,2] [,3]
[1,]    3   -1    2
[2,]    9    4    6
[1] "matrix2:"
     [,1] [,2] [,3]
[1,]    5    0    3
[2,]    2    9    4
Result of multiplication
     [,1] [,2] [,3]
[1,]   15    0    6
[2,]   18   36   24
Result of division
     [,1]      [,2]      [,3]
[1,]  0.6      -Inf 0.6666667
[2,]  4.5 0.4444444 1.5000000
```

UPDATING MATRIX ELEMENTS

This section consists of simple examples that illustrate additional ways to initialize matrices in R.

Example 1:

```
> Y <- matrix(1:12, nrow = 3, ncol = 4)
> Y
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> Y[1, 3]
[1] 7
> Y[1, ]
[1] 1 4 7 10
> Y[, 2]
[1] 4 5 6
```

Example 2:

```
> x <- 1:15
> dim(x) <- c(3, 5) >x
[1,]
[2,]
[3,]
1 4
2 5
3 6
7   10   13
8   11   14
9   12   15
[,1] [,2] [,3] [,4] [,5]
> class(x)
[1] "matrix"
```

The following example illustrates how to define a submatrix in R:

```
> Z <- X[1:2, 3:4]
> Z
     [,1] [,2]
[1,]    0    0
[2,]    0    0
```

The following snippet assigns `x` the contents of matrix `Y` and find the type of `x`:

```
> x <- Y[1, ]
> class(x)
[1] "integer"
```

An example of updating one element and one row of a matrix is as follows:

```
> X[1, 3] <- 1
> X[, 1] <- c(-1, -2, -3) >X
     [,1] [,2] [,3] [,4]
[1,]   -1    0    1    0
[2,]   -2    0    0    0
[3,]   -3    0    0    0
> X[, 4] <- 2 >X
     [,1] [,2] [,3] [,4]
[1,]   -1    0    1    2
[2,]   -2    0    0    2
[3,]   -3    0    0    2
```

LOGICAL CONSTRAINTS AND MATRICES

You can apply a logical condition to the elements of a matrix, and the result is a new matrix that has the same dimensionality. However, the values in the new matrix are either `TRUE` or `FALSE`, depending on whether or not the logical condition is true or false, respectively.

Consider the following example, which returns a value of `TRUE` for the elements of `X` that are positive, and `FALSE` for the non-positive values:

```
> X > 0
      [,1]  [,2]  [,3] [,4]
[1,] FALSE FALSE  TRUE TRUE
[2,] FALSE FALSE FALSE TRUE
[3,] FALSE FALSE FALSE TRUE
```

ASSIGNING VALUES TO MATRIX ELEMENTS

In addition to assigning values to elements during matrix creation operations, it's possible to use Boolean conditions to assign values. For example, the following expression assigns the value `val` to the elements of `X` where the Boolean condition `L` is true:

```
X[L] <- val
```

The following example assigns `NA` to element [1, 1] of `X`:

```
> X[1, 1] <- NA
> is.na(X)
      [,1]  [,2]  [,3]  [,4]
[1,]  TRUE FALSE FALSE FALSE
[2,] FALSE FALSE FALSE FALSE
[3,] FALSE FALSE FALSE FALSE
```

The following code snippet assigns 0 to all the elements of `X` whose value is `NA`:

```
> X[is.na(X)] <- 0
> X
     [,1] [,2] [,3] [,4]
[1,]    0    0    1    2
[2,]   -2    0    0    2
[3,]   -3    0    0    2
```

WORKING WITH MATRICES IN R (5)

The transpose of a matrix is the result of switching rows to columns and columns to rows. If we denote the element in row `i` and column `j` of the matrix `A` by `A(i,j)`, then the coordinates of the corresponding element in the transpose of `A` is `(j,i)`.

Listing 1.21 shows the content of `Transpose1.R` that illustrates how to find the transpose of a matrix in R.

LISTING 1.21: Transpose1.R

```
M = matrix( c(2,6,5,1,10,4), nrow=2,ncol=3,byrow = TRUE)
print() ("contents of M:")
M
t = M %*% t(M)
print() ("contents of t(M):")
t(M)
print() ("contents of t:")
t
```

Listing 1.21 initializes the 2×3 matrix `M` with integer values, followed by the matrix `t` that is the transpose of matrix `M`. Launch the code in Listing 1.21 to see the following output:

```
[1] "contents of M:"
     [,1] [,2] [,3]
[1,]    2    6    5
[2,]    1   10    4
[1] "contents of t(M):"
     [,1] [,2]
[1,]    2    1
[2,]    6   10
```

```
[3,]    5    4
[1] "contents of t:"
     [,1] [,2]
[1,]   65   82
[2,]   82  117
```

Listing 1.22 shows the content of `Transpose2.R` that illustrates how to find the transpose of a matrix in R.

LISTING 1.22: Transpose2.R

```
M = matrix( c(2,6,5,1,10,4,-1,-8,7,23,99,77), nrow=2,ncol=6,byrow = TRUE)
print() ("contents of M:")
M
t = M %*% t(M)
print() ("contents of t(M):")
t(M)
print() ("contents of t:")
t
```

Listing 1.22 is similar to Listing 1.21 that initializes and then displays the contents of the matrices `M`, `t`, and the product of `M` and `t`. Launch the code in Listing 1.22 to see the following output:

```
[1] "contents of M:"
     [,1] [,2] [,3]
[1,]    2    6    5
[2,]    1   10    4
[1] "contents of t(M):"
     [,1] [,2]
[1,]    2    1
[2,]    6   10
[3,]    5    4
[1] "contents of t:"
     [,1] [,2]
[1,]   65   82
[2,]   82  117
```

WORKING WITH DATES IN R

Listing 1.23 shows the content of `date-values.R` that illustrates how to work with matrices in R.

LISTING 1.23: date-values.R

```
mydates <- as.Date(c("2019-06-22", "2021-02-13"))
print("mydates:")
print(mydates)

# number of days between 6/22/19 and 21/12/04
days <- mydates[1] - mydates[2]
print("days:")
print(days)
```

```
# print() today's date
today <- Sys.Date()
format(today, format="%B %d %Y")
print("today:")
print(today)

# convert date info in format 'mm/dd/yyyy'
strDates <- c("01/05/2021", "08/13/2022")
dates <- as.Date(strDates, "%m/%d/%Y")
print("dates:")
print(dates)

#The default format is yyyy-mm-dd
mydates <- as.Date(c("2020-08-13", "2022-08-13"))
print("mydates:")
print(mydates)

# convert dates to character data
strDates <- as.character(dates)
print("strDates:")
print(strDates)
```

The code in Listing 1.23 contains code snippets that illustrate how to use the `as.Date()` function to convert strings to dates and how to subtract two dates. Launch the code in Listing 1.23 to see the following output:

```
[1] "mydates:"
[1] "2019-06-22" "2021-02-13"
[1] "days:"
Time difference of -602 days
[1] "December 06 2021"
[1] "today:"
[1] "2021-12-06"
[1] "dates:"
[1] "2021-01-05" "2022-08-13"
[1] "mydates:"
[1] "2020-08-13" "2022-08-13"
[1] "strDates:"
[1] "2021-01-05" "2022-08-13"
```

THE SEQ FUNCTION IN R

Earlier in the chapter, you saw a code sample that contains the `seq()` function, which is a function in R that generates sequences of data. Listing 1.24 shows the content of `SequenceFunctions.R` that illustrates how to work with sequences in R.

LISTING 1.24: SequenceFunctions.R

```
#Generate a sequence from 1 to 10:
x <- seq(10)
x
```

```
#Generate a sequence from -4 to 4:
x <- seq(-4,4)
x

#Generate a sequence from -4 to 4 with a step of 2:
x <- seq(-4,4,by=2)
x

#Generate a sequence from -4 to 4 with a step of 0.5:
x <- seq(-2,2,by=0.5)
x

exp(x)
#[1]     2.718282     7.389056    20.085537    54.5

#Generate 10 equally distributed numbers from -2 to 2:
seq(-2,2,length.out=10)

x = seq(-pi,pi,length=20)
print("PI values:")
print(x)
```

Listing 1.24 contains code snippets and comments that explain the purpose of each code snippet. Launch the code in Listing 1.24 to see the following output:

```
 [1]  1  2  3  4  5  6  7  8  9 10
 [1] -4 -3 -2 -1  0  1  2  3  4
 [1] -4 -2  0  2  4
 [1] -2.0 -1.5 -1.0 -0.5  0.0  0.5  1.0  1.5  2.0
 [1] -2.0000000 -1.5555556 -1.1111111 -0.6666667 -0.2222222  0.2222222
 [7]  0.6666667  1.1111111  1.5555556  2.0000000
 [1] 0.1353353 0.2231302 0.3678794 0.6065307 1.0000000 1.6487213 2.7182818
 [8] 4.4816891 7.3890561
 [1] "PI values:"
 [1] -3.1415927 -2.8108987 -2.4802047 -2.1495108 -1.8188168 -1.4881228
 [7] -1.1574289 -0.8267349 -0.4960409 -0.1653470  0.1653470  0.4960409
[13]  0.8267349  1.1574289  1.4881228  1.8188168  2.1495108  2.4802047
[19]  2.8108987  3.1415927
```

Listing 1.25 shows the content of `seq-function.R` that illustrates how to generate a sequence of numbers in R.

LISTING 1.25: seq-function.R

```
N = 300
set.seed(110)
X = seq(1:N)
Y = X/10+4*sin(X/10)+sample(-1:6,X,replace=T)+rnorm(X)
head(Y,20)
```

Listing 1.25 initializes the variable `N` with the value 300, followed by the variable `X` that contains the numbers from 1 to `N` (see the previous section for examples involving the `seq()` function).

Next, the variable `Y` is initialized as a function that contains a mixture of a linear term `X/10`, the trigonometric `sin()` function, values from the `sample()` function, and randomly selected values from a normal distribution via the `rnorm()` function. Launch the code in Listing 1.25 to see the following output:

```
 [1] 2.2998879 3.1848840 0.7925252 4.1971693 3.3087362 3.7413279
 [7] 4.4599534 5.9499792 3.3060370 7.0560683 6.1686419 8.4564929
[13] 6.7033295 6.5800118 5.4410868 7.8930577 7.4921991 6.6397045
[19] 5.9008181 5.9036865
```

SUMMARY

This chapter introduced you to R variables, and how to define variables whose type is strings, lists, vectors, and matrices in R. Then you learned ways to initialize variables during their creation and how to update the values of variables.

You learned how to update the values of a specific row in two-dimensional matrices in R. Moreover, you saw how to use conditional logic to test the values in a two-dimensional matrix, and also use conditional logic to update the elements in a two-dimensional matrix. Finally, you learned how to work with dates in R.

CHAPTER 2

LOOPS, CONDITIONAL LOGIC, AND DATAFRAMES

This chapter discusses four main topics: working with loops in R, working with conditional logic in R, working with data frames in R, and how to perform various types of data visualization in R.

The first section of this chapter contains short code samples that illustrate various types of loops (including nested loops) in R, which includes `for` loops, `while` loops, and `repeat` loops.

The second section discusses conditional logic, starting with simple if-then statements. Conditional logic includes if-then-else statements; they can also be nested, as shown in one of the code samples in this section.

The third section discusses data frames, an extremely powerful datatype in R that are counterparts to data frames in Python Pandas. The fourth section discusses data visualization, such as bar charts, line graphs, histograms, scatter plots, and pie charts.

WORKING WITH SIMPLE LOOPS IN R

Listing 2.1 shows the content of `simpleloop1.R` that illustrates a simple `for` loop in R that calculates the sum of some integers.

LISTING 2.1 simpleloop1.R

```
x <- c(2,5,3,9,8,11,6)

count <- 0
sum <- 0
for (val in x) {
    count = count+1
    sum = sum + val
}
```

```
print(paste0("count: ",count)
print(paste0("sum:   ",sum)
```

Listing 2.1 initializes the vector `x` with seven positive integers and the variable `count` with the value 0. Next, there is a `for` loop that iterates through the elements in `x`, incrementing the value of `count` during each iteration. Launch the code in Listing 2.1 to see the following output:

```
[1] "count: 7"
[1] "sum:   44
```

Working with Other Types of Loops in R

In addition to `for` loops, an example of which you saw in the previous section, R supports a `while` loop and a `repeat` loop. Later in this chapter, you will see an example of a `while` loop, right after you see how to create a nested `for` loop, which is discussed in the next section.

WORKING WITH NESTED LOOPS IN R

Listing 2.2 shows the content of `nestedloop1.R` that illustrates how to define a nested loop in R.

LISTING 2.2: nestedloop1.R

```
x <- c(1,2,3)
y <- c(10,20,30)

for (x1 in x) {
   for (y1 in y) {
      print(paste0("(",x1,",",y1,")"))
   }
}
```

Listing 2.2 initializes the vectors `x` and `y` with positive integers, followed by a nested loop that displays pairs of numeric values: the first value is an element of `x`, and the second value is an element of `y`. Launch the code in Listing 2.2 to see the following output:

```
[1] "(1,10)"
[1] "(1,20)"
[1] "(1,30)"
[1] "(2,10)"
[1] "(2,20)"
[1] "(2,30)"
[1] "(3,10)"
[1] "(3,20)"
[1] "(3,30)"
```

WORKING WITH WHILE LOOPS IN R

In addition to `for` loops, R also supports `while` loops. Listing 2.3 shows the content of `whileloop1.R` that illustrates how to define a `while` loop in R.

LISTING 2.3: whileloop1.R

```
i <- 1

while (i < 6) {
   print(paste0("i: ",i))

   i = i+1
}
```

Listing 2.3 initializes the variable `i` with the value 1 and then executes a `while` loop that prints the value of `i` and then increments the value of `i`. The `while` loop executes as long as `i` is less than 6. Launch the code in Listing 2.3 to see the following output:

```
[1] "i: 1"
[1] "i: 2"
[1] "i: 3"
[1] "i: 4"
[1] "i: 5"
```

Instead of hard-coding the value 6 in the `while` loop, it's preferable to replace the number 6 with a variable that is initialized with the value 6, or whichever value you need for your purposes. Now that you have a basic understanding of `for` loops and `while` loops in R, let's explore conditional logic in R.

WORKING WITH CONDITIONAL LOGIC IN R

Conditional logic appears in almost every non-trivial program, regardless of the programming language. Conditional logic can vary in complexity from a simple `if` statement to multiple nested `if` statements, which in turn can contain other `if` statements. Although complex conditional logic can be a source of coding bugs, nothing prevents you from writing such code. At a minimum, provide meaningful comments for code blocks to facilitate a better understanding of their purpose.

Listing 2.4 shows the content of `ifelse1.R` that illustrates a simple example of conditional logic in a `for` loop in R.

LISTING 2.4: ifelse1.R

```
nums = c(5,7,2,9)

for (a in nums) {
   if (a %% 2 == 0) {
     print(paste0(a," is even"))
   } else {
     print(paste0(a," is odd"))
   }
}
```

Listing 2.4 initializes the variable `nums` with four positive integers, followed by a `for` loop that displays one message if the current number is even, and a

different message of the number is odd. Launch the code in Listing 2.4 to see the following output:

```
[1] "5 is odd"
[1] "7 is odd"
[1] "2 is even"
[1] "9 is odd"
```

COMPOUND CONDITIONAL LOGIC

Listing 2.5 shows the content of `CompoundIfLogic.R` that illustrates how to check if a number is divisible by multiple numbers.

LISTING 2.5: CompoundIfLogic.R

```
x <- 30
print(paste0("x = ",x))

if( (x %% 3 == 0) && (x %% 5 == 0) ) {
   print(paste0("x is a multiple of 3 and 5"))
} else if( x %% 5 == 0) {
   print(paste0("x is a multiple of 5"))
} else if( x %% 3 == 0) {
   print(paste0("x is a multiple of 3"))
} else {
   print(paste0("x is not a multiple of 3 or 5"))
}
```

Listing 2.5 initializes the variable `x` with the value 30 and displays its value. The main code block contains a sequence of if/else-if statements, which continue to execute until a conditional statement is true, after which a `print()` statement displays a message and then the execution of this code stops. For example, if the first `if` statement is *true*, then the remainder of the code will not be executed. If the first `if` statement is *false and* the first else-if is *true*, then the other else-if statements are *not* executed. Launch the code in Listing 2.5 and enter some values:

```
[1] "x = 30"
[1] "x is a multiple of 3 and 5"
```

Now let's turn to a task that *does* require conditional logic, such as checking if a number is prime.

CHECK IF A NUMBER IS PRIME IN R

A positive integer greater than 1 is a prime number if its only divisors are 1 and the number itself. Hence, the set {2, 3, 5, 7, 11, 13} consists of prime numbers.

One algorithm for determining whether a number N is prime involves dividing N by the number 2 and the odd numbers from 3 to N/2: if the remainder is 0, then N is a composite number. Otherwise, N is a prime number. Note that the upper bound can be reduced from N/2 to `sqrt(N)`, which can significantly reduce the computation time for larger values of N.

Listing 2.6 shows the content of `PrimeNumber.R` that illustrates how to work with a loop and conditional logic in R to determine whether a positive integer is prime. The upper bound in the `for` loop can be further decreased to become computationally more efficient for large values of `num`.

LISTING 2.6: PrimeNumber.R

```
num <- 20

# prime numbers are >= 2
flag = 0
if(num > 1) {
   # check for factors
   flag = 1
   # the following loop works for num > 2
   for(i in 2:(num-1)) {
      if ((num %% i) == 0) {
         print(paste(i,"is a divisor of",num))
         flag = 0
         break
      }
   }
}

if(num == 2)
   flag = 1

if(flag == 1) {
   print(paste(num,"is a prime number"))
} else {
   print(paste(num,"is not a prime number"))
}
```

Listing 2.6 initializes the value of `num` to 20 and initializes the variable `flag` to 0. Next, an `if` statement checks if `num` is greater than 1: if so, then another block of code that consists of a `for` loop is executed.

The `for` loop iterates through the integers from 2 to `num`–1 inclusive, and if any of those numbers divides `num` with remainder zero, a message is displayed, the variable `flag` is set to 0, and an early exit occurs.

The remaining portion of Listing 2.6 checks the value of `flag` and uses its value to display an appropriate message. Launch the code in Listing 2.6 to see the following output:

```
[1] "2 is a divisor of 20"
[1] "20 is not a prime number"
```

CHECK IF NUMBERS IN AN ARRAY ARE PRIME IN R

The previous section showed how to determine whether a positive integer is a prime number. Listing 2.7 shows the content of `PrimeNumbers.R` that illustrates how to check if any of the numbers in an array are prime.

LISTING 2.7: PrimeNumbers.R

```
prime <- function(num) {
   # prime numbers are >= 2
   flag = 0
   if(num > 1) {
      # check for factors
      flag = 1
      for(i in 2:(num-1)) {
         if ((num %% i) == 0) {
            flag = 0
            break
         }
      }
   }

   if(num == 2)
      flag = 1

   if(flag == 1) {
      print(paste(num,"is a prime number"))
   } else {
      print(paste(num,"is not a prime number"))
   }
}

for (num in 10:20){
  prime(num)
}

arr <- c(7, 17, 25, 99)
for (num in arr){
  prime(num)
}
```

Listing 2.7 defines the function `prime()`, whose code is the same as the code in Listing 2.6. The last portion of Listing 2.7 contains a `for` loop that iterates through the numbers from 10 to 10 and invokes the `prime()` function to determine whether those numbers are prime.

The second `for` loop is similar: it also iterates through the numbers in a list and invokes the `prime()` function to determine whether that number is prime. Launch the code in Listing 2.7 to see the following output:

```
[1] "10 is not a prime number"
[1] "11 is a prime number"
[1] "12 is not a prime number"
[1] "13 is a prime number"
[1] "14 is not a prime number"
```

```
[1] "15 is not a prime number"
[1] "16 is not a prime number"
[1] "17 is a prime number"
[1] "18 is not a prime number"
[1] "19 is a prime number"
[1] "20 is not a prime number"
[1] "7 is a prime number"
[1] "17 is a prime number"
[1] "25 is not a prime number"
[1] "99 is not a prime number"
```

CHECK FOR LEAP YEARS IN R

Whether a positive integer is a leap year can be determined via nested `if` statements. Listing 2.8 shows the content of `CheckForLeapYear.R` that illustrates how to determine whether a positive integer is a leap year in R.

LISTING 2.8: CheckForLeapYear.R

```
#########################################
# A year is a leap year provided that:
# 1) it is a multiple of 4 AND
# 2) a century must be a multiple of 400
#
# => 2000 is a leap year but 1900 is not.
#########################################
year <- 1904

if((year %% 4) == 0) {
    if((year %% 100) == 0) {
        if((year %% 400) == 0) {
            print(paste(year,"is a leap year"))
        } else {
            print(paste(year,"is not a leap year"))
        }
    } else {
        print(paste(year,"is a leap year"))
    }
} else {
    print(paste(year,"is not a leap year"))
}
```

Listing 2.8 initializes the variable `year` with the value 1904, followed by a set of nested `if` statements that implement the logic described in the comment block. Launch the code in Listing 2.8 to see the following output:

```
[1] "1900 is not a leap year"
```

WELL-FORMED TRIANGLE VALUES IN R

Recall that three positive numbers (not necessarily integers) are the angles of a triangle if the sum of those numbers equals 180. Since the three numbers are positive, they must be greater than 0 and less than 180.

In addition, the sum of any two of the three numbers must be greater than 0 and less than 180. Note that this result is for the Euclidean plane, and this is not a requirement for elliptic geometry or hyperbolic geometry.

Listing 2.9 shows the content of `SumOfAngles.R` that illustrates how to determine whether three angles form a triangle in the Euclidean plane.

LISTING 2.9: SumOfAngles.R

```
a1 = 40, a2 = 80, a3 = 0

a1 = 40
a2 = 80
a3 = 60

print(paste0("a1: ",a1))
print(paste0("a2: ",a2))
print(paste0("a3: ",a3))

# ensure the following are true:
# 1) a1>0 and a1 < 180
# 2) a2>0 and a2 < 180
# 3) a1+a1 < 180

if( ((a1 <= 0) || (a1 >= 180))  ||
    ((a2 <= 0) || (a2 >= 180)) )
{
   print(paste0("angles out of range: ",a1,a2))
} else {
   if( a1+a2 >= 180 ) {
      print(paste0("a1 + a2 is too large:", a1+a2))
   } else {
      a3 = 180 - (a1+a2)
      print(paste0("a1, a2, and a3 form a triangle:", a1,"
",a2," ",a3))
   }
}
```

Listing 2.9 initializes the variables `a1`, `a2`, and `a3` with three positive integer values. The next set of if/else statements implement the logic described in the comment block. Launch the code in Listing 2.9 to see the following output:

```
[1] "a1: 40"
[1] "a2: 80"
[1] "a3: 60"
[1] "a1, a2, and a3 form a triangle:40 80 60"
```

WHAT ARE FACTORS IN R?

Factors in R are similar to the `enum` (enumeration) data type in other programming languages. Factors are created via the `factor()` function. Listing 2.10 shows the content of `factors1.R` that illustrates how to define factors in R.

LISTING 2.10: factors1.R

```
a <-  "Hello"
mycolors <- c('green','green','yellow','red','red','red','green')

# Create a factor object:
myfactors <- factor(mycolors)
print("contents of the myfactors vector:")
print(myfactors)

print("the number of levels in myfactors:")
print(nlevels(myfactors))
```

Listing 2.10 defines a vector of strings `mycolors` and then initializes the variable `myfactors` with the "factors" in the variable `mycolors`, which consists of three distinct colors. Launch the code in Listing 2.10 to see the following output:

```
[1] "contents of the myfactors vector:"
[1] green  green  yellow red    red    red    green
Levels: green red yellow
[1] "the number of levels in myfactors:"
[1] 3
```

WHAT ARE DATA FRAMES IN R?

A *data frame* in R is comparable to a spreadsheet: you can perform various column-related and row-related operations on a data frame, in much the same way that you can perform those operations on a spreadsheet. For example, you can insert, delete, or move rows and columns. You can update values based on various criteria, such as filling in missing values or modifying existing values.

Data frames are essentially data objects in tabular form, with heterogeneous columns of data, created via the `data.frame()` function. A list in R is compatible with `data.frame` if any of the following is true:

- Components must be vectors (numeric, character, and logical) or factors.
- All vectors and factors must have the same lengths.

Matrices and other data frames can be combined with vectors to form a data frame if the dimensions are compatible.

Listing 2.11 shows the content of `simple_df.R` that defines a data frame consisting of the three columns `col1`, `col2`, and `col3`, each of which contains three positive integers.

LISTING 2.11: simple_df.R

```
mydf <- data.frame(
  attr1 <- c(1,2,3),
  attr2 <- c(4,5,6),
  attr3 <- c(7,8,9)
)
```

```
print("contents of attr1 of mydf:")
print(mydf[,1])
print("contents of attr2 of mydf:")
print(mydf[,2])
print("contents of attr3 of mydf:")
print(mydf[,3])

print("contents of column 1 of mydf:")
print(mydf[1,])
print("contents of column 2 of mydf:")
print(mydf[2,])
print("contents of column 3 of mydf:")
print(mydf[3,])

# second dataframe:
attr4 <- c('a','b','c')
attr5 <- c('d','e','f')

mydf2 <- data.frame(
  attr4,
  attr5
)

print("contents of mydf2:")
print(mydf2)
```

Listing 2.11 initializes the variable `mydf` as a data frame with the positive integers from 1 to 9, which are also used to initialize the variables `attr1`, `attr2` , and `attr3`. The next portion of code displays the contents of the same variables, expressed as elements of the variable `mydf`.

The next code block addresses the contents of three columns, and the final code section initializes the variable `mydf2` as a data frame that consists of the contents of `attr4` and `attr5`, and then displays its contents. Launch the code in Listing 2.11 to see the following output:

```
[1] "contents of attr1 of mydf:"
[1] 1 2 3
[1] "contents of attr2 of mydf:"
[1] 4 5 6
[1] "contents of attr3 of mydf:"
[1] 7 8 9
[1] "contents of column 1 of mydf:"
  attr1....c.1..2..3. attr2....c.4..5..6. attr3....c.7..8..9.
1                   1                   4                   7
[1] "contents of column 2 of mydf:"
  attr1....c.1..2..3. attr2....c.4..5..6. attr3....c.7..8..9.
2                   2                   5                   8
[1] "contents of column 3 of mydf:"
  attr1....c.1..2..3. attr2....c.4..5..6. attr3....c.7..8..9.
3                   3                   6                   9
[1] "contents of mydf2:"
  attr4 attr5
1     a     d
2     b     e
3     c     f
```

The next sequence of sections in this chapter contains examples of performing operations on the contents of data frames in R.

WORKING WITH DATAFRAMES IN R (1)

Listing 2.12 shows the content of `dataframe2.R` that illustrates how to define a data frame with heterogeneous values in R.

LISTING 2.12: dataframe2.R

```
# Create the data frame:
emp.data <- data.frame(
 emp_id = c (1:5),
 emp_name = c("John","Jane","Sally","Sarah","Robert"),
 salary = c(1000.12,2515.2,3600.0,1750.0,1800.25),

 start_date =
   as.Date(c("2021-01-01","2021-07-01","2022-02-02",
"2022-05-05","2021-12-28")),
   stringsAsFactors = FALSE
)

# Print the data frame:
print(emp.data)

# Get the structure of the data frame:
str(emp.data)

# Print the summary:
print(summary(emp.data))
```

Listing 2.12 initializes the variable `emp` as a data frame that contains values for `emp_id`, `emp_name`, `salary`, and `start_date` for five fictitious employees. The remaining code in Listing 2.12 shows the contents of the data frame `emp`, the structure of `emp`, and a summary report that contains information about the minimum, maximum, median, and mean values in the data frame. Launch the code in Listing 2.12 to see the following output:

```
  emp_id emp_name  salary start_date
1      1     John 1000.12 2021-01-01
2      2     Jane 2515.20 2021-07-01
3      3    Sally 3600.00 2022-02-02
4      4    Sarah 1750.00 2022-05-05
5      5   Robert 1800.25 2021-12-28
'data.frame':     5 obs. of  4 variables:
 $ emp_id    : int  1 2 3 4 5
 $ emp_name  : chr  "John" "Jane" "Sally" "Sarah" ...
 $ salary    : num  1000 2515 3600 1750 1800
 $ start_date: Date, format: "2021-01-01" "2021-07-01" ...
     emp_id    emp_name              salary       start_date
 Min.   :1   Length:5           Min.   :1000   Min.   :2021-01-01
 1st Qu.:2   Class :character   1st Qu.:1750   1st Qu.:2021-07-01
 Median :3   Mode  :character   Median :1800   Median :2021-12-28
 Mean   :3                      Mean   :2133   Mean   :2021-10-13
 3rd Qu.:4                      3rd Qu.:2515   3rd Qu.:2022-02-02
 Max.   :5                      Max.   :3600   Max.   :2022-05-05
```

WORKING WITH DATA FRAMES IN R (2)

Listing 2.13 shows the content of `dataframe3.R` that illustrates how to display portions of a data frame in R.

LISTING 2.13: dataframe3.R

```
# Create the data frame:
emp.data <- data.frame(
 emp_id = c (1:5),
 emp_name = c("Rick","Dan","Michelle","Ryan","Gary"),
 salary = c(623.3,515.2,611.0,729.0,843.25),

 start_date =
   as.Date(c("2012-01-01","2013-09-23","2014-11-15",
   "2014-05-11","2015-03-27")),
   stringsAsFactors = FALSE
)

# Print the data frame:
print(emp.data)

# Get the structure of the data frame:
str(emp.data)

# Print the summary:
print(summary(emp.data))

# Extract Specific columns:
result <- data.frame(emp.data$emp_name,emp.data$salary)
print(result)

# Extract first two rows:
result <- emp.data[1:2,]
print(result)

# Extract 3rd and 5th row with 2nd and 4th column.
result <- emp.data[c(3,5),c(2,4)]
print(result)
```

Listing 2.13 starts with the same code as Listing 2.12, with the new section of code shown in bold. The new code block initializes the variable `result` with the values for the employee names and employee salaries. Launch the code in Listing 2.13 to see the following output:

```
  emp_id emp_name salary start_date
1      1     Rick 623.30 2012-01-01
2      2      Dan 515.20 2013-09-23
3      3 Michelle 611.00 2014-11-15
4      4     Ryan 729.00 2014-05-11
5      5     Gary 843.25 2015-03-27
'data.frame':   5 obs. of  4 variables:
 $ emp_id    : int  1 2 3 4 5
 $ emp_name  : chr  "Rick" "Dan" "Michelle" "Ryan" ...
 $ salary    : num  623 515 611 729 843
```

```
 $ start_date: Date, format: "2012-01-01" "2013-09-23" ...
     emp_id    emp_name              salary         start_date
 Min.   :1   Length:5           Min.   :515.2   Min.   :2012-01-01
 1st Qu.:2   Class :character   1st Qu.:611.0   1st Qu.:2013-09-23
 Median :3   Mode  :character   Median :623.3   Median :2014-05-11
 Mean   :3                      Mean   :664.4   Mean   :2014-01-14
 3rd Qu.:4                      3rd Qu.:729.0   3rd Qu.:2014-11-15
Max.   :5                      Max.   :843.2   Max.   :2015-03-27
  emp.data.emp_name emp.data.salary
1              Rick          623.30
2               Dan          515.20
3          Michelle          611.00
4              Ryan          729.00
5              Gary          843.25
  emp_id emp_name salary start_date
1      1     Rick  623.3 2012-01-01
2      2      Dan  515.2 2013-09-23
  emp_name start_date
3 Michelle 2014-11-15
5     Gary 2015-03-27
```

WORKING WITH DATA FRAMES IN R (3)

Listing 2.14 shows the content of `dataframe4.R` that illustrates how to add a new attribute to a data frame in R.

LISTING 2.14: dataframe4.R

```
emp.data <- data.frame(
 emp_id = c (1:5),
 emp_name = c("Rick","Dan","Michelle","Ryan","Gary"),
 salary = c(623.3,515.2,611.0,729.0,843.25),

 start_date =
   as.Date(c("2012-01-01","2013-09-23","2014-11-15",
   "2014-05-11","2015-03-27")),
   stringsAsFactors = FALSE
)

# Print the data frame:
print(emp.data)

# Get the structure of the data frame:
str(emp.data)

# Print the summary:
print(summary(emp.data))

# Extract Specific columns:
result <- data.frame(emp.data$emp_name,emp.data$salary)
print(result)

# Extract first two rows:
result <- emp.data[1:2,]
print(result)
```

```
# Extract 3rd and 5th row with 2nd and 4th column.
result <- emp.data[c(3,5),c(2,4)]
print(result)

# Create the data frame.
emp.data <- data.frame(
   emp_id = c (1:5),
   emp_name = c("Rick","Dan","Michelle","Ryan","Gary"),
   salary = c(623.3,515.2,611.0,729.0,843.25),

   start_date = as.Date(c("2012-01-01", "2013-09-23",
   "2014-11-15", "2014-05-11",
      "2015-03-27")),
   stringsAsFactors = FALSE
)

# Add the "dept" column:
emp.data$dept <- c("IT","Operations","IT","HR","Finance")
v <- emp.data
print(v)
```

Listing 2.14 starts with the same code as Listing 2.13, and the new block of code is shown in bold. The new code adds a `dept` attribute (which contains five values) to the `emp` variable. Launch the code in Listing 2.14 to see the following output:

```
  emp_id emp_name salary start_date
1      1     Rick 623.30 2012-01-01
2      2      Dan 515.20 2013-09-23
3      3 Michelle 611.00 2014-11-15
4      4     Ryan 729.00 2014-05-11
5      5     Gary 843.25 2015-03-27
'data.frame':   5 obs. of  4 variables:
 $ emp_id    : int  1 2 3 4 5
 $ emp_name  : chr  "Rick" "Dan" "Michelle" "Ryan" ...
 $ salary    : num  623 515 611 729 843
 $ start_date: Date, format: "2012-01-01" "2013-09-23" ...
     emp_id    emp_name             salary        start_date
 Min.   :1   Length:5           Min.   :515.2   Min.   :2012-01-01
 1st Qu.:2   Class :character   1st Qu.:611.0   1st Qu.:2013-09-23
 Median :3   Mode  :character   Median :623.3   Median :2014-05-11
 Mean   :3                      Mean   :664.4   Mean   :2014-01-14
 3rd Qu.:4                      3rd Qu.:729.0   3rd Qu.:2014-11-15
Max.   :5                      Max.   :843.2   Max.   :2015-03-27
  emp.data.emp_name emp.data.salary
1              Rick          623.30
2               Dan          515.20
3          Michelle          611.00
4              Ryan          729.00
5              Gary          843.25
  emp_id emp_name salary start_date
1      1     Rick  623.3 2012-01-01
2      2      Dan  515.2 2013-09-23
  emp_name start_date
3 Michelle 2014-11-15
5     Gary 2015-03-27
```

```
  emp_id emp_name salary start_date       dept
1      1     Rick 623.30 2012-01-01         IT
2      2      Dan 515.20 2013-09-23 Operations
3      3 Michelle 611.00 2014-11-15         IT
4      4     Ryan 729.00 2014-05-11         HR
5      5     Gary 843.25 2015-03-27    Finance
```

WORKING WITH DATA FRAMES IN R (4)

Listing 2.15 shows the content of `dataframe5.R` that illustrates how to append a new row to a data frame in R.

LISTING 2.15: dataframe5.R

```
emp.data <- data.frame(
   emp_id = c (1:5),
   emp_name = c("Rick","Dan","Michelle","Ryan","Gary"),
   salary = c(623.3,515.2,611.0,729.0,843.25),
   start_date = as.Date(c("2012-01-01","2013-09-23",
"2014-11-15","2014-05-11", "2015-03-27")),
   dept = c("IT","Operations","IT","HR","Finance"),
   stringsAsFactors = FALSE
)

# Create the second data frame:
emp.newdata <-  data.frame(
   emp_id = c (6:8),
   emp_name = c("Jane","Jack","John"),
   salary = c(578.0,722.5,632.8),
   start_date = as.Date(c("2013-05-21","2013-07-30", "2014-06-17")),
   dept = c("Dev","Sales","BizDev"),
   stringsAsFactors = FALSE
)

# Bind the two data frames:
emp.finaldata <- rbind(emp.data,emp.newdata)
print(emp.finaldata)
```

Listing 2.15 initializes the variable `emp` as a data frame containing data for five employees, as shown in previous code samples. The next portion of Listing 2.15 adds three new employees to `emp`. Launch the code in Listing 2.15 to see the following output:

```
  emp_id emp_name salary start_date       dept
1      1     Rick 623.30 2012-01-01         IT
2      2      Dan 515.20 2013-09-23 Operations
3      3 Michelle 611.00 2014-11-15         IT
4      4     Ryan 729.00 2014-05-11         HR
5      5     Gary 843.25 2015-03-27    Finance
6      6     Jane 578.00 2013-05-21        Dev
7      7     Jack 722.50 2013-07-30      Sales
8      8     John 632.80 2014-06-17     BizDev
```

WORKING WITH DATA FRAMES IN R (5)

Listing 2.16 shows the content of `dataframe6.R` that illustrates how to work with a data frame in R.

LISTING 2.16: dataframe6.R

```
city <- c("Tampa","Seattle","Hartford","Denver")
state <- c("FL","WA","CT","CO")
zipcode <- c(33602,98104,06161,80294)

# Combine above three vectors into one data frame:
addresses <- cbind(city,state,zipcode)

# Print a header:
cat("# # # # The First data frame\n")

# Print the data frame:
print(addresses)

# Create another data frame with similar columns:
new.address <- data.frame(
   city = c("Oakwood","Saperton"),
   state = c("CO","FL"),
   zipcode = c("80230","33949"),
   stringsAsFactors = FALSE
)

# Print a header:
cat("# # # The Second data frame\n")

# Print the data frame:
print(new.address)

# Combine rows form both the data frames:
all.addresses <- rbind(addresses,new.address)

# Print a header:
cat("# # # The combined data frame\n")

# Print the result:
print(all.addresses)
```

Listing 2.16 initializes the variables `city`, `state`, and `zipcode` with five cities, states, and zip codes, respectively. Next, the variable `addresses` is initialized with the contents of the preceding three variables.

Another code block initializes the variable `new` with another set of location-related values, and then displays its contents. The final section of code concatenates the contents of `addresses` and `new.address`, and uses the result to initialize the variable `all`. Launch the code in Listing 2.16 to see the following output:

```
# # # # The First data frame
     city       state zipcode
```

```
[1,] "Tampa"    "FL"  "33602"
[2,] "Seattle"  "WA"  "98104"
[3,] "Hartford" "CT"  "6161"
[4,] "Denver"   "CO"  "80294"
# # # The Second data frame
      city state zipcode
1  Oakwood    CO   80230
2 Saperton    FL   33949
# # # The combined data frame
      city state zipcode
1    Tampa    FL   33602
2  Seattle    WA   98104
3 Hartford    CT    6161
4   Denver    CO   80294
5  Oakwood    CO   80230
6 Saperton    FL   33949
```

READING EXCEL FILES IN R

Listing 2.17 shows the content of `readXSL.R` that illustrates how easily you can read an Excel spreadsheet into a data frame in R.

LISTING 2.17: readXLS.R

```
library(readxl)
dfb <- read_excel("employees.xlsx")

print("The first five rows of employees.xlsx:")
head(dfb)

print("A Summary of employees.xlsx:")
summary(dfb)
```

In Listing 2.17, after loading the `readxl` library, the variable `dfb` is initialized from the contents of the `employees.xlsx` spreadsheet. The first five rows are displayed, followed by a summary, as shown below:

```
head(dfb)
[1] "The first five rows of employees.xlsx:"
# A tibble: 6 x 10
     id fname lname gender     title    q1    q2    q3    q4 country
  <dbl> <chr> <chr>  <chr>     <chr> <dbl> <dbl> <dbl> <dbl>   <chr>
1  1000  john smith      m marketing 20000 12000 18000 25000     usa
2  2000  jane smith      f developer 30000 15000 11000 35000  france
3  3000  jack jones      m     sales 10000 19000 12000 15000     usa
4  4000  dave stone      m   support 15000 17000 14000 18000  france
5  5000  sara stein      f   analyst 25000 22000 18000 28000   italy
6  6000  eddy bower      m developer 14000 32000 28000 10000  france
[1] "A Summary of employees.xlsx:"
       id           fname              lname              gender
 Min.   :1000   Length:6           Length:6           Length:6
 1st Qu.:2250   Class :character   Class :character   Class :character
 Median :3500   Mode  :character   Mode  :character   Mode  :character
 Mean   :3500
 3rd Qu.:4750
 Max.   :6000
```

```
    title                    q1               q2               q3       
 Length:6           Min.   :10000   Min.   :12000   Min.   :11000  
 Class :character   1st Qu.:14250   1st Qu.:15500   1st Qu.:12500  
 Mode  :character   Median :17500   Median :18000   Median :16000  
                    Mean   :19000   Mean   :19500   Mean   :16833  
                    3rd Qu.:23750   3rd Qu.:21250   3rd Qu.:18000  
                    Max.   :30000   Max.   :32000   Max.   :28000  
       q4             country         
 Min.   :10000   Length:6          
 1st Qu.:15750   Class :character  
 Median :21500   Mode  :character  
 Mean   :21833                     
 3rd Qu.:27250                     
 Max.   :35000                     
```

READING SQLITE TABLES IN R

Listing 2.18 shows the content of `readSQLite.R` that illustrates how to read the contents of a built-in SQLITE database in R.

LISTING 2.18: readSQLite.R

```
library(RSQLite)
library(DBI)

print("Establishing database connection...")
db = RSQLite::datasetsDb()

# display the tables in the database
print("Reading database tables...")
dbListTables(db)

print("Reading contents of mtcars table...")
dbReadTable(db, "mtcars")

# filter the data
print("Listing rows in the mtcars table...")
dbGetQuery(db, "SELECT * FROM mtcars")

print("Disconnecting database connection...")
dbDisconnect(db)
```

Listing 2.18 starts by referencing the `RSQLite` and `DBI` libraries for managing database connections. Next, `db` is initialized with the list of built-in databases in R. The specific table that we want to examine is called `mtcars`, which we access via this code snippet:

```
dbReadTable(db, "mtcars")
```

The next portion of Listing 2.18 executes a `SELECT` statement that retrieves all the rows from the `mtcars` table and displays its contents. The final code snippet disconnects from the database. Launch the code in Listing 2.18 to see the following output:

```
Head
 [1] "Establishing database connection..."
 [1] "Reading database tables..."
 [1] "BOD"              "CO2"              "ChickWeight"      "DNase"
 [5] "Formaldehyde"     "Indometh"         "InsectSprays"     "LifeCycleSavings"
 [9] "Loblolly"         "Orange"           "OrchardSprays"    "PlantGrowth"
[13] "Puromycin"        "Theoph"           "ToothGrowth"      "USArrests"
[17] "USJudgeRatings"   "airquality"       "anscombe"         "attenu"
[21] "attitude"         "cars"             "chickwts"         "esoph"
[25] "faithful"         "freeny"           "infert"           "iris"
[29] "longley"          "morley"           "mtcars"           "npk"
[33] "pressure"         "quakes"           "randu"            "rock"
[37] "sleep"            "stackloss"        "swiss"            "trees"
[41] "warpbreaks"       "women"
 [1] "Reading contents of mtcars table..."
             row_names  mpg cyl  disp  hp drat    wt  qsec vs am gear carb
1            Mazda RX4 21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
2        Mazda RX4 Wag 21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
3           Datsun 710 22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
4       Hornet 4 Drive 21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
5    Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
6              Valiant 18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
7           Duster 360 14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
8            Merc 240D 24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
9             Merc 230 22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
10            Merc 280 19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
// rows omitted for brevity
22    Dodge Challenger 15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
23         AMC Javelin 15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
24          Camaro Z28 13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
25    Pontiac Firebird 19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
26           Fiat X1-9 27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
27       Porsche 914-2 26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
28        Lotus Europa 30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
29      Ford Pantera L 15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
30        Ferrari Dino 19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
31       Maserati Bora 15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
32          Volvo 142E 21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
[1] "Disconnecting database connection..."
```

READING TEXT FILES IN R

Listing 2.19 shows the content of `readtable.R` that illustrates how to read data from a text file in R.

LISTING 2.19: readtable.R

```
# read data and exclude header row
#df <- read.table("a.txt", header = FALSE)
#df

# read tab-delimited data and include header row
df <- read.table("a.txt", header = TRUE, sep = "\t", quote="\"")

#display contents of df:
df

### # Read in csv files
### df <- read.table("test.csv", header = FALSE, sep = ",")
### df <- read.csv("test.csv", header = FALSE)
### df <- read.csv2("test.csv", header= FALSE)
### # Inspect the result
### df
```

```
# Read a delimited file
### df <- read.delim("test_delim.txt", sep="$")
### df <- read.delim2("test_delim.txt", sep="$")
### # Inspect the result
### df
```

Listing 2.19 initializes the variable `df` with the contents of the text file `a.txt` without the header row (the first row) and then displays its contents.

The next code snippet also initializes `df` with the contents of `a.txt`, but this time it does include the header row, along with the tab character ("\t") as the column separator and the quote character (") as the character for quoted strings.

The next code block shows how to initialize `df` with the contents of the CSV file `test.csv`, with different values for `header` and `sep`. The final code block shows you how to initialize `df` with the contents of the text file `test_delim.txt`, with different values for `sep`. Launch the code in Listing 2.19 to see the following output:

```
          Name EmpId                          Address
1 Jane Edwards 12345 123 Main Street Chicago Illinois
2   John Smith 23456 432 Lombard Avenue SF California
                                                   V1
1                                Name,EmpId,Address
2 Jane Edwards,12345,123 Main Street Chicago Illinois
3   John Smith,23456,432 Lombard Avenue SF California
          Name EmpId                          Address
1 Jane Edwards 12345 123 Main Street Chicago Illinois
2   John Smith 23456 432 Lombard Avenue SF California
```

SAVING AND RESTORING OBJECTS IN R

Listing 2.20 shows the content of `save_restore.R` that illustrates how to read data from a text file in R.

LISTING 2.20: save_restore.R

```
print("Saving v to file saved_vector.Rdata")
v <- c(1,2,NA,4)
save(v, file="saved_vector.Rdata")

# dataframe:
mydf <- data.frame(
  attr1 <- c(1,2,3),
  attr2 <- c(4,5,6),
  attr3 <- c(7,8,9)
)
print("Saving df to file saved_dataframe.Rdata")
save(mydf, file="saved_dataframe.Rdata")

print("New contents of v:")
v <- c(-1234)
print(v)
```

```
print("New contents of mydf:")
mydf <- data.frame(c(-1234))
print(mydf)

print("Restoring v from file saved_vector.Rdata")
load("saved_vector.Rdata")

print("Restoring mydf from file saved_dataframe.Rdata")
load("saved_dataframe.Rdata")

print("Restored contents of v:")
print(v)

print("Restored contents of mydf:")
print(mydf)
```

Listing 2.20 consists of three parts. The first part defines and saves a vector `v` and also defines and saves a data frame `mydf`. The second part assigns different values to `v` and `mydf` to test whether they will be assigned the restored values.

The third part restores the values of `v` and `mydf`, which confirms that the code is working correctly and as expected. Launch the code in Listing 2.20 to see the following output:

```
[1] "Saving v to file saved_vector.Rdata"
[1] "Saving df to file saved_dataframe.Rdata"
[1] "New contents of v:"
[1] -1234
[1] "New contents of mydf:"
  c..1234.
1    -1234
[1] "Restoring v from file saved_vector.Rdata"
[1] "Restoring mydf from file saved_dataframe.Rdata"
[1] "Restored contents of v:"
[1]  1  2 NA  4
[1] "Restored contents of mydf:"
  attr1....c.1..2..3. attr2....c.4..5..6. attr3....c.7..8..9.
1                   1                   4                   7
2                   2                   5                   8
3                   3                   6                   9
```

DATA VISUALIZATION IN R

R supports an assortment of charts and graphs for displaying data in a graphical manner. In fact, R makes it surprisingly easy to render data in graphical form and to save that graphics data as a PNG file.

Some of the built-in chart-related functions can generate the following types of output:

- Bar charts
- Line graphs
- Histograms

- Pie charts
- Box plots

The next sections contain several basic code samples for rendering data as bar charts and pie charts.

WORKING WITH BAR CHARTS IN R (1)

Listing 2.21 shows the content of `barchart1.R` that illustrates how to display a bar chart in R.

LISTING 2.21: barchart1.R

```
# Create the data for the chart:
H <- c(7,12,28,3,41)

# Give the chart file a name:
png(file = "barchart1.jpg")

# Plot the bar chart:
barplot(H)

# Save the file:
dev.off()
```

In Listing 2.21, after initializing the variable `H` with five integer values, the `png()` function specifies the filename `barchart.jpg`, the function `barplot()` generates a bar chart, and then `dev()` saves the bar chart to the file `barchart.jpg`. Launch the code in Listing 2.21 that generates a bar chart. Figure 2.1 shows the contents of `barchart.jpg`.

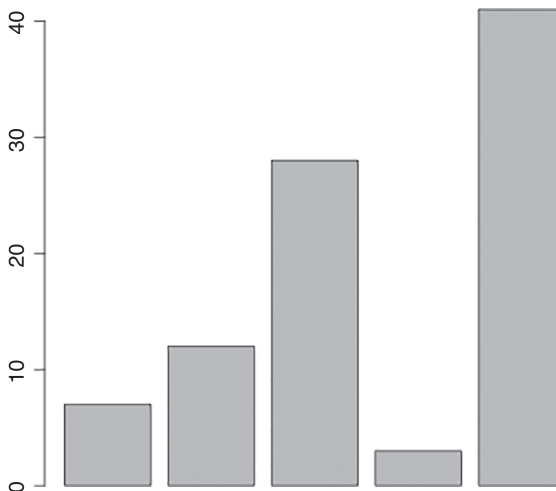

FIGURE 2.1 A bar chart created using the code in Listing 2.21.

WORKING WITH BAR CHARTS IN R (2)

Listing 2.22 shows the contents of `barchart2.R` that illustrates how to display a bar chart in R.

LISTING 2.22: barchart2.R

```
# Create the data for the chart:
H <- c(7,12,28,3,41)
M <- c("Mar","Apr","May","Jun","Jul")

# Give the chart file a name:
png(file = "barchart_months_revenue.png")

# Plot the bar chart:
barplot(H,names.arg=M,xlab="Month",ylab="Revenue",col="blue",
main="Revenue chart",border="red")

# Save the file:
dev.off()
```

Listing 2.22 extends the code in Listing 2.21 by specifying labels for the horizontal and vertical axes. Launch the code in Listing 2.22 to generate a bar chart.

Figure 2.2 shows the contents of `barchart_months_revenue.png`.

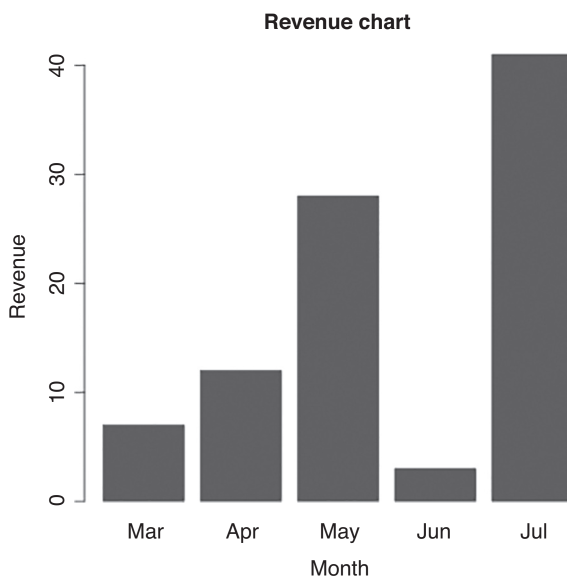

FIGURE 2.2 A bar chart created from the code in Listing 2.22.

WORKING WITH LINE GRAPHS IN R (1)

Listing 2.23 shows the content of `linegraph1.R` that illustrates how to display a line graph in R.

LISTING 2.23: linegraph1.R

```
# Create the data for the chart:
v <- c(7,12,28,3,41)

# Give the chart file a name:
png(file = "line_graph1.jpg")

# Plot the line graph:
plot(v,type = "o")

# Save the file:
dev.off()
```

Listing 2.23 is also similar to Listing 2.21, except that a line graph is generated from the `plot()` function. Launch the code in Listing 2.23 that generates a line graph.

Figure 2.3 shows the contents of `line_graph1.jpg`.

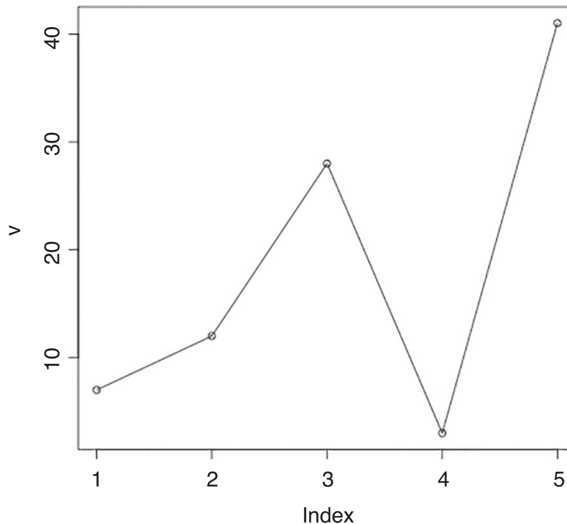

FIGURE 2.3 A line graph created from the code in Listing 2.23.

WORKING WITH LINE GRAPHS IN R (2)

Listing 2.24 shows the content of `linegraph_labels1.R` that illustrates how to display a line graph in R.

LISTING 2.24: linegraph_labels1.R

```
# Create the data for the graph:
v <- c(7,12,28,3,41)

# Give the chart file a name:
png(file = "line_graph_label_colored.jpg")
```

```
# Plot the bar chart:
plot(v, type = "o", col = "red",
     xlab = "Month", ylab = "Rain fall",
     main = "Rain fall chart")

# Save the file:
dev.off()
```

Listing 2.24 is similar to Listing 2.23 and also adds labels for the horizontal and vertical axes. Launch the code in Listing 2.24 to generate a line graph.

Figure 2.4 displays the contents of `line_graph_label_colored.jpg`.

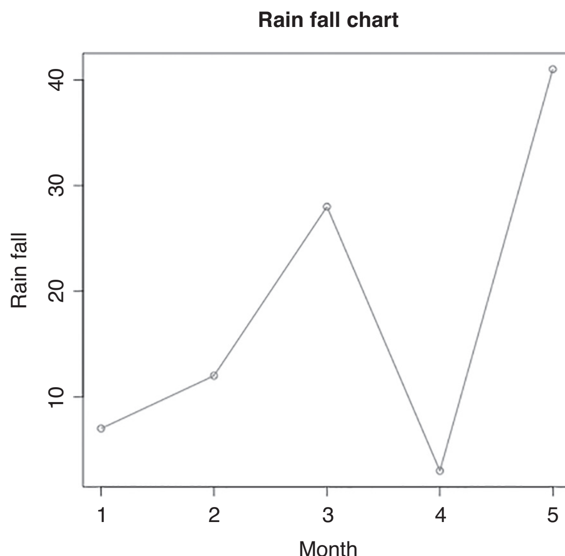

FIGURE 2.4 A labeled line graph created with the code from Listing 2.24.

WORKING WITH MULTI-LINE GRAPHS IN R

Listing 2.25 shows the content of `multilinegraph1.R` that illustrates how to display multi-line graphs in R.

LISTING 2.25: multilinegraph1.R

```
# Create the data for the chart:
v <- c(7,12,28,3,41)
t <- c(14,7,6,19,3)

# Give the chart file a name:
png(file = "line_chart_2_lines.jpg")

# Plot the bar chart.
plot(v, type = "o", col = "red",
     xlab = "Month", ylab = "Rain fall",
     main = "Rain fall chart")

lines(t, type = "o", col = "blue")
```

```
# Save the file:
dev.off()
```

Listing 2.25 is similar to Listing 2.24, except that two lines are generated. Launch the code in Listing 2.25 to generate a line chart. Figure 2.5 shows the contents of `line_chart2_lines.jpg`.

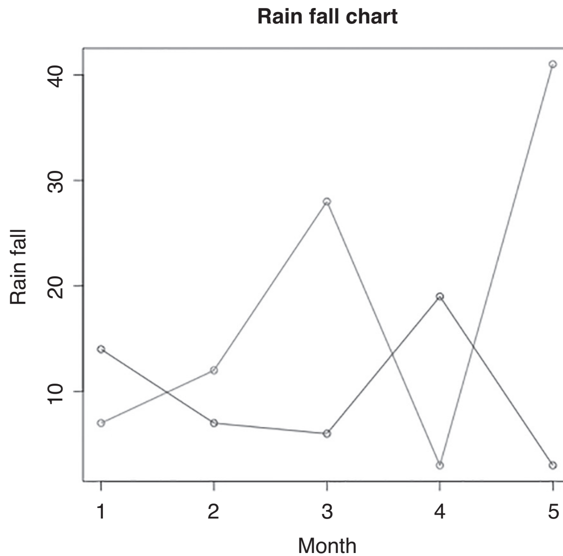

FIGURE 2.5 A labeled line chart created with the code from Listing 2.25.

WORKING WITH HISTOGRAMS IN R

Listing 2.26 shows the content of `histogram1.R` that illustrates how to display a histogram in R.

LISTING 2.26: histogram1.R

```
# Create data for the graph:
v <- c(9,13,21,8,36,22,12,41,31,33,19)

# Give the chart file a name:
png(file = "histogram1.png")

# Create the histogram:
hist(v,xlab = "Weight",col = "yellow",border = "blue")

# Save the file:
dev.off()
```

Listing 2.26 is similar to Listing 2.21, and generates a histogram instead of a bar chart. Launch the code in Listing 2.26 to generate a histogram. Figure 2.6 shows the contents of `histogram1.png`.

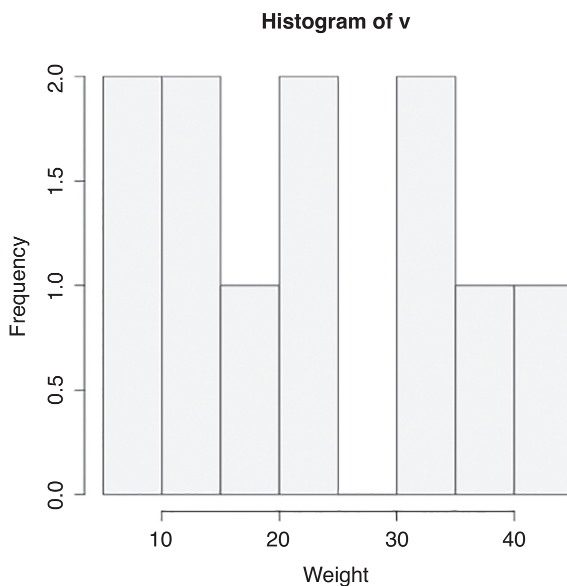

FIGURE 2.6 A histogram created with the code from Listing 2.26.

WORKING WITH SCATTER PLOTS IN R (1)

Listing 2.27 shows the content of `scatterplot1.R` that illustrates how to display a scatter plot in R.

LISTING 2.27: scatterplot1.R

```
input <- mtcars[,c('wt','mpg')]
print(head(input))

# Get the input values:
input <- mtcars[,c('wt','mpg')]

# Give the chart file a name:
png(file = "scatterplot.png")

# Plot the chart for cars with weight between
# 2.5 to 5 and mileage between 15 and 30:
plot(x = input$wt,y = input$mpg,
   xlab = "Weight",
   ylab = "Milage",
   xlim = c(2.5,5),
   ylim = c(15,30),
   main = "Weight vs Mileage"
)

# Save the file:
dev.off()
```

Listing 2.27 is similar to Listing 2.22, and the code generates a scatter plot instead of a bar chart. Launch the code in Listing 2.27 to generate a scatter plot. Figure 2.7 shows the contents of `scatterplot.png`.

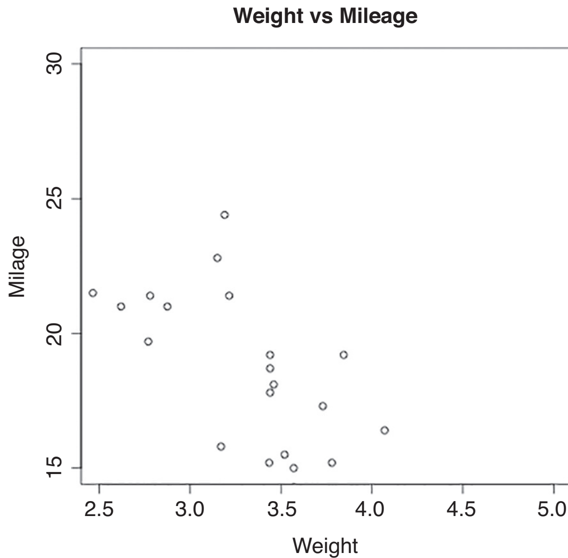

FIGURE 2.7 A scatter plot created with the code from Listing 2.27.

WORKING WITH SCATTER PLOTS IN R (2)

Listing 2.28 shows the content of `scatterplotMatrix1.R` that illustrates how to display a scatter plot in R.

LISTING 2.28: scatterplotMatrix1.R

```
input <- mtcars[,c('wt','mpg')]
print(head(input))

# Get the input values:
input <- mtcars[,c('wt','mpg')]

# Give the chart file a name:
png(file = "scatterplot_matrices.png")

# Plot the matrices between 4 variables giving 12 plots:
# One variable with 3 others and total 4 variables:

pairs(~wt+mpg+disp+cyl,data = mtcars,
   main = "Scatterplot Matrix")

# Save the file:
dev.off()
```

Listing 2.28 is similar to Listing 2.21, and generates a scatter plot instead of a bar chart. Launch the code in Listing 2.28 to generate a scatter plot. Figure 2.8 shows the contents of `scatterplot_matrices.png`.

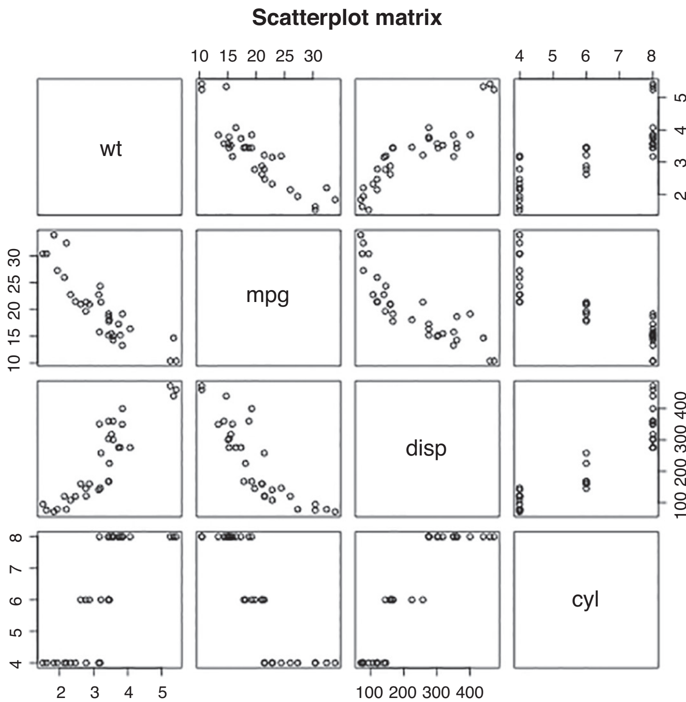

FIGURE 2.8 A scatter plot created with the code from Listing 2.28.

WORKING WITH BOX PLOTS IN R

Listing 2.29 shows the content of `boxplot1.R` that illustrates how to display a box plot in R.

LISTING 2.29: boxplot1.R

```
input <- mtcars[,c('mpg','cyl')]
print(head(input))

# Give the chart file a name:
png(file = "boxplot.png")

# Plot the chart:
boxplot(mpg ~ cyl, data = mtcars, xlab = "Number of Cylinders",
        ylab = "Miles Per Gallon", main = "Mileage Data")

# Save the file:
dev.off()
```

Listing 2.29 is similar to Listing 2.22, and generates a box plot instead of a bar chart. Launch the code in Listing 2.29 to generate a box plot. Figure 2.9 shows the contents of `boxplot.png`.

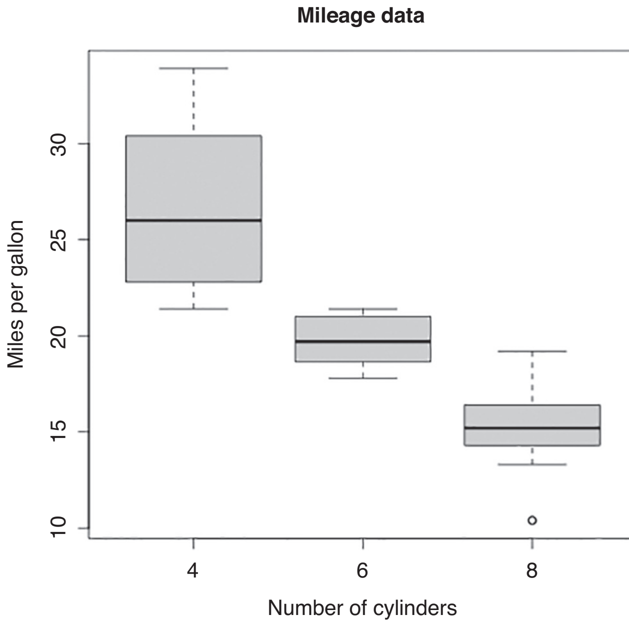

FIGURE 2.9 A box plot created using the code from Listing 2.29.

WORKING WITH PIE CHARTS IN R (1)

Listing 2.30 shows the content of `piechart1.R` that illustrates how to display a pie chart in R.

LISTING 2.30: piechart1.R

```
# Create data for the graph:
x <- c(21, 62, 10, 53)
labels <- c("London", "New York", "Singapore", "Mumbai")

# Give the chart file a name:
png(file = "piechart1.jpg")

# Plot the chart:
pie(x,labels)

# Save the file:
dev.off()
```

Listing 2.30 is similar to Listing 2.22, and generates a pie chart instead of a bar chart. Launch the code in Listing 2.30 to generate a pie chart. Figure 2.10 shows the contents of `piechart1.jpg`.

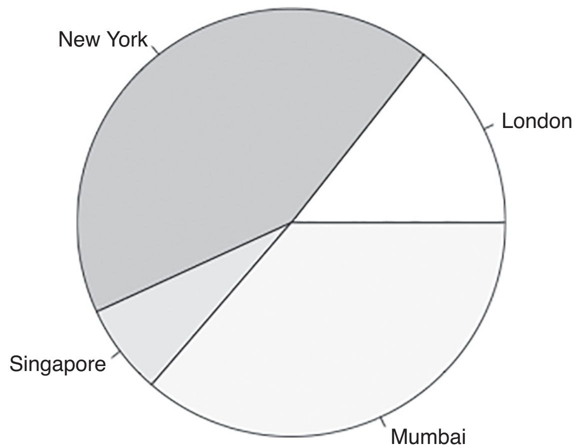

FIGURE 2.10 A pie chart created from the code in Listing 2.30.

WORKING WITH PIE CHARTS IN R (2)

Listing 2.31 shows the content of `piechart3D1.R` that illustrates how to display a 3D pie chart in R.

LISTING 2.31: piechart3D1.R

```
# Get the library:
library(plotrix)

# Create data for the graph:
x <-  c(21, 62, 10,53)
lbl <-  c("London","San Francisco","Rio de Janeiro","Rome")

# Give the chart file a name:
png(file = "3d_pie_chart.jpg")

# Plot the chart:
pie3D(x,labels = lbl,explode = 0.1, main = "Pie Chart of Countries ")

# Save the file:
dev.off()
```

Listing 2.31 is similar to Listing 2.22, and generates a pie chart instead of a bar chart. Launch the code in Listing 2.31 to generate a pie chart. Figure 2.11 shows the contents of `3d_piechart.jpg.`

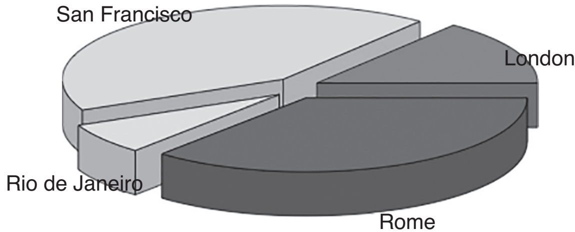

FIGURE 2.11 A pie chart created from the code in Listing 2.31.

SUMMARY

This chapter introduced you to loops in R, along with nested loops, in order to display data in a column format. Next, you learned about conditional logic, followed by code samples that illustrates how to write "if" statements, "if-then" statements, and "if-then-else" statements in R.

In addition, you learned about data frames in R, with an assortment of code samples involving data frames. You learned how to read the contents of text files into R data frames. Finally, you learned how to create visualizations in R involving bar charts, line graphs, scatter plots, and pie charts.

CHAPTER 3

WORKING WITH FUNCTIONS IN R

This chapter discusses some useful built-in R functions and how to define your own custom R functions. Later in this chapter, you will learn how to define recursive functions in R and use them to solve various tasks.

The first section discusses some built-in functions in R, such as statistical functions, trigonometric functions, and string-related functions. The second section contains examples of working with CSV files, XML files, and JSON files, and how to convert them to data frames in R. The third section explains how to define custom R functions.

The fourth section introduces you to recursion, which is a very powerful and elegant way to solve certain tasks. For example, you will learn how to define recursive functions for calculating factorial values, Fibonacci numbers, GCD (Greatest Common Divisor), and LCM (Lowest Common Multiple). Note that this section also illustrates how to calculate factorial values and Fibonacci numbers using an iterative algorithm.

For your convenience, the file `library_list.R` in Chapter 6 enables you to install more than 30 R packages that are used in this book. Feel free to add other R packages to this file.

NAN AND FUNCTIONS IN R

R provides a vast set of built-in functions, some of which you have already seen in previous chapters. For example, `toupper()` and `tolower()` are built-in functions that convert a string to uppercase and lowercase letters, respectively.

However, sometimes the existing functions in R do not provide the functionality that you need to perform specific tasks. R makes it very easy to define your own custom functions.

Listing 3.1 shows the content of `BasicFunctions.R` that illustrates how to handle NaN values in various R functions.

LISTING 3.1 BasicFunctions.R

```
x <- c(1,2,NA,3)
mean(x)
mean(x, na.rm=TRUE)

#check for missing values
is.na(x) # returns TRUE of x is missing
y <- c(1,2,3,NA)
is.na(y) # returns a vector (F F F T)

table(is.na(x))

sum(is.na(x))

sum(!is.na(x))
v <- c(NA, NA, 0.5, 1, 12, 15, 3)
summary(v)
v <- c(-1, "-nodata-", 0.5, 1, 12, 15, 3)
table(v)

summary(na.omit(x))
```

Listing 3.1 initializes the variable `x` as a list with four values (including `NA`) and then invokes the `mean()` method to calculate mean of those values. The second invocation of the `mean()` method specifies `na.rm=TRUE`, which is required for handling `NA` values. Launch the code in Listing 3.1 to see the following output:

```
[1] NA
[1] 2
[1] FALSE FALSE  TRUE FALSE
[1] FALSE FALSE FALSE  TRUE

FALSE  TRUE
    3     1
[1] 1
[1] 3
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
    0.5     1.0     3.0     6.3    12.0    15.0       2
v
     -1 -nodata-      0.5        1       12       15        3
      1        1        1        1        1        1        1
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    1.0     1.5     2.0     2.0     2.5     3.0
```

MATH-RELATED FUNCTIONS IN R

R supports a variety of math-related functions and trigonometric functions, as listed below:

- `sqrt()`
- `sum())`
- `cos()`
- `sin()`
- `tan()`
- `log(x)`
- `log10()`
- `exp()`
- `sqrt()`
- `round(x)`
- `signif(x)`
- `trunc(x) - rounding functions`
- `sqrt()`
- `sum()`
- `%% modulus`
- `%/% integer divisilln`
- `%*% matrix multiplication`
- `%o% outer product (a%o% equivalent to outer(a,b,"*"))`

The built-in trigonometric functions in R include `sin(x)`, `cos(x)`, `sin(x)`, `tan(x)`, `acos(x)`, `asin(x)`, `atan(x)`, and `atan2(y,x)`. With the exception of the function `atan2(y,x)`, the argument for all trigonometric functions in R is specified in radians (not degrees).

Listing 3.2 shows the content of `TrigFunctions.R` that illustrates how to use some math functions and trigonometric functions in R.

LISTING 3.2: TrigFunctions.R

```
# sine(π/2):
print("sin(pi/2):")
sin(pi/2)

# cosine(π):
print("cos(pi):")
cos(pi)

# tangent(π/3):
print("tan(pi/3):")
tan(pi/3)

# cotangent(π/3):
print("cotangent(pi/3):")
1/tan(pi/3)
```

```
#angle x where cos(x) = -1:
print("acos(-1):")
acos(-1)

#angle x where tan(x) = 0.5:
print("atan(0.5):")
atan(0.5)

#atan2() take the y and x values as arguments:
print("atan2(1,2):")
atan2(1,2)
```

Listing 3.2 invokes the trigonometric functions `sin()`, `cos()`, and `tan()` with the value `π/3`, followed by several other trigonometric functions. Launch the code in Listing 3.2 to see the following output:

```
[1] "sin(pi/2):"
[1] 1
[1] "cos(pi):"
[1] -1
[1] "tan(pi/3):"
[1] 1.732051
[1] "cotangent(pi/3):"
[1] 0.5773503
[1] "acos(-1):"
[1] 3.141593
[1] "atan(0.5):"
[1] 0.4636476
[1] "atan2(1,2):"
[1] 0.4636476
[1] 2
```

STRING-RELATED FUNCTIONS IN R

The following functions are useful for preprocessing tasks in NLP, and they involve the `tm_map()` function:

- `removeNumbers()`
- `removePunctuation()`
- `removeWords`
- `stemDocument()`
- `stripWhiteSpace()`
- `tolower()`

The `removeNumbers()` function removes digits in a text string. For example, "This Is Short123!" is replaced with "This Is Short!".

The `removePunctuation()` function removes punctuation in a text string. For example, "This Is Short123!" is replaced with "This Is Short123." Keep in mind that this function also removes characters such as emojis.

The `removeWords()` function removes the stop words. For example, "This Is Short123!" is replaced with "Short123!".

The stemCompletion() function takes as arguments the stemmed words and a dictionary of complete words, whereas the stemDocument() function replaces words with their stem.

The stripWhiteSpace() function removes whitespaces and tab characters in a text string. For example, "This Is   Short123!" is replaced with "this is short123!". This function removes leading, trailing, and embedded white spaces and tab characters.

The tolower() function replaces alphabetic characters with their lowercase counterpart. For example, "This Is Short123!" is replaced with "this is short123!". As you can see, digits and punctuation are unaffected by the tolower() function. Note that converting a string to lowercase can lose information: "rose" might have originally been "Rose" (a proper name).

THE GSUB() FUNCTION IN R

R supports the gsub() function that enables you to perform string-based substitutions, along with support for regular expressions.

Listing 3.3 shows the content of gsub_examples.R that illustrates how to use gsub() and basic regular expressions in R.

LISTING 3.3: gsub_examples.R

```
library(tm)

str <- c("123", "this","is","a","sentence!?")

print(paste0("str:",str))
print(paste0(str))
print(paste0(str, collapse=" "))

print(paste0("=> Replace non-alpha with X and spaces with Z:",collapse=" "))
str2 = gsub(pattern="\\W", replace="X",str)
print(paste0(str2, collapse="Z"))

print(paste0("=> Replace digits with blanks and Y for blanks:",collapse=" "))
str3 = gsub(pattern="\\d", replace=" ",str2)
print(paste0(str3, collapse="Y"))

print(paste0("=> Replace initial 't' with periods Y and Z for
blanks:",collapse=" "))
str4 = gsub(pattern="\\bt", replace=".....",str2)
print(paste0(str4, collapse="Z"))

print(paste0("=> Replace one-character words with SINGLE:",collapse=" "))
str5 = gsub(pattern="\\b[A-z]\\b", replace=" SINGLE ",str2)
print(paste0(str5, collapse="Z"))

print(paste0("=> Remove whitespaces:",collapse=" "))
str6 = stripWhitespace(str2)
print(paste0(str6, collapse=""))
```

Listing 3.3 initializes the variable str as a text string and displays its contents. Next, str2 is initialized as the result of replacing non-alphabetic characters with the letter X. Note that uppercase patterns are

the "opposite" of lowercase patterns. Hence, the pattern `\\W` machines any non-alphabetic character because the pattern `\\w` matches any alphabetic character.

The next portion of Listing 3.3 initializes `str3` as the result of replacing digits by blank spaces in `str2` via the pattern \\d. The next portion of Listing 3.3 initializes `str4` as the result of replacing tab characters (\t) with five adjacent periods (.). The final portion of Listing 3.3 initializes `str6` as the result of removing white spaces from the string `str2`. Launch the code in Listing 3.3 to see the following output:

```
[1] "str:123"        "str:this"       "str:is"         "str:a"
[5] "str:sentence!?"
[1] "123"        "this"       "is"         "a"          "sentence!?"
[1] "123 this is a sentence!?"
[1] "=> Replace non-alpha with X and spaces with Z:"
[1] "123ZthisZisZaZsentenceXX"
[1] "=> Replace digits with blanks and Y for blanks:"
[1] "   YthisYisYaYsentenceXX"
[1] "=> Replace initial 't' with periods Y and Z for blanks:"
[1] "123Z.....hisZisZaZsentenceXX"
[1] "=> Replace one-character words with SINGLE:"
[1] "123ZthisZisZ SINGLE ZsentenceXX"
[1] "=> Remove whitespaces:"
[1] "123thisisasentenceXX"
```

MISCELLANEOUS BUILT-IN FUNCTIONS

The following miscellaneous functions are described briefly, and more information is available in the online documentation:

- `grep()`: regular expressions
- `identical()`: test if two objects are identical
- `length()`: returns the number of elements in vector
- `ls()`: list objects in current environment
- `order(x)`: list the sorted element numbers of x
- `range(x)`: minimum and maximum
- `rep(x,n)`: repeat the number x, n times
- `rev(x)`: elements of x in reverse order
- `seq(x,y,n)`: sequence (x to y, spaced by n)

In addition, R supports the following file-related functions:

- `getwd()`: return working directory
- `setwd()`: set working directory
- `choose.files()`: get path to a file
- `sort()`: sorts the #s in a list

You can find additional information by reading the online documentation.

SET FUNCTIONS IN R

R provides several built-in operators for set-related operations, as listed below:

- `union()`
- `intersect()`
- `setdiff()`
- `setequal()`

The set functions `union()`, `intersect()`, `setdiff()`, and `setequal()` discard duplicates in the arguments. Moreover, these set functions apply `as.vector()` to their arguments, which coerces factors to character vectors.

Listing 3.4 shows the content of `SetFunctions.R` that illustrates how to use some arithmetic functions and set-related functions in R.

LISTING 3.4: SetFunctions.R

```
print("one:")
(one <- c(sort(sample(1:20, 8)), NA))

print("two:")
(two <- c(sort(sample(5:30, 5)), NA))

union(one, two)
intersect(one, two)
setdiff(one, two)
setdiff(two, one)
setequal(one, two)

# is.element(x, y) => identical to x %in% y.
# the elements of one that are in two (9)
is.element(one, two)

# length 6
# the elements of two that are in one (6)
is.element(two, one)
```

Listing 3.4 starts by displaying a sample set `one` that consists of integers between 1 and 20, followed by the string `NA`. Next, another sample set `two` that consists of 5 integers between 5 and 30 is displayed, as well as the string `NA`.

The next portion of Listing 3.4 involves set-related functions `union()` and `intersect()` that display the union and intersection, respectively, of the sets `one` and `two`. Three more set functions are invoked to display the elements in set `one` that are not in set `two`, then the elements in set `two` that are not in set `one`, and then the `setequal()` function that displays `TRUE` if `one` and `two` are equal (otherwise `FALSE` is displayed).

The final portion of Listing 3.4 displays `TRUE` for each element of set `one` that is in set `two` (otherwise `FALSE` is displayed), followed by similar output

when the role of set `one` and set `two` is reversed. Launch the code in Listing 3.4 to see the following output:

```
[1] "one:"
[1]  3  4  7  8 12 13 14 16 NA
[1] "two:"
[1]  9 11 13 20 28 NA
[1]  3  4  7  8 12 13 14 16 NA  9 11 20 28
[1] 13 NA
[1]  3  4  7  8 12 14 16
[1]  9 11 20 28
[1] FALSE
[1] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE
[1] FALSE  TRUE FALSE FALSE FALSE  TRUE
```

R supports the built-function `eigen()` for determining eigenvalues and eigenvectors in linear algebra. In addition, R supports the built-in function `deriv()` for calculating symbolic and algorithmic derivatives.

THE "APPLY" FAMILY OF BUILT-IN FUNCTIONS

The following functions are in the "apply family" of R functions, and they are similar to the `map()` function that is available in many other languages:

- `apply`: apply a function (e.g., mean)
- `lapply`: read multiple files
- `sapply`: apply for lists or vectors

Listing 3.5 shows the content of `apply_functions.R` that illustrates how replace a missing value in a column with the mean of the other values in that same column.

LISTING 3.5: apply_functions.R

```
# Create example data
my_data <- data.frame(x1 = 1:5,
                      x2 = 2:6,
                      x3 = 3,
                      x4 = -1)

print(paste0("Initial array values:",collapse=" "))
my_data

# invoke the apply() function:
print(paste0("Row-wise sum of values:",collapse=" "))
apply(my_data, 1, sum)

print(paste0("Column-wise sum of values:",collapse=" "))
apply(my_data, 2, sum)
```

```
# create example list:
my_list <- list(1:5,
                letters[1:3],
                123)

print(paste0("Heterogenous list of values:",collapse=" "))
my_list

print(paste0("Invoke the lapply() function:",collapse=" "))
lapply(my_list, length)

print(paste0("Invoke the sapply() function:",collapse=" "))
sapply(my_list, length)
# 5 3 1

print(paste0("Invoke the vapply() function:",collapse=" "))
vapply(my_list, length, integer(1))
```

Listing 3.5 starts by initializing `my_data` and then displaying its contents. The next pair of code blocks display the row-wise sum and the column-wise sum, respectively, of the values in `my_data`.

The next portion of Listing 3.5 initializes a matrix with heterogenous values, and then invokes the functions `apply()`, `sapply()`, and `vapply()`. These three functions calculate row-based and column-based sums (see the comment lines). Launch the code in Listing 3.5 to see the following output:

```
[1] "Initial array values:"
  x1 x2 x3 x4
1  1  2  3 -1
2  2  3  3 -1
3  3  4  3 -1
4  4  5  3 -1
5  5  6  3 -1
[1] "Row-wise sum of values:"
[1]  5  7  9 11 13
[1] "Column-wise sum of values:"
x1 x2 x3 x4
15 20 15 -5
[1] "Heterogenous list of values:"
[[1]]
[1] 1 2 3 4 5

[[2]]
[1] "a" "b" "c"

[[3]]
[1] 123

[1] "Invoke the lapply() function:"
[[1]]
[1] 5
```

```
[[2]]
[1] 3

[[3]]
[1] 1

[1] "Invoke the sapply() function:"
[1] 5 3 1
[1] "Invoke the vapply() function:"
[1] 5 3 1
```

The next section discusses the `dplyr` package, which merits an entire chapter for a detailed description, but we'll only cover some of its more salient features.

THE "MUST LEARN" DPLYR PACKAGE IN R

R supports the `dplyr` package, which is an extremely powerful R package for managing data frames in R. The `dplyr` package enables you to select columns and filter rows, as well as find distinct values and overlapping values. Moreover, this package enables you to perform group-by aggregation on datasets.

Some frequently used `dplyr` APIs are listed below:

- `arrange()`
- `filter()`
- `mutate()`
- `select()`
- `summarize()`

For more information, see the following sites:

- *https://online.datasciencedojo.com/blogs/data-manipulation-and-exploration-with-dplyr*
- *https://genomicsclass.github.io/book/pages/dplyr_tutorial.html*
- *https://bensstats.wordpress.com/2021/09/14/pythonmusings-6-dplyr-in-python-first-impressions-of-the-siuba-%E5%B0%8F%E5%B7%B4-module/*

The preceding functions can also be used with the `group_by()` function that displays data in a per-group basis.

The `arrange()` API changes the order of rows (do not confuse this with the `arange()` API that is available in other languages).

The `filter()` API enables you to select a subset of rows based on a Boolean expression. For example, if a column contains integer values, you can select the rows for which the integer value is even (or even and larger than 10, or even and between 20 and 40, and so forth). There is no practical limit to the `Boolean` expression, and you can use any combination of logical operators,

such as OR, AND, and NOT. The mutate() API creates new columns (this API performs a column insert operation). The select() API selects columns from a data frame. The summarise() API provides a summary of the values in columns. The group_by() API is similar to the GROUP keyword in SQL statements.

Before we continue, make sure that you have installed the dplyr package, which can be performed by either of the following code snippets:

```
# install tidyverse:
install.packages("tidyverse",repos = "https://cloud.r-project.org")

# install only dplyr:
install.packages("dplyr",repos = "https://cloud.r-project.org")
```

Listing 3.7 shows the content of dplyr-mtcars.R that illustrates how to use the some of the functionality in the dplyr package.

LISTING 3.7: dplyr-mtcars.R

```
library(datasets)
library(dplyr)

# select columns by name:
print("=> mpg,cyl,dps,qsec:")
selecn = select(mtcars,mpg,cyl,disp,qsec)
head(selecn)

# data filter
#filter(mtcars, mpg > 20)
print("Filter by mpg > 20 and cyl > 5:")
f = filter(mtcars, mpg > 20 & cyl > 5)
head(f)

# add a new column
#dm=mutate(mtcars, TempInC = (Temp - 32) * 5 / 9)
#head(dm)

# summarize and group by data
#print("=> summarize by mpg:")
#summarise(mtcars, mpg)

# group: average wind value per month
# Month is the basis of grouping
#print("=> group by cl and mean by mpg:")
#summarise(group_by(mtcars, cyl), mean(mpg, na.rm = TRUE))
```

Listing 3.7 starts with references to the datasets and dplyr R libraries. Next, the R select() function initializes the variable selectn with values from the built-in mtcars dataset that pertain to the attributes mpg, cyl, disp, and qsec. After the first five rows of selectn are displayed, the variable f is initialized with the rows of mtcars whose mpg is greater than 20 and whose cyl value is greater than 5. Launch the code in Listing 3.7 to see the following output:

```
[1] "=> mpg,cyl,dps,qsec:"
                   mpg cyl disp  qsec
Mazda RX4         21.0   6  160 16.46
Mazda RX4 Wag     21.0   6  160 17.02
Datsun 710        22.8   4  108 18.61
Hornet 4 Drive    21.4   6  258 19.44
Hornet Sportabout 18.7   8  360 17.02
Valiant           18.1   6  225 20.22
[1] "Filter by mpg > 20 and cyl > 5:"
   mpg cyl disp  hp drat    wt  qsec vs am gear carb
1 21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
2 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
3 21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
```

OTHER USEFUL R PACKAGES

R supports the following useful packages that provide useful APIs, some of which you will see discussed in greater detail later in this chapter:

- `caret`
- `data.table`
- `forcats`
- `ggplot2`
- `lubridate`
- `reticulate`
- `shiny`
- `stringr`
- `tidyr`

The `caret` package (an acronym for Classification and Regression Training) facilitates the model training step, with support for regression tasks as well as classification tasks.

The `data.table` package provides APIs for extracting subsets of rows and columns of data and also perform data aggregation operations with the `by_group()` API. This package works well for large datasets.

The `forcats` package is designed to work with categorial variables, which are factors in R. The APIs in this package enable you to change the order in which factors are displayed by various criteria.

The `ggplot2` package is for creating graphics effects, and another graphics-related package is `plotly`.

The `lubridate` package contains a set of date-related APIs that enable you to work with various date formats as well as time zones, daylight savings time, leap years, and so forth.

The `reticulate` package enables you to use Python code and R code together, in R programs and also in RStudio.

The `shiny` package is for creating interactive Web applications.

The `stringr` package contains a comprehensive set of APIs for string-related functionality. In addition, `stringy` is built on top of the `stringi` package: the latter contains string-related functions that are not included in strings.

The `tidyr` package is well-suited for data in which each cell contains a single value and columns are variable.

THE PIPE OPERATOR %>%

The `pipe` operator `%>%` enables you to pipe output from one function to the input of another function (that's why it's called a pipe). In addition, `dplyr` imports this operator from the `magrittr` package. Instead of nesting functions (reading from the inside to the outside), piping reads the functions from left to right.

Listing 3.8 shows the content of `pipe1.R` that illustrates how to sort an array of random numbers in R.

LISTING 3.8: pipe1.R

```
library(magrittr)

x <- rnorm(5)
print(paste0("content of x:"))
print(paste0(x))

# Update value of x and assign it to x
x %>% abs %>% sort
print(paste0("content of x:"))
print(paste0(x))
```

Listing 3.8 starts with references to the `magrittr` R library, followed by initializing the variable `x` with 5 random values, and then displaying those values via the `rnorm()` function. The next portion of Listing 3.8 "pipes" the values in `x` to the `abs()` function that returns the absolute value of the numbers in its input, and the result is then passed to the `sort()` function that sorts its input values. Launch the code in Listing 3.8 to see the following output:

```
[1] "content of x:"
[1] "1.23041905548521"   "0.426501888248532"  "1.08469130060061"
[4] "-0.995749885650863" "-1.02953061190553"
[1] "content of x:"
[1] "0.426501888248532" "0.995749885650863" "1.02953061190553"
[4] "1.08469130060061"  "1.23041905548521"
```

Listing 3.9 shows the content of `pipe2.R` that illustrates how to invoke additional R functions in a pipeline.

LISTING 3.9: pipe2.R

```
library(magrittr)

x <- c(1,2,3,4,5)
print(paste0("content of x:"))
print(paste0(x))

# Perform operations on x:
x %>% log() %>%
    diff() %>%
    exp() %>%
    round(1)
```

Listing 3.9 initialized `x` with the integers between 1 and 5 inclusive and displays those values. The main portion of the code in Listing 3.9 passes the values of `x` to the `log()` function, then the `diff()` function, then the `exp()` function, and lastly rounds the final output values to one decimal place. Launch the code in Listing 3.9 to see the following output:

```
[1] "content of x:"
[1] "1" "2" "3" "4" "5"
[1] 2.0 1.5 1.3 1.2
```

Listing 3.10 shows the content of `pipe3.R` that illustrates how to perform multiple operations on a vector of numbers in R.

LISTING 3.10: pipe3.R

```
library(magrittr)

x <- c(-50,20,30,12,-88,100,-500)
print("x:")
print(x)

x %>% abs %>% sort
print("sorted:")
print(x)

#The tee operator %T>%;
rnorm(200) %>%
matrix(ncol = 2) %T>%
plot %>%
colSums
```

Listing 3.10 initializes the variable `x` with a list of integer values and then passes `x` to the `abs()` and `sort()` functions, which calculate the absolute value of each input value, followed by sorting the resulting list of non-negative numbers.

The next portion of Listing 3.10 invokes the `rnorm()` function that generates 100 random numbers from a normal distribution. This set of numbers is passed to the `matrix()` function that generates a 100 × 2 matrix before

calculating the column sum of both columns. Now launch the code in Listing 3.10 to see the following output:

```
[1]  -50   20   30   12  -88  100 -500
[1] "x:"
[1]  12  20  30  50  88 100 500
[1]  3.912239 -4.359854
```

The next section shows you how to work with CSV files in R, followed by sections that illustrate how to work with XML documents and JSON files in R.

WORKING WITH CSV FILES IN R

R provides built-in functions for reading the contents of a CSV file. Listing 3.11 shows the content of the CSV file `input.csv` that is referenced in Listing 3.12.

LISTING 3.11: input.csv

```
id,name,salary,start_date,dept
1,Rick,623.3,2012-01-01,IT
2,Dan,515.2,2013-09-23,Operations
3,Michelle,611,2014-11-15,IT
4,Ryan,729,2014-05-11,HR
 ,Gary,843.25,2015-03-27,Finance
6,Nina,578,2013-05-21,IT
7,Simon,632.8,2013-07-30,Operations
8,Guru,722.5,2014-06-17,Finance
```

In Listing 3.11, there are 8 rows of comma-delimited data records, the fifth of which is missing an `id` value.

Listing 3.12 shows the content of `readinputcsv1.R` that illustrates how to read the contents of a CSV file in R.

LISTING 3.12: readinputcsv1.R

```
# Some European countries use a ";" as the delimiter in .csv files
# Use read.csv2() as above instead of read.csv

data <- read.csv("input.csv")

print(is.data.frame(data))
print(ncol(data))
print(nrow(data))

print(paste0("First Six Rows of CSV file:",paste=" "))
head(data)

print(paste0("Entire CSV file:",paste=" "))
print(data)
```

Listing 3.12 invokes the built-in R function `read.csv()` to initialize the variable `data` with the contents of the CSV file `input.csv`. The next section

in Listing 3.12 displays `TRUE` if the data is a data frame (and `FALSE` otherwise), followed by the number of columns and the number of rows in the data. Launch the code in Listing 3.12 to see the following output:

```
[1] TRUE
[1] 5
[1] 8
[1] "First Six Rows of CSV file: "
  id     name salary start_date       dept
1  1     Rick 623.30 2012-01-01         IT
2  2      Dan 515.20 2013-09-23 Operations
3  3 Michelle 611.00 2014-11-15         IT
4  4     Ryan 729.00 2014-05-11         HR
5 NA     Gary 843.25 2015-03-27    Finance
6  6     Nina 578.00 2013-05-21         IT
[1] "Entire CSV file: "
  id     name salary start_date       dept
1  1     Rick 623.30 2012-01-01         IT
2  2      Dan 515.20 2013-09-23 Operations
3  3 Michelle 611.00 2014-11-15         IT
4  4     Ryan 729.00 2014-05-11         HR
5 NA     Gary 843.25 2015-03-27    Finance
6  6     Nina 578.00 2013-05-21         IT
7  7    Simon 632.80 2013-07-30 Operations
8  8     Guru 722.50 2014-06-17    Finance
```

In addition to support for delimited text files, R supports another common data format called XML, which is discussed in the next section.

WORKING WITH XML IN R

R provides built-in functions for reading the contents of an XML document. Listing 3.13 shows the content of `readxml.R` that illustrates how to read the contents of an XML file in R.

LISTING 3.13: readxml.R

```
#install.packages("XML",repos = "https://cloud.r-project.org")
library(XML)

# Give the input file name to the function:
result <- xmlParse(file = "input.xml")

# Print the result:
print("Contents of XML file:")
print(result)
```

Listing 3.13 loads the XML library and the methods library, and then initializes the variable `result` with the result of parsing the XML file `input.xml`. The last code snippet in Listing 3.4 shows the contents of the XML file. Launch the code in Listing 3.13 to see the following output:

```
<records>
   <employee>
      <id>1</id>
      <name>rick</name>
      <salary>623.3</salary>
      <startdate>1/1/2012</startdate>
      <dept>it</dept>
   </employee>
// details omitted for brevity
   <employee>
      <id>8</id>
      <name>guru</name>
      <salary>722.5</salary>
      <startdate>6/17/2014</startdate>
      <dept>finance</dept>
   </employee>
</records>
```

Listing 3.14 shows the contents of the XML document `input.xml` that is referenced in Listing 3.13.

LISTING 3.14: input.xml

```
<records>
   <employee>
      <id>1</id>
      <name>rick</name>
      <salary>623.3</salary>
      <startdate>1/1/2012</startdate>
      <dept>it</dept>
   </employee>

   <employee>
      <id>2</id>
      <name>dan</name>
      <salary>515.2</salary>
      <startdate>9/23/2013</startdate>
      <dept>operations</dept>
   </employee>

    <employee>
      <id>3</id>
      <name>michelle</name>
      <salary>611</salary>
      <startdate>11/15/2014</startdate>
      <dept>it</dept>
   </employee>

   <employee>
      <id>4</id>
      <name>ryan</name>
      <salary>729</salary>
      <startdate>5/11/2014</startdate>
      <dept>hr</dept>
    </employee>
```

```
    <employee>
       <id>5</id>
       <name>gary</name>
       <salary>843.25</salary>
       <startdate>3/27/2015</startdate>
       <dept>finance</dept>
    </employee>

    <employee>
       <id>6</id>
       <name>nina</name>
       <salary>578</salary>
       <startdate>5/21/2013</startdate>
       <dept>it</dept>
    </employee>

    <employee>
       <id>7</id>
       <name>simon</name>
       <salary>632.8</salary>
       <startdate>7/30/2013</startdate>
       <dept>operations</dept>
    </employee>

    <employee>
       <id>8</id>
       <name>guru</name>
       <salary>722.5</salary>
       <startdate>6/17/2014</startdate>
       <dept>finance</dept>
    </employee>
</records>
```

READING AN XML DOCUMENT INTO AN R DATAFRAME

The previous section showed you how to read an XML document in R and this section shows you how to populate an R data frame with an XML document. Listing 3.15 shows the content of `readxmltodataframe.R` that illustrates how to read the contents of an XML file into an R data frame.

LISTING 3.15: readxmltodataframe.R

```
library(XML)

# Convert the input xml file to a data frame:
xmldataframe <- xmlToDataFrame("input.xml")

print("Contents of XML dataframe:")
print(xmldataframe)
```

Listing 3.15 loads the XML library and then initializes the variable `xmldataframe` with the result of parsing the XML file `input.xml`. The last code snippet in Listing 3.15 shows the contents of the data frame

xmldataframe, which contains the contents of the XML file input.xml. Launch the code in Listing 3.15 to see the following output:

```
[1] "Contents of XML dataframe:"
  id     name salary  startdate       dept
1  1     rick  623.3   1/1/2012         it
2  2      dan  515.2  9/23/2013 operations
3  3 michelle    611 11/15/2014         it
4  4     ryan    729  5/11/2014         hr
5  5     gary 843.25  3/27/2015    finance
6  6     nina    578  5/21/2013         it
7  7    simon  632.8  7/30/2013 operations
8  8     guru  722.5  6/17/2014    finance
```

WORKING WITH JSON IN R

Listing 3.16 shows the content of readjson.R that illustrates how to read the contents of a JSON file in R.

LISTING 3.16: readjson.R

```
# Load the package required to read JSON files:
library("rjson")

# Give the input file name to the function:
result <- fromJSON(file = "input.json")

# Print the result:
print(result)
```

Listing 3.16 loads the rjson library and then initializes the variable result with the result of parsing the JSON file input.json. The last code snippet in Listing 3.16 shows the content of result, which contains the contents of the JSON file input.json. Launch the code in Listing 3.16 to see the following output:

```
[1] "Contents of JSON file:"
$ID
[1] "1" "2" "3" "4" "5" "6" "7" "8"

$Name
[1] "Rick"     "Dan"      "Michelle" "Ryan"     "Gary"     "Nina"     "Simon"
[8] "Guru"

$Salary
[1] "623.3"  "515.2"  "611"    "729"    "843.25" "578"    "632.8"  "722.5"

$StartDate
[1] "1/1/2019"   "9/23/2020"  "11/15/2021" "5/11/2021"  "3/27/2020"
[6] "5/21/2020"  "7/30/2020"  "6/17/2021"

$Dept
[1] "IT"         "Operations" "IT"         "HR"         "Finance"
[6] "IT"         "Operations" "Finance"
```

Listing 3.17 shows the content of the JSON file `input.json` that is referenced in Listing 3.16.

LISTING 3.17: input.json

```
{
   "ID":["1","2","3","4","5","6","7","8" ],
   "Name":["Rick","Dan","Michelle","Ryan","Gary","Nina","Simon","Guru" ],
   "Salary":["623.3","515.2","611","729", "843.25","578","632.8","722.5" ],

   "StartDate":[ "1/1/2019","9/23/2020","11/15/2021","5/11/2021","3/27/2020",
"5/21/2020","7/30/2020","6/17/2021"],
   "Dept":[ "IT","Operations","IT","HR","Finance","IT","Operations","Finance"]
}
```

In addition to reading the contents of a JSON file, you can create an R data frame that contains JSON-based data, as discussed in the next section.

READING A JSON FILE INTO AN R DATAFRAME

R also provides the ability to read JSON files into R data frames. Listing 3.18 shows the content of `jsontodataframe.R` that illustrates how to read the contents of a JSON file into an R data frame.

LISTING 3.18: jsontodataframe.R

```
#install.packages("rjson",repos = "https://cloud.r-project.org")
library(rjson)

# Give the input file name to the function:
result <- fromJSON(file = "input.json")

# Convert JSON file to a data frame:
json_data_frame <- as.data.frame(result)

print(json_data_frame)
```

Listing 3.18 loads the `rjson` library and then initializes the variable `result` with the result of parsing the JSON file `input.json`. Next, the variable `json_data_frame` is populated with the result of converting the `result` variable to an R data frame. The last code snippet in Listing 3.18 shows the contents of `result`, which contains the contents of the JSON file `input.json`. Launch the code in Listing 3.18 to see the following output:

```
  ID     Name Salary  StartDate       Dept
1  1     Rick  623.3   1/1/2019         IT
2  2      Dan  515.2  9/23/2020 Operations
3  3 Michelle    611 11/15/2021         IT
4  4     Ryan    729  5/11/2021         HR
5  5     Gary 843.25  3/27/2020    Finance
6  6     Nina    578  5/21/2020         IT
7  7    Simon  632.8  7/30/2020 Operations
8  8     Guru  722.5  6/17/2021    Finance
```

STATISTICAL FUNCTIONS IN R

One of the strengths of R is the plethora of built-in statistical functions, such as `mean()`, `std()`, `var()`, and `cov()`.

Listing 3.19 shows the content of `mean-value1.R` that illustrates how to calculate the mean of a set of numbers in R.

LISTING 3.19: mean-value1.R

```
# create a vector:
x <- c(12,7,3,4.2,18,2,54,-21,8,-5)

# find the mean:
result.mean <- mean(x)
print(result.mean)

# create a vector:
x <- c(12,7,3,4.2,18,2,54,-21,8,-5,NA)

# find the mean:
result.mean <-  mean(x)
print(result.mean)

# drop NA values and find the mean:
result.mean <- mean(x,na.rm = TRUE)
print(result.mean)

# create a vector:
x <- c(12,7,3,4.2,18,2,54,-21,8,-5)

# find the mean:
result.mean <- mean(x,trim = 0.3)
print(result.mean)

# create a vector:
x <- c(12,7,3,4.2,18,2,54,-21,8,-5)

# find the median:
median.result <- median(x)
```

Listing 3.19 initializes the vector `x` and then calculates the mean of the values in `x`, the first time without an `NA` value and the second time with an `NA` value, which requires `na.rm`. The next portion of Listing 3.19 calculates the mean value based on a "trimmed" set of numbers in `x`, followed by the median value. Launch the code in Listing 3.19 to see the following output:

```
[1] 8.22
[1] NA
[1] 8.22
[1] 5.55
[1] 5.6
```

SUMMARY FUNCTIONS IN R

This section contains a code sample with other built-in statistical functions in R. Listing 3.20 shows the content of `summary-values.R` that illustrates how to calculate the mean, weighted mean, min, max, median, and standard deviation of a set of numbers in R.

LISTING 3.20: summary-values.R

```
# define a sample of 50 values:
x <- sample(1:200, size = 50, replace = TRUE)
print(paste0("mean(x):   ", mean(x)))
print(paste0("min(x):    ", min(x)))
print(paste0("max(x):    ", max(x)))
print(paste0("median(x):", median(x)))

# make a copy of x:
y <- x

# randomly set 10 values to NA:
y[sample(1:50, size=10, replace=TRUE)] <- NA
print("mean of y with NA values:")
print(mean(y, na.rm=TRUE))

# Calculate a weighted mean:
scores  <- c(250, 100, 80, 360)
weights <- c(1/2, 1/4, 1/8, 1/8)
print("scores:")
print(scores)
print("weights:")
print(weights)

wm <- weighted.mean(x=scores, w=weights)
print(paste0("weighted mean: ",wm))

print(paste0("Variance of x: ",var(x)))
print(paste0("STD of y:", sd(y, na.rm=TRUE)))
```

Listing 3.20 initializes the vector `x` via the `sample()` function that selects a set of 50 values with replacements (in this example) from the integers that range from 1 to 200. After initializing `x`, the next code block displays the `mean()`, `min()`, `max()`, and `median()` values of `x`.

The next portion of Listing 3.20 initializes `y` as a copy of `x` and then random replaces 10 of its values with `NA`, and then calculates the mean of `y` with those `NA` values.

The final portion of Listing 3.20 initializes the variables scores and weights, displays their values, and then computes the weighted mean of scores and weights using the R function `weighted.mean()`. In addition, the variance of `x` and the standard deviation of `y` are displayed.

Launch the code in Listing 3.20 to see the following output:

```
[1] "mean(x):  100.12"
[1] "min(x):   2"
[1] "max(x):   196"
[1] "median(x):104"
[1] "mean of y with NA values:"
[1] 92.53659
[1] "scores:"
[1] 250 100  80 360
[1] "weights:"
[1] 0.500 0.250 0.125 0.125
[1] "weighted mean: 205"
[1] "Variance of x: 3314.10775510204"
[1] "STD of y:56.4916354697647"
```

DEFINING A CUSTOM FUNCTION IN R

A *custom function* in R is a function that is written by you. Such a function has the following syntax, where the ellipsis indicates the location of your custom R code:

```
Myfunc <- function(args) { ... }
```

Listing 3.21 shows the content of `CustomFunctions.R` that defines a custom function to double a number and a custom function to square a number in R.

LISTING 3.21: CustomFunctions.R

```
double <- function(a)
{
   return (2*a)
}

square <- function(a)
{
   return (a*a)
}

print(paste0("3 doubled: ", double(3)))
print(paste0("3 squared: ", square(3)))
```

Listing 3.21 defines the custom R functions `double()` and `square()` that double and square a number, respectively. Launch the code in Listing 3.11 to see the following output:

```
[1] "3 doubled: 6"
[1] "3 squared: 9"
```

Listing 3.22 shows the content of `CustomFunctionsLoop.R` that defines a custom function to double a number and a custom function to square a number in R.

LISTING 3.22: CustomFunctionsLoop.R

```
b <- 4

# prints squares of numbers in sequence:
new.function <- function(a)
{
   for(i in 1:a)
   {
      b <- i^2
      print(b)
   }
}
new.function(6)
```

Listing 3.22 initializes the variable `b` with the value 4 and then defines the function `new.function` that iterates through a range of numbers and displays the squares of those numbers. The last code snippet in Listing 3.22 invokes the R function `new.function()` with the value 6, which generates the following output:

```
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
[1] 36
```

You can also define functions that are invoked recursively, which is a topic that is discussed in the next section.

RECURSION IN R

Recursion is powerful and elegant, yet it can be difficult to debug recursion-based functions. Some examples of recursion in R that you will see later in this chapter involve calculating factorial values and Fibonacci numbers.

Sometimes it's easier to define a recursive algorithm to solve a task than to do so with a non-recursive function. However, recursive functions have a non-recursive "counterpart." It can sometimes be extremely difficult to define the non-recursive function that performs the same functionality as the recursive function.

The following example includes a recursive function as well as a non-recursive function for calculating factorial values. (It's much simpler to calculate Fibonacci values using a recursive function.)

CALCULATING FACTORIAL VALUES IN R (NON-RECURSIVE)

Listing 3.23 shows the content of `Factorial1.R` that illustrates how to calculate factorial values *without* recursion in R.

LISTING 3.23: Factorial1.R

```
# factorial:  fact(n) = n!
num = 5
factorial = 1

# check is the number is negative, positive or zero
if(num < 0) {
    print("Sorry, factorial does not exist for negative numbers")
} else if(num == 0) {
    print("The factorial of 0 is 1")
} else {
    for(i in 1:num) {
        factorial = factorial * i
    }
    print(paste("The factorial of", num ,"is",factorial))
}
```

Listing 3.23 starts by prompting for a number and then initializing the variable factorial with the value 1. Some error checking is performed on the input value, and if the input is an integer greater than 1, a for loop is executed that iteratively multiplies the variable factorial with the numbers from 1 to `num` (i.e., the input number). Launch the code in Listing 3.23 to see the following output:

```
[1] "The factorial of 5 is 120"
```

CALCULATING FACTORIAL VALUES IN R (RECURSIVE)

Listing 3.24 shows the content of `Factorial2.R` that illustrates how to work with recursion to compute the factorial value of a positive integer in R.

LISTING 3.24: Factorial2.R

```
# factorial:  fact(n) = n*fact(n-1)

recur_factorial <- function(n) {
    if(n <= 1) {
        return(1)
    } else {
        return(n * recur_factorial(n-1))
    }
}

recur_factorial(5)
```

Listing 3.24 defines the recursive function `recur_factorial()` that implements the formula for factorial values that is displayed in the initial comment in Listing 3.24. The final code snippet in Listing 3.24 invokes the `recur_factorial()` function with the number 5, after which the factorial value of 5 is displayed. Launch the code in Listing 3.24 to see the following output:

```
[1] 120
```

CALCULATING FIBONACCI NUMBERS IN R (NON-RECURSIVE)

Listing 3.25 shows the content of `Fibonacci1.R` that illustrates how to calculate Fibonacci numbers *without* recursion in R.

LISTING 3.25: Fibonacci1.R

```
# Fibonacci:  F(n) = F(n-1) + F(n-2)
nterms = 20

# first two terms
n1 = 0
n2 = 1
count = 2

# check if the number of terms is valid
if(nterms <= 0) {
    print("Please enter a positive integer")
} else {
    if(nterms == 1) {
        print("Fibonacci sequence:")
        print(n1)
    } else {
        print("Fibonacci sequence:")
        print(n1)
        print(n2)

        while(count < nterms) {
            nth = n1 + n2
            print(nth)
            # update values
            n1 = n2
            n2 = nth
            count = count + 1
        }
    }
}
```

Listing 3.25 initializes terms as 20, which equals the number of Fibonacci numbers that will be calculated, as well as the two "start" values for the Fibonacci sequence. By convention, these values are 0 and 1.

The next portion of Listing 3.25 contains conditional logic and since `nterms` is greater than 1, the innermost else block is executed. This code block

contains a while loop that iteratively computes the third through twentieth Fibonacci values. Launch the code in Listing 3.25 to see the following output:

```
[1] "Fibonacci sequence:"
[1] 0
[1] 1
[1] 1
[1] 2
[1] 3
[1] 5
[1] 8
[1] 13
[1] 21
[1] 34
[1] 55
[1] 89
[1] 144
[1] 233
[1] 377
[1] 610
[1] 987
[1] 1597
[1] 2584
[1] 4181
```

CALCULATING FIBONACCI NUMBERS IN R (RECURSIVE)

Listing 3.26 shows the content of `Fibonacci2.R` that illustrates how to calculate Fibonacci numbers with *recursion* in R.

LISTING 3.26: Fibonacci2.R

```
# Fibonacci:  F(n) = F(n-1) + F(n-2)
recurse_fibonacci <- function(n) {
    if(n <= 1) {
        return(n)
    } else {
        return(recurse_fibonacci(n-1) + recurse_fibonacci(n-2))
    }
}

# take input from the user
nterms = as.integer(readline(prompt="How many terms? "))

# check if the number of terms is valid
if(nterms <= 0) {
    print("Plese enter a positive integer")
} else {
    print("Fibonacci sequence:")
    for(i in 0:(nterms-1)) {
        print(recurse_fibonacci(i))
    }
}
```

Listing 3.26 defines the recursive function `recurse_fibonacci()` that uses recursion to calculate Fibonacci numbers. The recursion occurs in the else block of code, which involves a for loop that invokes the `recurse_fibonacci()` function. Launch the code in Listing 3.26 to see the following output:

```
[1] "Fibonacci sequence:"
[1] 0
[1] 1
[1] 1
[1] 2
[1] 3
[1] 5
[1] 8
[1] 13
[1] 21
[1] 34
[1] 55
[1] 89
[1] 144
[1] 233
[1] 377
[1] 610
[1] 987
[1] 1597
[1] 2584
[1] 4181
```

CONVERT A DECIMAL INTEGER TO A BINARY INTEGER IN R

Listing 3.27 shows the content of `converttobinary.R` that illustrates how to convert an integer to a binary number in R.

LISTING 3.27: converttobinary.R

```
# Convert decimal num into binary num via recursive function

convert_to_binary <- function(n) {
      if(n > 1) {
          convert_to_binary(as.integer(n/2))
      }
      cat(n %% 2)
   }

print(paste0("52 in binary:"))
convert_to_binary(52)
cat("\n")
```

Listing 3.27 defines the function `convert_to_binary()` that uses recursion to generate the binary string for a decimal number. Given an initial value `n`, this function invokes itself with `n/2` as long as `n` is greater than 1: when it does equal 1, the value `n %% 2` is displayed, which equals `n` modulo 2 (i.e., either 0 or 1).

As the recursive sequence of invocations "unwinds," the value n %% 2 is repeatedly displayed with a different value of n, which generates the binary representation of the initial value of n. Launch the code in Listing 3.27 to see the following output:

```
[1] "52 in binary:"
110100
```

CALCULATING THE GCD OF TWO INTEGERS IN R

Listing 3.28 shows the content of `GCD.R` that illustrates how to work use recursion to find the GCD (greatest common divisor) of two positive integers in R.

LISTING 3.28: GCD.R

```
# find the GCD of two input numbers
gcd <- function(x, y) {
    # choose the smaller number
    if(x > y) {
        smaller = y
    } else {
        smaller = x
    }
    for(i in 1:smaller) {
        if((x %% i == 0) && (y %% i == 0)) {
            gcd = i
        }
    }
    return(gcd)
}

# take input from the user
num1 = 10
num2 = 24

print(paste("The G.C.D. of", num1,"and", num2,"is", gcd(num1, num2)))
```

Listing 3.28 defines the function `gcd()` that calculates the GCD of two positive integers. The first step initializes the variable `smaller` with the smaller of `x` and `y`. The second step involves a loop that iterates from 1 to smaller and updates the value of `gcd` with the loop variable `i` whenever `x` and `y` are divisible by `i`. When the loop is completed, the variable `gcd` contains the GCD of `x` and `y`.

The next portion of Listing 3.28 initializes `num1` and `num2` and invokes the function `gcd()` with these two variables, after which the GCD of these two variables is displayed. Launch the code in Listing 3.28 to see the following output:

```
[1] "The G.C.D. of 10 and 24 is 2"
```

CALCULATING THE LCM OF TWO INTEGERS IN R

Listing 3.29 shows the content of `LCM.R` that illustrates how to calculate the LCM (lowest common multiple) of two positive integers in R.

LISTING 3.29: LCM.R

```
#  find the GCD of two input numbers
gcd <- function(x, y) {
    gcd1 <- 1
    # choose the smaller number
    if(x > y) {
        smaller = y
    } else {
        smaller = x
    }
    for(i in 1:smaller) {
        if((x %% i == 0) && (y %% i == 0)) {
            gcd1 = i
        }
    }

    return(gcd1)
}

# the LCM involves a simple operation:
lcm <- function(x,y) {
    return((x * y)/gcd(x,y))
}

x = 10
y = 24

print(paste("The L.C.M. of", x,"and", y,"is", lcm(x, y)))
```

Listing 3.29 defines the custom R function `gcd(x,y)` shown in an earlier example, followed by a code snippet that calculates the LCM of two positive integers. The final portion of Listing 3.29 calculates the LCM of `x` and `y`. Launch the code in Listing 3.29 to see the following output:

```
[1] "The L.C.M. of 10 and 24 is 120"
```

SUMMARY

This chapter introduced you to built-in functions in R, with examples of some of the more useful functions that you will probably use in your code. Next, you learned how to define your own functions in R so that you can perform custom tasks for which there aren't any convenient built-in R functions.

You learned about recursion in R, and how to define recursive functions to calculate various quantities, such as the factorial value of a positive integer, Fibonacci numbers, the GCD of two positive integers, and the LCM of two positive integers.

CHAPTER 4

NLP CONCEPTS (I)

This chapter introduces you to NLP, starting with a high-level introduction to some major language groups and the substantive grammatical differences among the languages. You will learn some basic concepts in NLP, such as text normalization, stop words, stemming, and lemmatization (the dictionary form of words), POS (parts of speech) tagging, and NER (named entity recognition). This chapter contains a highly eclectic mix of topics.

While some NLP algorithms are mentioned in this chapter, the relevant code samples are provided in Chapter 6. Depending on your NLP background, you might decide to read the sections in a non-sequential fashion. If your goal is to proceed quickly to the code samples, you can skip some sections in this chapter and later return to read those omitted sections.

The first part of the chapter introduces you to NLP and a brief history of the major stages of NLP. You will also learn about NLP applications, use cases, NLU, and NLG. Then you will learn about word sense disambiguation. This section only provides a very brief description of these topics, some of which can fill entire books and full-length courses.

The second part of this chapter discusses various NLP techniques and the major steps in an NLP-related process. You will also learn about standard NLP-related tasks, such as text normalization, tokenization, stemming, lemmatization, and the removal of stop words. As you will see, some of these tasks (e.g., tokenization) involve implicit assumptions that are not true for all languages.

The final section introduces NER (Named Entity Recognition) and topic modeling, which pertains to finding the main topic(s) in a text document.

The next section starts with an introduction to NLP, followed by various NLP-related concepts.

WHAT IS NLP?

Natural Language Processing (NLP) is an important branch of AI that pertains to processing human languages with machines. You are surrounded by NLP through voice assistants, search engines, and machine translation services whose purpose is to simplify your tasks and aspects of your daily life.

NLP faces a variety of challenges, such as determining the context of words and their many meanings in different sentences in a document or corpus. Other challenging tasks include identifying emotions (such as irony and sarcasm), statements with multiple meanings, and sentences with contradictory statements.

Facebook has created an impressive model for language translation, called the M2M model, which was trained on more than 2,000 languages and provides translation between any pair of 100 languages.

In high level terms, there are three main approaches to solving NLP tasks: rule-based (oldest), traditional machine learning, and neural networks (most recent). Rule-based approaches, which can utilize regular expressions, work well on various NLP tasks. Traditional machine learning for NLP tasks (which includes various types of classifiers) involves training a model on a training set and then making inferences on a test set of data. This approach is still useful for handling NLP tasks such as sequence labeling.

By contrast, neural networks take *word embeddings* (vector-based representations of words) as input and are then trained using backward error propagation. Examples of neural network architectures include CNNs, RNNs, and LSTMs. Moreover, there has been significant research in combining deep learning with NLP, which has resulted in state of the art (SOTA) results.

In particular, the transformer architecture (which relies on the concept of attention) has eclipsed earlier neural network architectures. In fact, the transformer architecture is the basis for BERT, which is a pre-trained NLP model with 1.5 billion parameters, along with numerous other pre-trained models that are based (directly or indirectly) on BERT. Chapter 7 introduces the transformer architecture and BERT-related models.

Regardless of the methodology, NLP algorithms involve samples in the form of documents or collections of documents containing text. A corpus can vary in size, and can be domain specific and/or language specific. In some cases, such as GPT-3 (discussed in Chapter 7), models are trained on a corpus of 500 gigabytes of text.

As a historical aside, the Brown University Standard Corpus of Present-Day American English, also called Brown Corpus, was created during the 1960s for linguistics. This corpus contains 500 samples of English-language text, with a total of approximately 1,000,000 words. More information about this corpus is here:

https://en.wikipedia.org/wiki/Brown_Corpus

As a concrete example of NLP, consider the task of determining the main topics in a document. While this task is straightforward for a text document

consisting of a few pages, finding the main topics of a hundred documents, each of which might contain several hundred pages, is impractical to complete via a manual process (and if you gave this work to multiple people, you would have to pay them).

Fortunately, there is an NLP technique called *topic modeling* that performs the task of analyzing documents and determining the main topics in those documents. This type of document analysis can be performed in a variety of situations that involve large amounts of text. Keep in mind that NLP can help you analyze documents that contain structured data as well as unstructured data (or a combination of both types of data).

The Evolution of NLP

NLP has undergone many changes since the mid-twentieth century, the earliest of which might seem primitive when you compare them with modern NLP. Several major stages of NLP are listed below, starting from 1950 up until 2020 or so, that highlight the techniques that were commonly used in NLP.

- 1950s-1980s: rule-based systems
- 1990s-2000s: corpus-based statistics
- 2000s-2014: machine learning
- 2014-2020: deep learning

Early NLP (1950s-1990s) focused primarily on rule-based systems, which means that those techniques used a lot of conditional logic. When you consider the structure of a sentence in English, it's often of the form subject-verb-object. However, a sentence can have one or more subordinate clauses, each of which can involve multiple nouns, prepositions, adjectives, and adverbs.

Even more complex is maintaining a reference between two sentences, such as the following: "Yesterday was a hot day and many people were uncomfortable. I wonder what that means for the coming days."

Although you can infer the meaning of the word "that" in the second sentence, the correct interpretation is difficult using rule-based methods. This era of NLP did perform some statistical analyses of sentences to predict which words were more likely to follow a given word.

The next phase of NLP (1990s-2000s) shifted away from a rule-based analysis toward a primarily statistical analysis of collections of documents. The third phase involved machine learning for NLP, which embraced algorithms, such as decision trees, and structures, such as Markov chains. Once again, an important task involved predicting the next word in a sequence of words.

The most recent phase of NLP is the past decade and the combination of neural networks with NLP. In fact, 2012 was a significant turning point involving Convolutional Neural Networks (CNNs) that achieved a breakthrough in

terms of accuracy specifically for classifying images. Researchers then learned how use CNNs in order to analyze audio waves and perform NLP tasks.

The use of CNNs for NLP then evolved into the use of Recurrent Neural Networks (RNNs) and Long Short Term Memory (LSTM), which are two architectures that belong to deep learning, for even better accuracy.

These architectures have been superseded by the Transformer architecture (also considered a part of deep learning) that was developed by Google toward the end of 2017. Transformer-based architectures (there are many of them) have achieved state-of-the-art performance that surpass all the previous attempts in the NLP arena.

A WIDE-ANGLE VIEW OF NLP

This section contains aspects of NLP, as well as many NLP applications and use cases, which are summarized in this list:

- NLP applications
- NLP use cases
- NLU (Natural Language Understanding)
- NLP (Natural Language Generation)
- Text Summarization
- Text Classification

The following subsections provide additional information for each topic in the preceding list.

NLP Applications and Use Cases

There are many useful and well-known applications that rely on NLP, some of which are listed here:

- Chatbots
- Search (text and audio)
- Advertisement
- Automated translation
- Sentiment analysis
- Document classification
- Speech recognition
- Customer support

In particular, chatbots are receiving a great deal of attention because of their increasing ability to perform tasks that previously required human interaction.

Sentiment analysis is a subset of text summarization that attempts to determine the attitude or emotional reaction of a speaker toward a particular topic (or in general). Possible sentiments are positive, neutral, and negative, which are typically represented by the numbers 1, 0, and –1, respectively.

Document classification is a generalization of sentiment analysis and typically involves more than three possible flags per article:

https://towardsdatascience.com/natural-language-processing-pipeline-decoded-f97a4da5dbb7

In addition to the preceding list of sample applications, there are many use cases for NLP, some of which are listed below:

- Question Answering
- Filter email messages
- Detect fake news
- Improve clinical documentation
- Automatic Text Summarization
- Sentiment Analysis and Semantics
- Machine Translation and Generation
- Personalized marketing

Some of the use cases in the preceding list (such as sentiment analysis) are discussed in later chapters.

NLU and NLG

NLU is an acronym for Natural Language Understanding, and although you might not find many books or articles about this topic, it's a very significant subset of NLP. In high-level terms, NLU attempts to understand human language to determining the context of a text string or document. NLU addresses various NLP tasks, such as sentiment analysis and topic classification.

Another extremely important NLU task is called *relation extraction*, which is the task of extracting semantic relations that may exist in a text string. Moreover, the sources of input text can be from chatbots, documents, blog posts, and so forth. As a simple example, consider this block of text and notice the different meanings of the pronouns "he" and "them:"

"John lived in France and he attended an international school. Mary lived in Germany and she also attended an international school. Dave lived in London and met both of them in Paris. One of these days, when he has some free time, they will meet up again. Steve met all of them on New Year's Eve."

Although the preceding paragraph is easy for humans to understand, it poses some challenges for NLU, such as determining the correct answers to the following questions:

1. Who does the first occurrence of "he" refer to?
2. Who does the second occurrence of "he" refer to? Is it ambiguous?
3. Who does the first occurrence of "them" refer to?
4. Who does the second occurrence of "them" refer to? Is it ambiguous?

One of the challenges of human language involves the correct interpretation of words that are used ambiguously in a sentence, and such ambiguity can be classified into several types. For example, *lexical ambiguity* occurs when a word has multiple meanings, which can change the meaning of a sentence that contains that word. One approach to handling this type of ambiguity involves POS (Parts Of Speech) techniques, which is illustrated in the chapter with NLTK content.

Another type of ambiguity is *syntactical ambiguity*, also called *grammatical ambiguity*, which occurs when a *sequence* of words (instead of a single word) has multiple meanings.

Yet another type of ambiguity is *referential ambiguity*, which can occur when a noun in one location is referenced elsewhere via a pronoun, and the reference is not completely clear.

In addition to NLU, another very important subset of NLP is Natural Language Generation (NLG), which is the process of producing meaningful phrases and sentences in the form of natural language from some internal representation. One impressive example of NLG is the ability of GPT-3 (discussed in Chapter 7) to generate meaningful responses to a wide variety of questions.

NLP can be used to analyze speech (not discussed in this book), words, and the structure of sentences. As such, we need to become acquainted with text classification, which is the topic of the next section.

What is Text Classification?

Text classification is a supervised approach for determining the category or class of a text-based corpus, which can be in the form of a blog post, the contents of a book, the contents of a web page, and so forth. The possible classes are known in advance, and they do not change; the classes are often (but not always) mutually-exclusive.

Text classification involves examining text to determine the nature of its content, such as

- topic labeling (the major topics of a document)
- the sentiment of the text (positive or negative)
- the human language of the text
- categorizing products on websites
- whether it's spam

However, most text-based data is unstructured, which complicates the task of analyzing text-based documents. From a business perspective, machine learning text classification algorithms are valuable when they structure and analyze text in a cost-effective manner, thereby expediting business processes and decision-making processes.

Text classification is important for customer service, which can involve routing customer requests based on the (human) language of the text, determining

if it's a request for assistance (products or services), or detecting issues with products.

Note that some older text classification algorithms are based on the Bag of Words (BoW) that only determines word frequency in documents. The BoW algorithm is explained in Chapter 5, and there are code samples for the BoW algorithm in Chapter 6.

One more thing: text summarization is related to text classification (but not discussed in this book).

INFORMATION EXTRACTION AND RETRIEVAL

The purpose of information extraction is to automatically extract structured information from one or more sources, which could contain unstructured data in documents. For example, an article might provide the details of an IPO of a successful start-up, or the acquisition of one company by another company. Information extraction would involve generating a summary sentence from the contents of the article. In a larger context, information extraction is related to topic modeling (i.e., finding the main topics in a document), which is discussed toward the end of this chapter.

Information extraction also requires information retrieval, where the latter involves methods for indexing and classifying large documents. Information extraction involves various subtasks, such as identifying named entities (i.e., nouns for people, places, and companies), automatically populating a template with information from an article, or extracting data from tables in a document.

As a simple example, suppose that a program regularly scrapes (retrieves) the contents of HTML pages to summarize their contents. One of the first tasks that must be performed is data cleaning, which in this case involves removing HTML tags, removing punctuation, converting text to lower case, and then splitting sentences into tokens (words). Fortunately, the BeautifulSoup Python library is well suited for the preceding tasks.

Another area of great interest in NLP is the proliferation of chatbots, which interact with users to provide information (such as the directions or hours of operation) or perform specific tasks (such as make reservations, book hotels, or car rentals).

WORD SENSE DISAMBIGUATION

Up until several years ago, word sense disambiguation was an elusively difficult task because words can be overloaded (i.e., possess multiple meanings). A well-known NYT article describes one humorous misinterpretation in machine learning. The following sentence was translated into Russian and then translated from Russian into English:

The spirit is willing, but the flesh is weak.
The result of the second translation is here:
The vodka is good, but the meat is rotten.

Here is the link to the NYT article:

https://www.nytimes.com/1983/04/28/business/technology-the-computer-as-translator.html

As another simple example of an overloaded word, consider the following four sentences:

You can bank on that result.
You can take that to the bank.
You see that river bank?
Bank the car to the left.

In the preceding four sentences, the word "bank" has four meanings. The task of determining the meaning of a word requires some type of context. In the not too distant past, the state of word sense disambiguation resulted in a precipitous drop in enthusiasm vis-a-vis machine learning. However, the situation has dramatically improved during the past several years. For example, in 2018, Microsoft developed a system for translating from Chinese to English; the system had an accuracy that was comparable to humans.

NLP TECHNIQUES IN ML

Earlier, you briefly learned about NLU (Natural Language Understanding) and NLG (Natural Language Generation). The purpose of NLU is to "understand" a section of text, and then use NLG to generate a suitable response (or find a suitable response from a repository). This type of task is also related to Question Answering and Knowledge Extraction.

Since there are many types of NLP tasks, there are also many NLP techniques that have been developed, some of which are listed below:

- text embeddings
- text summarization
- text classification
- sentence segmentation
- POS (part-of-speech tagging)
- NER (Named entity recognition)
- word sense disambiguation
- text categorization
- topic modeling
- text similarity
- syntax and parsing
- language modeling
- dialogs
- probabilistic parsing
- clustering

A subset of the items in the preceding list is discussed in this chapter, and in some cases, there are associated Python code samples in Chapter 6.

NLP Steps for Training a Model

Although the specific set of text-related tasks depends on the specific task that you're trying to complete, the following set of steps is common:

[1] convert words to lowercase
[1] noise removal
[2] normalization
[3] text enrichment
[3] stopword removal
[3] stemming
[3] lemmatization

The number in brackets in the preceding bullet list indicates the type of task. Specifically, the values [1], [2], and [3] indicate "must do," "should do," and "task dependent," respectively.

TEXT NORMALIZATION AND TOKENIZATION

Text normalization involves several tasks, such as the removal of unwanted hash tags, emojis, URLs, special characters such as "&," "!," and "$." However, you might need to make decisions regarding some punctuation marks.

First, what about the period (.) punctuation mark? If you retain every period in a dataset, consider whether to treat this character as a token during the tokenization step. However, if you remove every period from a dataset, this will also remove every ellipsis (three consecutive periods), and also the period from the strings "Mr.," "U.S.A.," and "P.O." If the dataset is small, perform a visual inspection of the dataset, and if the dataset is very large, try inspecting several smaller and randomly selected subsets of the original dataset.

Second, although you might think it's a good idea to remove question marks (?), the opposite is true: in general, question marks enable you to identify questions (as opposed to statements) in a corpus.

Third, you also need to determine whether to remove numbers, which can convey quantity when they are separate tokens ("1,000 barrels of oil") or they can be data entry errors when they are embedded in alphabetic strings. For example, it's acceptable to remove the 99 from the string "large99 oranges," but what about the 99 in "99large oranges?"

Another standard normalization task involves converting all words to lowercase (*case folding*). Chinese characters do not have uppercase text, so converting text to lowercase is unnecessary. Text normalization is entirely unrelated to normalizing database tables in an RDBMS, or normalizing (scaling) numeric data in machine learning tasks. The task of converting categorical (character) data into a numeric counterpart.

Although converting letters to lowercase (aka case folding) is a straightforward task, this step can be problematic. For instance, accents are optional for uppercase French words, and after case folding some words do require an accent. A simple example is the French word *peche*, which means *fish* or *peach* with one accent mark, and *sin* with a different accent mark. The Italian counterparts are *pesce*, *pesca*, and *peccato*, respectively, and there is no issue regarding accents marks. Incidentally, the plural of *pesce* is *pesci* (so *Joe Pesci* is *Joe Fish or Joe Fishes*, depending on whether you are referring to one type of fish or multiple types of fish). To a lesser extent, converting English words from uppercase to lowercase can cause issues: is the word “stone” from the noun “stone” or from the surname “Stone?”

After normalizing a dataset, tokenization involves “splitting” a sentence, paragraph, or document into its individual words (tokens). The complexity of this task can vary significantly between languages, depending on the nature of the alphabet of a specific language. In particular, tokenization is straightforward for Indo-European languages because those languages use a space character to separate words.

However, even though tokenization can be straightforward when working with regular text, the process can be more challenging when working with biomedical data that contains acronyms and a higher frequency use of punctuation. One NLP technique for handling acronyms is NER (Named Entity Recognition), which is discussed later in this chapter.

Word Tokenization in Japanese

Unlike most languages, the use of a space character in Japanese text is optional. Unlike virtually all other languages, Japanese supports multiple alphabets, and sentences often contain a mixture of these alphabets. Specifically, Japanese supports Romanji (essentially the English alphabet), Hiragana, Katakana (used exclusively for words imported to Japanese from other languages), and Kanji characters.

As a simple example, navigate to Google translate in your browser and enter the following Romanji sentence, written without white spaces, which means “I gave a book to my friend” in English:

watashiwatomodachinihonoagemashita

The partially correct translation is the following text in Hiragana:

わたしはこれだけのほげあげました

Now enter the same Romanji sentence, but this time with spaces between each word, as shown here:

watashi wa tomodachi ni hon o agemashita

Now Google translate produces the following correct translation in Hiragana:

私はともだちに本をあげました

Notice that the preceding sentence also contains Kanji characters, starting with the Kanji character 私 for "watashi" (I) and the Kanji character 本 for "hon" (book).

Mandarin and Cantonese are two more languages that involves complicated tokenization. Both of these languages are tonal, and they use pictographs instead of alphabets. An alternative to Mandarin is Pinyin, which is the romanization of the sounds in Mandarin. Interestingly, Mandarin has 6 tones, but only 4 of those tones are commonly used, whereas Cantonese has 9 tones (with no counterpart to Pinyin).

As a simple example, the following sentences are in Mandarin and in Pinyin, respectively, and their translation into English is "How many children do you have":

```
你有几个孩子
Nǐ yǒu jǐ gè háizi
Ni3 you3 ji3ge4 hai2zi
```
(digits instead of tone marks)

The second and third sentences above are both Pinyin. The third sentence contains the numbers 2, 3, and 4 that correspond to the second, third, and fourth tones, respectively, in Mandarin. The third sentence is used in situations where the tonal characters are not supported (such as in older browsers). Navigate to Google Translate and type the following words for the source language:

ni you jige haizi

Select Mandarin for the target language and you will see the following translation:

how many kids do you have

The preceding translation is quite impressive, when you consider that the tones were omitted: different tones can significantly change the meaning of words. If you are skeptical, look at the translation of the string "ma" when it's written with the first tone, then the second tone, and again with the third tone and the fourth tone: the meanings of these four words are entirely unrelated.

Tokenization can be performed via regular expressions (which are discussed in one of the appendices) and rule-based tokenization. However, rule-based tokenizers are not well-equipped to handle rare words or compound words that are very common in languages such as German. In Chapter 6, you will see code samples involving the NLTK tokenizer and the SpaCY tokenizer for tokening one or more English sentences.

Text Tokenization with Unix Commands

Text tokenization can be performed not only in Python but also from the Unix command line. For example, consider the text file `words.txt` whose contents are shown here:

lemmatization: removing word endings edit distance: measure the distance between two words based on the number of changes needed based on the inner product of 2 vectors a metric for determining word similarity

The following command illustrates how to tokenize the preceding paragraph using several Unix commands that are connected via the Unix pipe ("|") symbol:

```
tr -sc 'A-Za-z' '\n' < words.txt | sort | uniq
```

The output from the preceding command is shown below:

```
1a
2 based
1 between
1 changes
1 determining
2 distance
1 edit
1 endings
1for
1 inner
1 lemmatization
. . . .
```

As you can see, the preceding output is an alphabetical listing of the tokens of the contents of the text file `words.txt`, along with the frequency of each token.

HANDLING STOP WORDS

Stop words are words that are considered unimportant in a sentence. Although the omission of such words would result in grammatically incorrect sentences, the meaning of such sentences would most likely still be recognizable.

In English, stop words include the words "a," "an," and "the," along with common words and prepositions ("inside," "outside," and so forth). Stop words are usually filtered from search queries because they would return a vast amount of unnecessary information. As you will see later, Python libraries such as NLTK provide a list of built-in stop words, and you can supplement that list of words with your own list.

Removing stop words works fine with BoW and tf-idf, both of which are discussed in the next chapter. A more detailed explanation (and an example) is here:

https://towardsdatascience.com/why-you-should-avoid-removing-stop-words-aa7a353d2a52

A universal list of stop words does not exist, and different toolkits (such as NLTK and gensim) have different sets of stop words. The Sklearn library provides a list of stop words that consists of basic words ("and," "the," and "her"). However, a list of stop words for the text in a marketing-related website is probably different from such a list for a technical web site. Fortunately, Sklearn enables you to specify a custom list of stop words via the hyperparameter `stop_words`.

Finally, the following link contains a list of stop words for an impressive number of languages:

https://github.com/Alir3z4/stop-words

WHAT IS STEMMING?

Stemming refers to reducing words to their root or base unit, which involves truncating word suffixes. A stemmer operates on individual words without any context for those words. For example, "fast" is the stem for the words fast, faster, and fastest. Stemming algorithms are typically rule-based and involve conditional logic. In general, stemming is simpler than lemmatization (discussed later), and it's a special case of normalization.

Singular vs. Plural Word Endings

The manner in which the plural of a word is formed varies among languages. In many cases, the letter "s" "or es" is the plural form of words in English. In some cases, English words have a singular form that ends in s/us/x (basis/, abacus, and box, respectively), as well as irregular plural forms, such as "cactus/cacti" and "appendix/appendices". German can form the plural of a noun with "er" and "en," such as "buch/bucher" and "frau/frauen".

Common Stemmers

The following list contains several commonly used stemmers in NLP:

- Porter stemmer (English)
- Lancaster stemmer
- SnowballStemmers (more than 10 languages)
- ISRIStemmer (Arabic)
- RSLPSStemmer (Portuguese)

The Porter stemmer was developed in the 1980s, and while it's good in a research environment, it's not recommended for production. The Snowball stemmer is based on the Porter2 stemming algorithm, and it's an improved version of Porter (about 5% better).

The Lancaster stemmer is a good stemming algorithm, and you can even add custom rules to the Lancaster stemmer in NLTK (but results can be odd). The other stemmers in the preceding list support non-English languages.

As a simple example, the following code snippet illustrates how to define two stemmers using the NLTK library:

```
import nltk
from nltk.stem import PorterStemmer, SnowballStemmer

porter = PorterStemmer()
porter.stem("Corriendo")

snowball = SnowballStemmer("spanish", ignore_stopwords=True)
snowball.stem("Corriendo")
```

Notice that the second stemmer defined in the preceding code block also ignores the stop words.

Stemmers and Word Prefixes

Word prefixes can pose interesting challenges. For example, the prefix "un" often means "not" (such as the word unknown), but not in the case of "university." One approach for handling this type of situation involves creating a word list and after removing a prefix, check if the remaining word is in the list: if not, then the prefix in the original word is not a negative. Among the few (only?) stemmers that provides prefix stemming in NLTK are Arabic stemmers:

- *https://github.com/nltk/nltk/blob/develop/nltk/stem/arlstem.py#L115*
- *https://github.com/nltk/nltk/blob/develop/nltk/stem/snowball.py#L372*

However, it's possible to write custom Python code to remove prefixes. First, navigate to this URL to see a list of prefixes in the English language:

- *https://dictionary.cambridge.org/grammar/british-grammar/word-formation/prefixes*
- *https://stackoverflow.com/questions/62035756/how-to-find-the-prefix-of-a-word-for-nlp*

A Python code sample that implements a basic prefix finder is here:

https://stackoverflow.com/questions/52140526/python-nltk-stemmers-never-remove-prefixes

Over Stemming and Under Stemming

Over stemming occurs when too much of a word is truncated, which can result in unrelated words having the same stem. For example, consider the following sequence of words:

university, universities, universal, universe

The stem for the four preceding words is "univers," even though these words have different meanings.

Under stemming is the opposite of over stemming: this happens when a word is insufficiently "trimmed." For example, the words "data" and "datu" both have the stem "dat," but what about the word "date?" This simple example illustrates that it's difficult to create good stemming algorithms.

WHAT IS LEMMATIZATION?

Lemmatization determines whether words have the same root, which involves the removal of inflectional endings of words. Lemmatization involves the WordNet database during the process of finding the root word of each word in a corpus.

Lemmatization finds the base form of a word, such as the base word "good" for the three words "good," "better," and "best." Lemmatization determines the dictionary form of words and therefore requires knowledge of parts of speech. In general, creating a lemmatizer is more difficult than a heuristic stemmer. The NLTK lemmatizer is based on the WordNet database.

Lemmatization is also relevant for verb tenses. For instance, the words "run," "runs," "running," and "ran" are variants of the verb run. Another example of lemmatization involves irregular verbs, such as "to be" and "to have" in romance languages. Thus, the collection of verbs "is," "was," "were," and "be" are all variants of the verb "be." There is a trade-off: lemmatization can produce better results than stemming at the cost of being more computationally expensive.

Stemming/Lemmatization Caveats

In case you need to review (or learn) the terms recall and precision, the following link contains useful details:

https://en.wikipedia.org/wiki/Precision_and_recall

In the context of this section, stemming and lemmatization are designed for "recall," whereas "precision" tends to suffer. Moreover, results can also differ significantly in non-English languages, even those that seem related to English, because the implementation details of some concepts are quite different.

Although both techniques generate the root form of inflected words, you probably noticed that the stem might not be an actual word, whereas the lemma *is* an actual language word. As a rule of thumb: use stemming if you are primarily interested in higher speed, and use lemmatization if you are primarily interested in higher accuracy.

Limitations of Stemming and Lemmatization

Although stemming and lemmatization are suitable for Indo-European languages, these techniques are not as well-suited for Chinese because a Chinese character can be a combination of two other characters, all three of which can have different meanings.

For example, the character for mother is the combination of the radical for "female" and the radical for "horse." Hence, separating the two radicals for "mother" via stemming and lemmatization change the meaning of the word from "mother" to "female." More detailed information regarding Chinese natural language processing is available here:

https://towardsdatascience.com/chinese-natural-language-pre-processing-an-introduction-995d16c2705f

WORKING WITH TEXT: POS

The acronym POS refers to parts of speech, which involves identifying the parts of speech for words in a sentence. The purpose of POS tagging is to assign a part of speech to the words in a document. However, words can be assigned multiple speech tags: for example, drive can be a noun as well as a verb. The challenge of POS is to determine the correct tag for each word.

One approach for sequence labeling involves the HMM (Hidden Markov Model), which is based a probabilistic sequence model that is based on a Markov chain. One component of HMMs involves *transition probabilities*, which is the probability that a tag will follow a given tag. These probabilities can be calculated from the set of bigrams of a given corpus.

Another component of HMMs is called the *emission probabilities*, which involves the probability that a given tag will be "associated" with a given tag. HMMs make several other assumptions, and also leverage the Viterbi algorithm (not discussed in this book) in order to perform a so-called "decoding" task. For more information about HMMs and the Viterbi algorithm, perform an online search for relevant articles.

POS Tagging

POS are the grammatical function of the words in a sentence. Consider the following simple English sentence:

The sun gives warmth to the Earth.

In the preceding example, "Sun" is the subject, "gives" is the verb, "warmth" is the direct object, and "Earth" is the indirect object. In addition, the subject, direct object, and direct object are also nouns. Note that the following sentence has the same meaning, but this time the indirect noun must be inferred:

The sun gives the Earth warmth.

Words with multiple meanings are *overloaded*, and the function of a given word depends on the context. Here are three examples of using the word "bank" in three different contexts:

He went to the bank.
He sat on the river bank.
He can't bank on that outcome.

POS tagging refers to assigning a grammatical tag to the words in a corpus, and it is useful for developing lemmatizers. POS tags are used during the creation of parse trees and to define NERs (discussed in the next section). Chapter 6 contains a Python code sample that uses NLTK to perform POS tagging on a corpus (which is just a sentence, but you can easily extend it to an entire document).

POS Tagging Techniques

The major POS tagging techniques (followed by brief descriptions) are listed below:

- Lexical Based Methods
- Rule-Based Methods
- Probabilistic Methods
- Deep Learning Methods

Lexical Based Methods assign POS tags based on the most frequently occurring words in a given corpus. By contrast, Rule-Based Methods use grammar-based rules to assign POS tags. For example, words that end in the letter “s” are the plural form (which is not always true). Note that this rule applies to English and Spanish words. German words that end in the letter “e” are often plural forms (but they can be the feminine form of a word as well). Italian words ending in “i” or “e” are often the plural form of words (but many feminine words also have an “e” ending).

Probabilistic Methods assign POS tags based on the probability of the occurrence of a particular tag sequence. Finally, Deep Learning Methods use deep learning architectures (such as RNNs) for POS tagging.

WORKING WITH TEXT: NER

NER is an acronym for named entity recognition, which is known by various names, including named entity identification, entity chunking, and entity extraction. NER is a subtask of information extraction, and its purpose is to find named entities in a corpus and then classify those named entities based on predefined entity categories. As a result, NER can assist in transforming unstructured data to structured data.

In high level terms, a “named entity” is a real-world object that is assigned a name, which can be a word or a phrase that distinguishes one “item” from other items in a corpus. There are various pre-defined named entity types, such as PERSON (people, including fictional), ORG (Companies, agencies, institutions), and GPE (Countries, cities, states). Other entity types include Ethnicity, Name, Occupation, Quantity, Type, and Unit. A complete list of named entity types is here:

https://spacy.io/api/annotation

The extraction of meaningful information is a challenging task, partially due to ambiguity, especially in unstructured data. NER has benefited from machine learning, such as the kNN algorithm and CRF (Conditional Random Field). More recently, NER formed the basis for text extraction in the Transformer architecture, which has yielded significant advances in NLP.

Although NER is very useful, there are situations in which NER can produce incorrect results, such as

- insufficient number of tokens
- too many tokens
- incorrectly partitioning adjacent entities
- assigning an incorrect type

Later in this book you will see Python code samples from NLP toolkits, such as NLTK, that provide support for NER.

Abbreviations and Acronyms

As a reminder, an acronym consists of the first letter of several words, such as NLP (Natural Language Processing), whereas an abbreviation is a shortened form of a word, such as "prof" for professor. Depending on the domain, a corpus can contain many acronyms or abbreviations (or both).

Detection of abbreviations is a task of sentence segmentation and tokenization processes, which includes disambiguating sentence endings from punctuation attached to abbreviations. This task is domain-dependent and of varying complexity (and higher complexity for the medical field).

The following link contains information about CARD (clinical abbreviation recognition and disambiguation) that recognizes abbreviations in a corpus:

https://academic.oup.com/jamia/article/24/e1/e79/2631496

In addition, you can customize the tokenizer in spaCy (discussed later) by adding extra rules, as described here: *https://spacy.io/usage/linguistic-features.*

Furthermore, the PUNKT system was been developed for sentence boundary detection, and it can also detect abbreviations with high accuracy.

Chunking refers to the process of extracting phrases from unstructured text. For example, instead of treating "Empire State Building" as three unrelated words, they are treated as a single chunk.

NER Techniques

Currently NER techniques can be classified into four general categories, as shown below:

- Rule-based
- Feature-based supervised learning
- Unsupervised learning
- Deep learning

Rule-based techniques rely on manually specified rules, which means that they do not require annotated data. Unsupervised learning techniques do not require labeled data, whereas supervised learning techniques involve feature engineering. Various supervised machine learning algorithms for NER are available, such as the Hidden Markov Model (HMM), Decision Trees, Maximum Entropy Model, Support Vector Machine (SVM), and Conditional Random Field (CRF).

Finally, deep learning techniques automatically discover classification from the input data. However, deep learning techniques require a significant amount of annotated data, which might not be readily available. In addition, NER involves some complex tasks, such as detecting nested entities, multi-type entities, and unknown entities.

WHAT IS TOPIC MODELING?

Topic modeling is a technique for determining topics that exist in a document or a set of documents, which is useful for providing a synopsis of articles and documents. Topic modeling involves unsupervised learning (such as clustering), so the set of possible topics are unknown. The topics are defined during the process of generating topic models. Topic modeling is generally not mutually-exclusive because the same document can have its probability distribution spread across many topics.

In addition, there are hierarchical topic modeling methods for handling topics that contain multiple topics. Moreover, topics can change over time; they may emerge, later disappear, and then re-emerge as topics.

There are several algorithms available for topic modeling, some of which are listed below:

- LDA (Latent Dirichlet Allocation)
- LSA (Latent Semantic Analysis)
- Correlated Topic Modeling

Latent Dirichlet Allocation (LDA) is a well-known unsupervised algorithm for topic modeling. In high level terms, LDA determines the word tokens in a document and extracts topics from those tokens. LDA is a non-deterministic algorithm that produces different topics each time the algorithm is invoked.

By way of analogy, LDA resembles the well-known kMeans algorithm: LDA requires that you specify a value for the number of topics, just as kMeans requires a value for the number of clusters. Next, LDA calculates the probability that each word belongs to its assigned "topic" (cluster), and does so iteratively until the algorithm converges to a stable solution (i.e., words are no longer re-assigned to different topics).

After the clustering-related task is completed, LDA examines each document and determines which topics can be associated with that document. kMeans and LDA differ in one important respect: kMeans has a one-to-one

relationship between an item and a cluster, whereas LDA supports a one-to-many relationship whereby a document can be associated with multiple topics. The latter case makes intuitive sense: the longer the document, the greater the possibility that that document contains multiple topics. Moreover, LDA computes an associated probability that a document is associated with multiple topics. For example, LDA might determine that a document has three different topics, with probabilities 60%, 30%, and 10% for those three topics.

KEYWORD EXTRACTION, SENTIMENT ANALYSIS, AND TEXT SUMMARIZATION

Keyword extraction is an NLP process whereby the most significant and frequent words of a document are extracted. There are various techniques for performing keyword extraction, such as computing tf-idf values of words in a corpus (discussed in Chapter 4) and BERT models (discussed in Chapter 7). Other algorithms include TextRank, TopicRank, and KeyBERT, all of which are discussed in this article:

https://towardsdatascience.com/keyword-extraction-python-tf-idf-textrank-topicrank-yake-bert-7405d51cd839

Incidentally, NER (described in a previous section) relies on key word extraction as a step toward assigning a name to real-world objects. If you generalize even further, you can think of NER as a special case of relation extraction in NLU.

Sentiment analysis determines the sentiment of a document, which can be positive, neutral, or negative and often represented by the numbers 1, 0, and –1, respectively. Sentiment analysis is actually a subset of text summarization. Sentiment analysis can be implemented using supervised or unsupervised techniques, in a number of algorithms, including Naive Bayes, gradient boosting, and random forests.

Text summarization is just what the term implies: given a document, summarize its contents. Text summarization is a two-phase process that involves various techniques, including keyword extraction and topic modeling. The first phase creates a summary of the most important parts of a document, followed by the creation of a second summary that represents a summary of the document.

There are various text summarization algorithms, such as *LexRank* and *TextRank*. The LexRank algorithm uses a ranking model (based on similarity of sentences) in order to categorize the sentences in a document: sentences with a higher similarity have a higher ranking.

TextRank is an extractive and unsupervised technique that determines words embeddings for the sentences in a corpus, calculates and stores sentence similarities in a similarity matrix, and then converts the matrix to a graph. A summary is based on the top-ranked sentences in the graph.

SUMMARY

This chapter started with a high-level overview of human languages, how they might have evolved, and the major language groups. Next you learned about grammatical details that differentiate various languages from each other that highlight the complexity of generating native-level syntax as well as native-level pronunciation.

In addition, you got a brief introduction to NLP applications, NLP use cases, NLU, and NLG. Then you learn about concepts such as word sense disambiguation, text normalization, tokenization, stemming, lemmatization, and the removal of stop words. Finally, you learned about POS (Parts of Speech) and NER (Named Entity Recognition) and various algorithms that perform topic modeling in NLP.

CHAPTER 5

NLP CONCEPTS (II)

This is the second chapter that discusses NLP concepts, such as word relevance, vectorization, basic NLP algorithms, language models, and word embeddings. The next chapter contains R-based code samples that illustrate many of the concepts that are discussed in this chapter and the previous chapter.

The first part of this chapter discusses word relevance, text similarity, and text encoding techniques. The second part of this chapter discusses text encoding techniques and the notion of word encodings. The third part of this chapter introduces you to word embeddings, which are highly useful in NLP. In addition, you will learn about vector space models, n-grams, and skip-grams.

The final section discusses word relevance and dimensionality reduction techniques, some of which are based on advanced mathematical concepts. As such, these algorithms are covered in a high-level fashion, and you can perform an Internet search to find more detailed explanations of these algorithms. Alternatively, if you are not interested in the more theoretical underpinnings of machine learning algorithms, you can skim through this section of the chapter and return to this material when you need to learn more about the details of dimensionality reduction algorithms.

WHAT IS WORD RELEVANCE?

If you are wondering what it means to say that a word is "relevant," there is no precise definition. The underlying idea is that the relevance of a word in a document is related (proportional) to how much information that word provides in a document (and the latter is also imprecise). Stated differently, words have higher relevance if they enable us to gain a better understanding of the contents of a document without reading the entire document.

If a word rarely occurs in a document, that suggests that the word could have higher relevance. However, if a word occurs frequently, then the relevance of the word is generally (but not always) lower. For example, if the word "unicorn" has a limited number of occurrences in a document, then it has higher word relevance, whereas stop words such as "a," "the," and "or" have very low word relevance. Another scenario involves word relevance in multiple documents: suppose we have 100 documents, and the word "unicorn" appears frequently in a single document but not in the other 99 documents. Once again, the word "unicorn" probably has significant relevance.

Another factor in the relevance of a word is related to the number of synonyms that exist for a given word. The words "unicorn" and "death" do not have direct synonyms (though the latter does have euphemisms), which means that in some cases the words will appear more frequently in a document, and yet they still have higher word relevance than stop words.

In addition to determining the words that are relevant in a document or a corpus, we might also want to know whether or not two text strings (such as sentences or documents) are similar, which is the topic of the next section.

WHAT IS TEXT SIMILARITY?

Text similarity calculates the extent to which a pair of text strings (such as documents) are similar to each other. However, two text strings can be similar yet they can have different meanings.

For example, the two sentences "The man sees the dog" and "The dog sees the man" contain identical words (and also have the same word relevance), yet they differ in their meaning because English is word-order dependent. Replace "sees" with "bites" in the preceding pair of sentences to convey a more vivid contrast in meaning. Clearly, we need to take into account the context of the words in the two sentences, and not just the set of words.

Note that German is not word order dependent, so the words in a sentence can be rearranged without losing the original meaning. German is among the languages (such as Lithuanian and Slavic languages) that supports declension of articles and adjectives. An example of two identical German sentences is shown below, and notice that the word order is reversed in the second sentence:

Der Mann sieht **den** Hund.
Den Hund sieht der Mann.

The "den" particle represents either the direct object case for masculine singular words or the indirect object case for plural words. Hence, the "Hund" (dog) is the direct object on both of the preceding sentences.

One approach for handling the word order dependency aspect of languages such as English involves creating floating point vectors for words. Then we can calculate the cosine similarity of two vectors, and if the value is close to 1, we infer that the words associated with the vectors are closely related. This

technique is called *word vectorization*, and it's the topic of a section later in this chapter, after the section that discusses the meaning of text encoding.

SENTENCE SIMILARITY

There are various algorithms for calculating sentence similarity, such as the Jaccard Similarity, word2vec with the cosine similarity (the latter is discussed in this chapter), LDA with the Jenson-Shannon distance, and a universal sentence encoder.

One class of algorithms involves the cosine similarity, and another class of algorithms involves deep learning architectures, such as Transformer, LSTMs, and VAEs (and the latter two are beyond the scope of this book). You might be surprised to discover that you can even use the kMeans clustering algorithm in machine learning to perform sentence similarity. Another technique is the Universal Sentence Encoder, as discussed in the next section.

Sentence Encoders

Pre-trained sentence encoders for sentences are the counterpart of word2vec and GloVe (both are discussed later in this chapter) for words. The embeddings are useful for various tasks, including text classification. Sentence encoders can capture additional semantic information when they are trained on supervised and unsupervised data. Models that encode words in context are also called *sentence embedding models*.

In particular, Google created the Universal Sentence Encoder that encodes text into high dimensional vectors that can be used for various natural language tasks, and the pre-trained model is available at the TensorFlow Hub (TFH):

https://tfhub.dev/google/collections/universal-sentence-encoder

One variant of this model was trained with the Transformer encoder, which has a higher accuracy, and another variant was trained with Deep Averaging Network (DAN), which has a lower accuracy. In fact, there are 11 models available at the TFH that have been trained to perform different tasks.

WORKING WITH DOCUMENTS

Two tasks pertaining to documents involve document classification (for determining the nature of a document) and document similarity (i.e., comparing documents), both of which are discussed in the following subsections.

Document Classification

Document classification can be performed with different levels of granularity, from document-level down to sub-sentence level of granularity. The specific level that you choose depends on your task-specific requirements.

Document classification can be performed in several ways in machine learning. One way to do so involves well-known algorithms such as the SVM (Support Vector Machines) and Naive Bayes.

Document Similarity (doc2vec)

There are several algorithms for determining document similarity, including Jaccard (not discussed), doc2vec, and BERT (discussed in Chapter 7).

The doc2vec algorithm an unsupervised algorithm that converts documents into a corresponding vector and then computes their cosine similarity. The doc2vec algorithm learns fixed-length feature embeddings from variable-length pieces of texts. Despite its name, doc2vec works on sentences and paragraphs as well as documents. Details about the doc2vec algorithm are in the original paper:

https://arxiv.org/abs/1405.4053

The choice of algorithm for document similarity depends on the criteria that are used to judge document similarity, such as

- Tag Overlap
- Section
- Subsections
- Story Style
- Theme

The following article evaluates several algorithms for document similarity that takes into account the items in the preceding bullet list:

https://towardsdatascience.com/the-best-document-similarity-algorithm-in-2020-a-beginners-guide-a01b9ef8cf05

The following link contains an example of using the doc2vec algorithm:

https://medium.com/@japneet121/document- vectorization-301b06a041

TECHNIQUES FOR TEXT SIMILARITY

In general, a set of documents with the same theme typically contain words that are common throughout those documents. In some cases, a pair of documents might contain only generic words, and yet the documents share the same theme. For example, suppose one document only discusses tigers and another document only discusses lions. Although these two documents discuss a different animal, both documents pertain to wild animals, which clearly shows that they belong to the same theme.

There is an indirect connection between the documents that discuss tigers and lions: they are both "instances" of the higher-level (and more generic) topic called "wild animals." However, tf-idf values for these two documents will not determine that the documents are similar: doing so involves a distributed representation (such as doc2vec) for the word embeddings of the words in the two documents.

The following article performs a comparison of different algorithms for calculating document similarity:

https://towardsdatascience.com/the-best-document-similarity-algorithm-in-2020-a-beginners-guide-a01b9ef8cf05

The preceding article compares the accuracy of tf-idf, Jaccard, USE, and BERT (discussed in Chapter 7) on a set of documents to determine document similarity. Interestingly, tf-idf is the fastest algorithm (by far) of the four algorithms, and in some cases, tf-idf out-performed the other three algorithms in terms of accuracy.

Similarity Queries

Suppose that we have a corpus consisting of a set of text documents. A *similarity query* determines which of those documents is the most similar to a given query. Here is a very high-level sequence of steps in the algorithm:

1. Index every document in the corpus.
2. Find the distance between the query and each document.
3. Select the documents with the lowest distance values.

The distance between a query and a document can be computed in several ways, and one of the most popular techniques is called the cosine similarity, which is explained in more detail later in this chapter. However, the key idea involves calculating the (mathematical) cosine of the angle between the two vectors, which is between –1 and 1 inclusive. When this floating number is close to 1, the angle between the vectors is close to 0, which in turn suggests that the words associated with the two vectors are probably close in meaning. If the angle between the vectors is close to 1, then the angle between the vectors is close to 90 degrees, which in turn suggests that the words associated with the two vectors are probably unrelated. A value of –1 suggests that the two words might have opposite meanings (antonyms).

WHAT IS TEXT ENCODING?

Many online articles use the terms text encoding and text vectorization interchangeably to indicate a vector of numeric values. However, this chapter distinguishes between vectors whose values are calculated by training a neural network (word vectorization) versus vectors whose values are calculated directly (text encoding).

The purpose of this distinction is assist in understanding the differences (as well as similarities) among various vectorization documents (i.e., it's not to be pedantic). In simple terms, this distinction is not an industry standard.

Based on the distinction between text encoding and text vectorization, the following algorithms are text encodings:

- BoW
- N-grams
- Tf-idf

The algorithms in the preceding list have a simple intuition; however, they do not capture the context of words, nor do they track the grammatical aspects

(such as subject, verb, noun) of the words in a document. Note that BoW and n-grams generate word vectors that have integer values, whereas tf-idf generates floating point numbers. Moreover, these three techniques can result in sparse vectors when the vocabulary is large.

TEXT ENCODING TECHNIQUES

There are three well-known techniques for text encoding (three of which involve integer-valued vector), as listed below:

1. Document Vectorization
2. One-Hot Encoding
3. Index-Based Encoding

The following subsections provide a summary of each of the preceding text encoding techniques. In the next chapter, you will see code samples that illustrate these techniques. Another technique involves word embeddings, but since this technique involves more complexity than those in the preceding bullet list, word embeddings are discussed later. (If you would prefer not to wait: *word embeddings* are calculated by training a shallow neural network or by means of a technique called matrix factorization.)

Document Vectorization

Document vectorization creates a dictionary of unique words in the document and each word becomes a column in the vector space. Each text becomes a vector of 0s and 1s, where 1 indicates the presence of a word and 0 indicates the absence of a word. This is called a *one-hot* document vectorization. Although this does not preserve word order in input text, it's easy to interpret and generate.

As an illustration, the following technique performs document vectorization by performing the following steps:

```
Find the # of unique words in the corpus (let's call this M)
count the occurrences of each unique word in each document
for i = 1 to N (= number of documents):
    for document i create a 1xM vector W
       for j = 1 to M:
            W[j] = 1 if word j is in document i
```

For example, suppose we have the following 3 documents (N=3):

Doc1: Steve loves deep dish Chicago pizza.
Doc2: Dave also loves Chicago pizza.
Doc3: Both like Guinness.

The list of unique words (M=11) in the preceding three documents is shown here:

```
{also, both, Chicago, Dave, deep, dish, Guinness, like, loves, pizza, Steve}
```

A text encoding for `Doc1`, `Doc2`, and `Doc3` consists of 1×11 vectors containing integer values, as shown here:

```
Doc1: [0,0,1,0,1,1,0,0,1,1,1]
Doc2: [1,0,1,1,0,0,0,0,1,1,0]
Doc3: [0,1,0,0,0,0,1,1,0,0,0]
```

While document vectorization works reasonably well for a limited number of unique words, it's less efficient for a large number of unique words because the text encoding of sentences will tend to have many occurrences of 0, which is called *sparse data*. In this example, there are 11 unique words, but consider what happens when there are several hundred unique words contained in multiple sentences: each sentence is (generally) much shorter than the list of unique words, and therefore the corresponding vector contains mostly 0s.

The preceding technique populates vectors with 0 and 1 values. However, there is a *frequency-based vectorization* that uses the frequency of each word in the document instead of just its presence or absence. This is accomplished by modifying the innermost loop in the preceding code with the following code snippet:

```
W[j] = # of occurrences of word j in document i
```

One-Hot Encoding (OHE)

A OHE is a compromise between preserving the word order in the sequence and the easy interpretability of the result. Each word in a vocabulary is represented as a vector with a single 1 and the remaining values of the vector are all 0. For example, if you have a vocabulary of 10 words, then each row in a 10×10 identity matrix is a one-hot encoding that can be associated with one of the ten words in the vocabulary. In general, each row of an `nxn` identity matrix can represent a categorical variable that has `n` distinct values. Unfortunately, this technique can result in a very sparse and very large input tensor. Chapter 6 contains a code sample that illustrates a one-hot encoding of a vocabulary.

A OHE relies on a BoW representation of the words in a vocabulary. A OHE assumes that words are independent, which means that synonyms are represented by different vectors. The size of each vector equals the number of words in the vocabulary. Thus, a vocabulary of 100 words is encoded as 100 vectors, each of which as 100 elements (99 of them are 0 and one of them is 1).

As a simple example, the sentence "I love thick pizza" can be tokenized as ["I," "love," "thick," and "pizza"] and one-hot encoded as follows:

```
[1,0,0,0]
[0,1,0,0]
[0,0,1,0]
[0,0,0,1]
```

The sentence "We also love thick pizza" can be encoded as follows:

```
[0,1,1,1] = [0,1,0,0] + [0,0,1,0] + [0,0,0,1] = [0,1,1,1]
```

The left-side vector `[0,1,1,1]` is the component-based sum of the three vectors that represent the one-hot encoding of the words `love`, `thick`, and `pizza`, respectively.

There are two points to notice about this encoding. First, the first index of this vector is 0 because this sentence contains "we" instead of "i." Second, the words "we" and "also" are not part of the vocabulary: they are OOV (out of vocabulary) words.

One algorithm that can handle OOV words is fastText (developed by Facebook), which is discussed later in this chapter. Another approach involves a model that is based on bi-LSTMs (bidirectional LSTMs), as described here:

https://medium.com/@shabeelkandi/handling-out-of-vocabulary-words-in-natural-language-processing-based-on-context-4bbba16214d5

The key idea in the preceding link involves determining the most likely embedding for OOV words.

Another article regarding OOV words involves the skip-gram model that is discussed later in this chapter, but it's included here in case you are already familiar with this model (alternatively, you can wait until after we discuss the skip-gram model):

https://towardsdatascience.com/creating-word-embeddings-for-out-of-vocabulary-oov-words-such-as-singlish-3fe33083d466

Index-Based Encoding

This technique tries to address input data size reduction as well as the sequence order preservation. Index-based encoding maps each word to an integer index and groups the index sequence into a collection type column. Here is the sequence of steps (in high level terms):

- Create a dictionary of words from the corpus.
- Map words in the dictionary to indexes.
- Represent a document by replacing its words with indexes.

Although this technique supports variable-length documents, note that this technique creates an artificial (and misleading) distance between documents.

Additional Encoders

Although the previous sections discussed just three-word encoders, there are many other encoding techniques available, some of which are in the following list:

- BaseEncoder
- BinaryEncoder
- CatBoostEncoder

- CountEncoder
- HashingEncoder
- LeaveOneOutEncoder
- MEstimateEncoder
- OrdinalEncoder
- SumEncoder
- TargetEncoder

We will not cover these word encoders, but information regarding the text encoders (along with with Python code snippets) in the preceding list is available online:

https://towardsdatascience.com/beyond-one-hot-17-ways-of-transforming-categorical-features-into-numeric-features-57f54f199ea4

THE BOW ALGORITHM

Based on a dictionary of unique words that appear in a document, the BoW algorithm generates an array with the number of occurrences in the document of each dictionary word. The advantages of BoW include simplicity and an easy way to see the frequency of each word in a document. BoW is essentially an n-gram model with n = 1 (n-grams are discussed later in this chapter).

However, BoW does not maintain any word order and no form of context, and in the case of multiple documents, BoW does not take into account the length of the documents.

As a simple example, suppose that we have a dictionary consisting of the words in the sentence "This is a short sentence." Then the corresponding 1×5 vector for the dictionary is (this, is, a, short, sentence). Hence, the sentence "This sentence" is encoded as the vector (1, 0, 0, 0, 1). As you can see, this (and any other) sentence is treated as a "bag of words" in which word order is lost. In general, a dictionary consists of a list of N distinct words, and any sentence consisting of words from that vocabulary is mapped to a $1 \times N$ vector of zeroes and positive integers that indicate the number of times that words appear in a sentence.

The Sklearn library provides a `CountVectorizer` class that implements the BoW algorithm. The `CountVectorizer` class tokenizes the words in a corpus and generates a numeric vector that contains the word counts (frequency) of each word in the corpus. Moreover, this class can also remove stop words and examine the most popular N unigrams, bigrams, and trigrams.

However, words inside `CountVectorizer` are assigned an index value instead of storing words as strings. Here is the set of parameters (and their default values) for the `CountVectorizer` class, which are explained in more detail in the Sklearn documentation page for this class:

```
class sklearn.feature_extraction.text.CountVectorizer(*,
input='content', encoding='utf-8', decode_error='strict',
strip_accents=None, lowercase=True, preprocessor=None,
tokenizer=None, stop_words=None, token_pattern='(?u)\b\w\
```

```
w+\b', ngram_range=(1, 1), analyzer='word', max_df=1.0,
min_df=1, max_features=None, vocabulary=None, binary=False,
dtype=<class 'numpy.int64'>)
```

As another example, suppose that we have the following set of sentences:

1. I love Chicago deep dish pizza.
2. New York style pizza is also good.
3. San Francisco pizza can be very good.

The set of BoW word/index pairs is as follows:

```
{'love': 9, 'chicago': 3, 'deep': 4, 'dish': 5, 'pizza':
11, 'new': 10, 'york': 15, 'style': 13, 'is': 8, 'also': 0,
'good': 7, 'san': 12, 'francisco': 6, 'can': 2, 'be': 1,
'very': 14}
```

The BoW encoding for the initial three sentences is as follows:

```
I love Chicago deep dish pizza:
[[0 0 0 1 1 1 0 0 0 1 0 1 0 0 0 0]]
New York style pizza is also good:
[[1 0 0 0 0 0 0 1 1 0 1 1 0 1 0 1]]
San Francisco pizza can be very good:
[[0 1 1 0 0 0 1 1 0 0 0 1 1 0 1 0]]
```

As you have probably deduced, BoW models lose useful information, such as the semantics, structure, sequence and context around nearby words in each text document.

WHAT ARE N-GRAMS?

An n-gram is a technique for creating a vocabulary from N adjacent words together, hence it retains some word positions. The value of N specifies the size of the group. In many cases n-grams are from a text or speech corpus when items are words, n-grams may be called *shingles*. One common use for n-grams is to supply them to the word2vec algorithm, which in turn calculates vectors of floating-point numbers that represent words.

In highly simplified terms, the key idea of n-grams involves determining a context word that is missing from a sequence of words. For example, suppose we have five consecutive words in which the third word is missing. This is called a "bi-gram" because we have two words on the left side and two words on the right side of the missing word.

There are two types of n-grams: word n-grams and character n-grams. Word n-grams include all of the following:

- 1-gram or unigram when N=1
- a bigram or a word pair when N=2
- a trigram when N=3

The preceding list also applies to character-based n-grams. In addition, the items in n-grams can be any of the following: phonemes, syllables, letters, words/base pairs according to the application. Here are examples of 2-grams and 3-grams.

Example #1: "This is a sentence" has the following 2-grams (bi-grams):
(this, is), (is, a), (a, sentence)

Example #2: "This is a sentence" has the following 3-grams (tri-grams):
(this, is, a), (is, a, sentence)

Example #3: "The cat sat on the mat" has the following 3-grams:
"The cat sat"
"cat sat on"
"sat on the"
"on the mat"

As yet another example, with the corresponding code deferred until a later chapter, suppose that we have the following set of sentences:

I love Chicago deep dish pizza.
New York style pizza is also good.
San Francisco pizza can be very good.

The bigram pairs are as follows:

```
{'love chicago': 8, 'chicago deep': 3, 'deep dish': 4,
'dish pizza': 5, 'new york': 9, 'york style': 15, 'style
pizza': 13, 'pizza is': 11, 'is also': 7, 'also good': 0,
'san francisco': 12, 'francisco pizza': 6, 'pizza can': 10,
'can be': 2, 'be very': 1, 'very good': 14}
```

The n-gram encoding for the initial three sentences is as follows:

```
I love Chicago deep dish pizza:
[[0 0 0 1 1 1 0 0 1 0 0 0 0 0 0 0]]
New York style pizza is also good:
[[1 0 0 0 0 0 0 1 0 1 0 1 0 1 0 1]]
San Francisco pizza can be very good:
[[0 1 1 0 0 0 1 0 0 0 1 0 1 0 1 0]]
```

Compare the bigram encoding of the same three sentences using a BoW encoding in an earlier section.

Calculating Probabilities with n-grams

As a simple illustration, consider the following collection of sentences, which we'll use to calculate some probabilities:

1. 'the mouse ate the cheese'
2. 'the horse ate the hay'
3. 'the mouse saw the horse'
4. 'the mouse scared the horse'

The word "mouse" appears in three sentences, and it's followed by the word "ate" (once) and the word "scared" (once). We can calculate the associated probabilities of which of "ate" and "scared" will follow the word "mouse" as follows:

```
Number of occurrences of "mouse ate" = 1
Number of occurrences of "mouse" = 3
probability of "ate" following "mouse" = 1/3
```

In a similar fashion, we have the following values pertaining to the word "scared:"

```
Number of occurrences of "mouse scared" = 1
Number of occurrences of "mouse" = 3
probability of "scared" following "mouse" = 1/3
```

As a result, if we have the sequence of words "mouse ___," we can predict that the missing word is "ate" with a probability of 1/3, and it's "scared" with a probability of 1/3.

As another illustration, consider the following modification of the previous collection of sentences, which we'll also use to calculate some probabilities:

1. 'the big mouse ate the cheese'
2. 'the big mouse ate the hay'
3. 'the big mouse saw the horse'
4. 'the mouse scared the horse'

The word "mouse" appears in three sentences, and it's followed by the word "ate" (twice), the word "saw" (once), and the word "scared" (once). We can calculate the associated probabilities of which of "ate," "saw," and "scared" will follow the word "mouse" as follows:

```
Number of occurrences of "mouse ate" = 2
Number of occurrences of "mouse" = 4
probability of "ate" following "mouse" = 2/4
```

In a similar fashion, we have the following values pertaining to the word "saw:"

```
Number of occurrences of "mouse saw" = 1
Number of occurrences of "mouse" = 4
Hence the probability of "saw" following "mouse" = 1/4
```

Finally, we have the following values pertaining to the word "scared:"

```
Number of occurrences of "mouse scared" = 1
Number of occurrences of "mouse" = 4
probability of "scared" following "mouse" = 1/4
```

As a result, if we have the sequence of words "mouse ___," we can predict that the missing word is "ate" with a probability of 2/4, it's "saw" with a probability of 1/4, and it's "scared" with a probability of 1/4.

You can also calculate the probabilities of the word that follows the pair of words "big mouse ___:" the probability that the third word is "ate" is 2/3 and the probability that the third word is "saw" is 1/3.

Although these examples are simple (and hardly practical), they illustrate the intuition of n-grams. When we look at n-grams for realistic sentences in a corpus that contains millions of words, the probabilities (and therefore the predictive accuracy) increase dramatically.

Now let's explore the details of tf (term frequency) and idf (inverse document frequency), after which we can look at the tf-idf algorithm in more detail.

CALCULATING TF, IDF, AND TF-IDF

The following subsections discuss the numeric quantities tf, idf, and tf-idf (which equals the arithmetic product of tf and idf). As you will see, tf-idf provides a more accurate assessment of word relevance in a document than using just tf or idf.

The tf-idf algorithm is an improvement over BoW because tf-idf takes into account the number of occurrences of a given word in each document as well as the number of documents that contain that word. As a result, tf-idf indicates the relative importance of a specific word in a set of documents. In fact, the Sklearn package provides the class `TfidfVectorizer` that computes tf-idf values, as you will see later in a code sample.

What is Term Frequency (TF)?

The *term frequency* of a word equals the number of times that a word appears in a document. If you have a set of documents, and a word that appears in several of those documents, then its term frequency can be different in different documents. For example, consider the two documents `Doc1` and `Doc2`:

```
Doc1 = "This is a short sentence" (5 words)
Doc2 = "yet another short sentence" (4 words)
```

We can easily calculate the term frequencies for the words "is" and "short" in `Doc1` and `Doc2`, as shown here:

```
tf(is) = 1/5 for doc1
tf(is) = 0 for doc2
tf(short) = 1/5 for doc1
tf(short) = 1/4 for doc2
```

The following example shows you how to use term frequency to calculate numeric vectors associated with three documents to determine which pair of documents are more closely related.

Let's suppose that `doc1`, `doc2`, and `doc3` contain the words "cuisine," "pizza," "steak," "shrimp," and "caviar" with the following frequencies:

```
         doc1  doc2  doc3
-------------------------
beer    |  10 | 50 |   20
pizza   |  30 | 50 |   30
steak   |  50 |  0 |   50
shrimp  |  10 |  0 |    0
caviar  |   0 |  0 |    0
-------------------------
```

Let's normalize the column vectors in the preceding table, which gives us the following table of values:

```
         doc1  doc2  doc3
-------------------------
beer    | .10 |.50 |  .20
pizza   | .30 |.50 |  .30
steak   | .50 |  0 |  .50
shrimp  | .10 |  0 |    0
caviar  |   0 |  0 |    0
-------------------------
```

For simplicity, let's use an asterisk (`*`) to denote inner product of each pair of columns vectors, which means that we have the following values:

```
doc1*doc2 = (.10)*(.50)+(.30)*(.50)+0+0+0          = 0.20
doc1*doc3 = (.10)*(.20)+(.30)*(.30)+(.50)*(.50)+0+0 = 0.36
doc2*doc3 = (.50)*(.20)+(.30)*(.30)+0+0+0          = 0.19
```

Hence, the documents `doc1` and `doc3` are most closely related, followed by the pair `doc1` and `doc2`, and then the pair `doc2` and `doc3`.

The next section discusses inverse document frequency, followed by tf-idf, which we could use instead of the tf values to determine which pair of documents in the preceding example are most closely related.

What is Inverse Document Frequency (IDF)?

The following example illustrates how to calculate the idf value for the words in a set of documents. Given a set of N documents (ex: N = 10):

1. for each word in each document:
2. set dc = # of documents containing that word
3. set idf = log(N/dc)

Let's consider the following example with N = 2 and `Doc1` and `Doc2` defined as shown here:

```
Doc1 = "This is a short sentence"
Doc2 = "yet another short sentence"
```

Then the `idf` values for "is" and "short" for the documents `Doc1` and `Doc2` are shown below:

```
idf("is") = log(2/1) = log(2)
idf("short") = log(2/2) = 0.
```

What is tf-idf?

The tf-idf value of a word in a corpus is the product of its tf value and its idf value. The tf-idf values are a measure of word relevance (not frequency). Recall that tf (term frequency) measures the number of times that words appear in a given document, so a high frequency word indicates a topic in a document, and has a higher tf.

However, the idf (inverse-document frequency) of a word is inversely proportional to the log of the number of occurrences of a word in multiple documents. Thus, a word that appears in many documents makes that word less valuable, and hence lowers its idf value. By contrast, rare words are more relevant than popular ones, so they help to extract relevance. The tf-idf relevance of each word is a normalized data format that also adds up to 1.

Notice that the idf value involves the logarithm of N/dc: this is because word frequencies are distributed exponentially, and the logarithm provides a better weighting of a word's overall popularity. In addition, tf-idf assumes a document is a "bag of words."

Note the following idf and tf-idf values:

- idf = 0 for words that appear in every document
- tfidf = 0 for words that appear in every document
- idf = log(N) for words that appear in one document

In addition, a word that appears frequently in a *single* document will have a higher tf-idf value. Moreover, a word that appears frequently in a document is probably part of a topic.

For example, suppose that the word "syzygy" appears in a collection of documents. The word "syzygy" can be a differentiator because it probably appears in a low number of documents of that collection.

After the tf-idf values are computed for the words in the corpus, the words are sorted in decreasing order, based on their tf-idf value, and then the highest scoring words are selected. The number of selected words depends on you: it can be as small as 5 or as large as 100 (or even larger).

By way of comparison, BoW and tf-idf differ from word embeddings (discussed later in this chapter) in two important ways:

1. BoW and tf-idf calculate one number per word whereas word embeddings create one vector per word
2. BoW and tf-idf work better for classifying entire documents, whereas word embeddings are useful for determining the context of words in a document

Incidentally, you can implement a rudimentary search algorithm based on tf-idf scores for the words in a corpus, and make a determination based on the most relevant words (which is based on their tf-idf value) in a corpus. Even Google Search has used tf-idf assist in determining the top-ranked links to return to users.

As another example, with the corresponding code in a later chapter, suppose that we have the following set of sentences:

I love Chicago deep dish pizza.
New York style pizza is also good.
San Francisco pizza can be very good.

The tf-idf pairs are as follows:

```
{'love': 5, 'chicago': 0, 'deep': 1, 'dish': 2, 'pizza': 7,
'new': 6, 'york': 10, 'style': 9, 'good': 4, 'san': 8,
'francisco': 3}
```

The tf-idf encoding for the initial three sentences is as follows:

```
I love Chicago deep dish pizza:
[[0.47952794 0.47952794 0.47952794 0.         0.         0.47952794
0.         0.28321692 0.         0.         0.        ]]

New York style pizza is also good:
[[0.         0.         0.         0.         0.38376993 0.
0.50461134 0.29803159 0.         0.50461134 0.50461134]]

San Francisco pizza can be very good:
[[0.         0.         0.         0.5844829  0.44451431 0.  0.
0.34520502 0.5844829  0.         0.        ]]
```

Compare the tf-idf encoding of the same three sentences using a BoW encoding and an n-gram encoding in an earlier section.

Limitations of tf-idf

The tf-idf value is useful for determining the most relevant words in a set of documents, but can be less effective when trying to match a phrase in one or more documents. If you allow partial matches, then the set of matching phrases can contain phrases that are less relevant.

For example, suppose a set of documents pertains to various animals, and you want to find the documents that contain the phrase "strong racing horse." Would you accept the phrase "strong racing dog" as a match? If this phrase has the same tf-idf value as the original search phrase, then tf-idf cannot distinguish between them, and so tf-idf cannot reject the latter phrase in the matching set of documents.

A better solution involves word2vec (or even better, an attention-based mechanism such as the transformer architecture) because word2vec provides word vectors that contain contextual information about words (which is not the case for tf-idf values). In Chapter 7, you will learn about a technique that is even more powerful than word2vec, which involves the *attention mechanism* that is part of the foundation for the Transformer-based architecture.

BoW models lose useful information, such as the semantics, structure, sequence and context around nearby words in each text document. A better approach involves statistical language models, as discussed later in this chapter.

What is BM25?

The bm25 algorithm is a modification of the term-frequency of words, which involves the following formula:

```
bm25 = tf/(tf+k), where k = an integer-valued hyper parameter
```

The bm25 value can be adjusted by specifying different values for k: in all cases, the maximum bm25 value is 1.

Another adjustment to consider is the length of a document: a word that occurs once in a short document will have a higher TF value than a word that appears once in a long document. One way to take the document length into account is to replace `k` by the adjusted term `k*doc_len/avg_doc_len`, where:

```
dl = document length
avg_doc_len = the average length of the documents
```

This adjusted term is smaller for shorter documents than for average length or longer documents, so a single word that appears in documents will be weighted accordingly.

You can also replace `k` with `[1 - b + b*doc_len/avg_doc_len]`, where `b` is a floating point value between 0 and 1. The preceding expression approaches the quantity `k*doc_len/avg_doc_len` (i.e., the term in the preceding paragraph) as `b` approaches 1, and the expression approaches 1 as `b` approaches 0.

Furthermore, we can replace the expression `log(N/df)` for the `idf` value with the expression `log((N - df + 0.5)/(df + 0.5)`, which is a special case of the expression `log(N-df)/df`.

Note that the preceding expression is *negative* for terms that are in more than half of the corpus. Hence, we can take the preceding fact into account by using the following expression for the `idf`:

`idf = log(1 + (N - df + 0.5))/(df + 0.5)` which is approximately equal to the expression `log(N/df)`.

Pointwise Mutual Information (PMI)

PMI is an alternative to tf-idf, which works well for both word–context matrices as well as term–document matrices. However, PMI is biased toward infrequent events.

A better alternative to PMI is a variant known as Positive PMI (PPMI) that replaces negative PMI values with zero (which is conceptually similar to ReLU in machine learning). Some empirical results indicate that PPMI has superior performance when measuring semantic similarity with word-context matrices.

THE CONTEXT OF WORDS IN A DOCUMENT

There are two types of context for words: semantic context and pragmatic context, both of which are discussed in the following subsections. You will also learn about the distributional hypothesis regarding the context of words. The distributional hypothesis is based on something called a *heuristic*, which means that it is based on an assumption that is often true. In fact, the assumption is true to that extent that its accuracy is reliable enough that it outweighs the frequency of its incorrect estimates.

In a subsequent section, you will also learn about the cosine similarity metric that is used to measure the distance between two floating point vectors that represents two words.

What is Semantic Context?

Semantic context refers to the manner in which words are related to each other. For example, if you hear a sentence that starts with "Once in a blue ____," you might infer that the missing word is "moon." Another example is "I'm feeling fine and ___," where the missing word is "dandy."

The *distributional hypothesis* asserts that words that occur in a similar context tend to have similar meanings. The *context* of a word is the set of words that commonly occur around that word. For example, in the sentence "the cat sat on the mat," here is the context of the word "sat":

```
("the", "cat", "on", "the", "mat")
```

The key idea is worth repeating here: words with similar contexts share meaning and their reduced vector representations will be similar.

Another interesting concept is *pragmatics*, which is a subfield of linguistics that studies the relationship between context and meaning. As a simple example, consider the following sentence: "He was in his prison cell talking on his new cell phone while a nurse extracted some of his blood cell samples." As you can see, the word "cell" has three different meanings in the previous sentence, and therefore any embedding that takes into account both semantic and pragmatic context must generate three different vectors. More information about pragmatics is available online:

https://en.wikipedia.org/wiki/Pragmatics

Textual Entailment

Another interesting NLP task is called *textual entailment*, which analyzes a pair of sentences to predict whether the facts in the first sentence imply the facts in the second sentence. This type of analysis is important in various NLP-based applications, and actual results do vary (as you might expect). In fact, one of the techniques for training the BERT model is called NSP, which is an acronym for Next Sentence Prediction. More details regarding NSP are in Chapter 7.

Discrete, Distributed, and Contextual Word Representations

Discrete text representations refer to techniques in which words are represented independently of each other. For example, the tf-idf value of each word in a corpus is based on its term frequency multiplied by the logarithm of its inverse document frequency. Thus, the tf-idf value of each word is unaffected by the semantics of the other words in the corpus.

Moreover, if a new document is added to a corpus, or an existing document is reduced or increased in size, then the initial tf-idf value will change for some of the words in the original corpus. However, the new value does not include any of the semantics of the newly added words.

By contrast, *distributed text representations* create representations that are based on multiple words: thus, the representations of words are not mutually exclusive. For example, distributed text representations include co-occurrence matrices, word2vec, and GloVe, and fastText. Keep in mind that word2vec involves a neural network to generate word vectors, whereas GloVe uses a matrix-oriented technique (with SVD), which is discussed in Chapter 6. In addition, word2vec and GloVe are limited to one word embedding for every word, which means that a word that's used with two or more different contexts will have the same embedding for every occurrence of that word.

Finally, contextual word representations are representations that take into account all the other words in a given sentence. Hence, if a word appears in two sentences with two different meanings (i.e., context), then the word will have two different word embeddings for the two sentences. This is the fundamental idea that underlies the statement "all you need is attention."

WHAT IS COSINE SIMILARITY?

You are probably familiar with the Euclidean distance metric for finding the distance between a pair of points in the Euclidean plane: their distance can be calculated via the Pythagorean theorem. The Euclidean distance metric can be generalized to n-dimensions by generalizing the formula for the Pythagorean theorem from two dimensions to n-dimensions.

If we represent words as numeric vectors, then it's reasonable to ask the following question: if two words have similar meanings, then how do we

compare their vector representations? One way involves calculating the difference between the two vectors. For instance, suppose we are in two dimensions (because this will simplify the example), and word `U` is a vector u with components `[u1,u2]`, and word `V` is a vector v with components `[u1,u2]`. Then the difference between these two vectors is `U-V`, which is the two-dimensional vector `[u1-v1, u2-v2]`.

However, the difference between these vectors increases significantly if we multiply each of these vectors by a positive integer. In essence, we want to treat the vectors `U` and `V` as having the same property as `3*u` and `10*v` (or some other multiples of `u` and `v`), which we cannot accomplish if we use the Euclidean metric.

One solution involves calculating the cosine of the angle between a pair of vectors, which is called the *cosine similarity* of two vectors. The cosine function is a trigonometric function of the angle between the two vectors. In brief, suppose that a right-angled triangle has sides of length `a` and `b`, a hypotenuse of length `c` (that's the slanted side, which is also the longest side), and the angle between the sides of length `a` and `c` is `theta`. Then, the cosine of the angle `theta` is defined as follows:

```
cosine(theta) = a/c
```

The preceding formula applies to values of `theta` between 0 and 90 degrees (inclusive). Since `a` and `c` are positive, then `a/c > 0`, and since `a < c`, then `a/c < 1`. In addition, the definition can be extended as follows:

```
if   0 <= theta <= 90:  cosine(theta) = a/c (as defined above)
if  90 <= theta <= 180: cosine(theta) = (-1)*cosine(180-theta)
if 180 <= theta <= 270: cosine(theta) = (-1)*cosine(270-theta)
if 270 <= theta <= 360: cosine(theta) = (+1)*cosine(360-theta)
```

The cosine of `theta` is negative when `theta` is between 90 and 180, and its range of values is between 0 and –1. Since the cosine of `theta` is between 0 and 1 when `theta` is between 0 and 90, we arrive at the following result:

```
-1 <= cosine(theta) <= 1 (for 0<= theta <= 360)
```

We can generalize further for angles that are less than 0 or greater than 360: simply add (or subtract) multiples of 360 until we get an angle between 0 and 360.

```
Cosine(-100) = cosine(-100+1*360) = cosine(260) = (-1)*cosine(10)
cosine(750) = cosine(750-2*360) = cosine(30)
```

However, two vectors always form an angle that is between 0 and 180 inclusive. Since values of the cosine function are always between –1 and 1 inclusive, the cosine similarity of two vectors is also between –1 and 1 inclusive. As a reminder, the cosine of 0 degrees is 1, the cosine of 90 degrees is 0, and the cosine of 180 degrees is –1.

The intuition of cosine similarity is that "closer" vectors have a smaller angle between them, which means that the cosine of the angle is closer to 1, and so the words have similar meanings.

Two vectors whose angle between them is close to 90 degrees have a cosine similarity that is close to 0, and so the words are less related to each other. Finally, two vectors that "point" in opposite directions will have an angle of 180 degrees, and the cosine of 180 is –1, so the words will be unrelated.

The inner product of two vectors A and B is defined as

```
A "dot" B = |A|*|B|*cosine(theta)
cosine(theta) = (A "dot" B) /( |A|*|B|)

Example: suppose that A = [1, 1] B = [2, 0]:
cosine(theta) = (1*2+1*0)/[sqrt(2)*2] = 1/sqrt(2)
In this case, theta is 45 degrees
```

Note that vectors are often *normalized*, which means that they are scaled so that their length equals 1. Scaling a vector involves dividing a vector by its magnitude (also called the *norm*), which is calculated via the Pythagorean theorem.

```
Example #1:
If A = [1,1], then |A| = sqrt(1*1+1*1) = sqrt(2), and:
A/|A| = [1/sqrt(2), 1/sqrt(2)] (about [0.707,0.707])

Example #2:
If A = [2,0], then |A| = sqrt(2*2+0*0) = sqrt(4) = 2, and:
A/|A| = [2/2, 0/2] = [1, 0]

Example #3:
If A = [3,4], then |A| = sqrt(3*3+4*4) = sqrt(25) = 5, and:
A/|A| = [3/5, 4/5]

Example #4:
If A = [-4,3], then |A| = sqrt((-4)*(-4)+3*3) = sqrt(25) =
5, and:
A/|A| = [-4/5, 3/5]
```

Although cosine similarity works well in many cases, it's not a perfect solution. For example, it's possible to have two sparse vectors representing two sentences with similar meaning, even though they have no words in common, and yet their cosine similarity could be around 0.6.

In addition to cosine similarity, there are other well-known distance metrics, some of which are discussed in one of the appendices.

TEXT VECTORIZATION (AKA WORD EMBEDDINGS)

In common parlance, *text vectorization* involves the creation of word embeddings, where each word embedding is a dense one-dimensional vector of floating point numbers. Moreover, the word embeddings are generated by means of a shallow neural network. The good news is that there are various

publicly available word embeddings available, so you don't need to be concerned about generating those vectors.

Depending on your task, you might be able to work with small context vectors for words, such as 1×16 or 1×32 vectors. By comparison, the word embeddings in the BERT model (discussed in Chapter 7) are 1×512 vectors.

Since we can add floating point vectors that have the same number of components, we can calculate the average of two or more word vectors. Hence, it's possible to represent a document as the average vector of the individual word vectors in that document. However, such a vector is not necessarily meaningful with respect to the document.

You can use word embeddings to find co-occurrences. For example, "good" and "bad" both appear in a corpus and are near each other in an embedding space, despite the fact that "good" and "bad" are antonyms.

From a different perspective, it might be helpful to think of a word embedding as a projection of the index-based encoding (or a one-hot encoding) into a numerical vector to a lower-dimension space. For example, a point P in three-dimensional Euclidean space can be represented as `(x,y,z)`, and its projection onto the x-y plane is the point `(x,y,0)`, whereas its projection onto the x-z plane is the point `(x,0,z)`.

The new space is defined by the numerical output of an embedding layer in a neural network. This results in a close mapping of words with similar role, but it does involve a higher degree of complexity.

Text vectorization is typically performed after the other tasks that are discussed in Chapter 4, such as normalization, stop word removal, and lemmatization.

As you will see later in this chapter, word2vec is one of the first text vectorization algorithms that produces word embeddings by training a shallow neural network (i.e., a single hidden layer), and every word is represented by a vector of floating point numbers. These vectors are *context vectors* because they contain contextual information for the associated words (the meaning of context will be explained later).

However, word2vec does have a significant limitation: a word in a document can only have a single context vector. Hence, the same context vector is used for a given word, regardless of whether that word has a different context in different sentences.

The Transformer architecture (discussed in Chapter 7) achieved a breakthrough by overcoming this limitation of word2vec. Thus, the context vector for a given word depends on the context of that word in a sentence, which means that the same word can be represented by different context vectors.

OVERVIEW OF WORD EMBEDDINGS AND ALGORITHMS

This section contains several subsections, starting with a description of word embeddings, followed by brief description of word embedding algorithms.

Some of these algorithms, such as CBoW and skip-grams, are discussed in more detail later in this chapter.

In addition to word embeddings, there is the concept of *entity embedding* that generalizes the concept of a word embedding: an entity can be a word, a sentence, or a document.

Word Embeddings

According to Wikipedia, word embeddings are defined as:

the collective name for a set of language modeling and feature learning techniques in natural language processing (NLP) where words or phrases from the vocabulary are mapped to vectors of real numbers.

The goal is to capture as much semantic information as possible by finding a reliable word representation with real-number vectors. Techniques such as term frequencies or one-hot encodings do not provide any context for words in a sentence or a document. On the other hand, word embeddings do provide context for words, which enables you to create more powerful language models.

A word embedding is a representation of the underlying text corpus (i.e., a collection of text-based documents). Word embeddings are a context-independent embedding or representation.

Word embeddings are useful for document classification, which involves supervised learning (i.e., labeled data). You can also use word embeddings for document clustering, which involves unsupervised learning (i.e., unlabeled data).

Word embeddings reduce large one-hot word vectors into smaller vectors while simultaneously preserving some of the meaning and context of the words. One of the most popular methods for performing this reduction is called word2vec.

Fortunately, word embeddings are useful for analyzing text data in many languages (not just English text). Moreover, there are pretrained word embeddings available, and it's worthwhile performing an analysis at those word embeddings to see if they meet your needs. If not, then you can certainly create custom word embeddings.

Word Embedding Algorithms

There are several well-known word embedding algorithms, as shown in the following list:

- word2vec
- GloVe
- Fasttext

The word2vec algorithm consists of two algorithms: CBoW (Continuous Bag of Words) and skip-grams. Both word2vec algorithms create word

embeddings (i.e., vectors of floating point numbers) by training a shallow neural network that contains a single hidden layer.

The GloVe algorithm was developed at Stanford (more details are in Chapter 6), whereas the fastText algorithm is from Facebook, with more details elsewhere in this chapter. One of the most popular Python-based libraries for word embeddings is word2vec, which is the topic of the next section.

WHAT IS WORD2VEC?

A group of Google researchers developed word2vec in 2013, and it has become the foundation of NLP that is also incorporated in BERT. Word2vec provides an efficient method to represent words as vectors in a lower-dimensional space.

Word2vec takes text-based input and generates a vector consisting of floating points for each word in a text corpus. This task involves a neural network consisting of an input layer, a hidden layer (with no activation function), and an output layer that has the same dimension as the input layer. If you have studied deep learning, then you probably recognize this neural network as an autoencoder. If need be, you can use a dimensionality reduction technique to further reduce the dimensionality of the word vectors.

One point to keep in mind is that word2vec is described as an unsupervised algorithm because there is no need to label the training data. However, the shallow network that is used to generate word embeddings involves backward error propagation, which in turn requires labeled data. More accurately, word2vec involves self-supervision, which is a subset of supervised learning.

The material presented earlier in this chapter discussed the CBoW model (which uses n-grams) and the skip-gram model, both of which are part of word2vec. Later you will learn about GloVe, which is another word2vec model.

Word2vec uses cosine similarity to measure the distance between a pair of vectors, let's call them u and v. If the cosine similarity is close to 1 (which means the angle is close to 0), then the two words that correspond to vectors u and v probably have a similar meaning. If the cosine similarity is close to 0 (which means the angle is close to 90), then the associated words are probably unrelated. Finally, if the cosine similarity is close to −1 (the angle is close to 180), the associated words are good candidates for antonyms.

Word2vec is used for making predictions rather than counting words. In particular, word2vec is designed to accomplish the following tasks:

- learn the distributed representations for words
- focus on the meaning of words
- attempt to understand meaning and semantic relationships among words
- does not require labels
- works similar to deep approaches (such as RNNs)

- is computationally more efficient
- learns quickly relative to other models

Recall that the context of a word is the set of words that occur on either side of a given word. For example, consider the following sentence:

"The quick brown fox **jumped** over the lazy dog."

The context of the word "**jumped**" in the preceding sentence is shown here:

("The", "quick", "brown", "fox", "over", "the", "lazy", "dog")

In word2vec, words with similar contexts have similar reduced vector representations. Word2vec also has a skip-gram model whose goal is to predict the context words that surround a given word. For example, suppose we start with the given word "jumped:" the skip-gram model would attempt to predict the context that is listed earlier in this section.

The context is derived through an iterative process that produces an embedding layer where the rows are vector representations of the words in a vocabulary.

In word2vec, every word in a vocabulary is represented as a vector. As a result, word2vec groups the vectors of similar words together in a vector space, and it detects similarities mathematically. Thus, word2vec creates vectors that are distributed numerical representations of word features, such as the context of individual words. In addition, word2vec does not require human intervention.

There are two well-known techniques that are part of word2vec: CBoW and skip-grams.

The Intuition for word2vec

An underlying assumption of word2vec is that the meaning of words can be inferred from their surrounding words. Suppose that two words have similar neighbors (meaning: the context in which it's used is about the same), then these words are probably quite similar in meaning or are at least related. For example, the words "shocked" and "appalled" are usually used in a similar context.

Word2vec is well-suited for sentiment analysis based on a corpus of user-based reviews (such as movies and books). This type of data is unstructured because there are almost no restrictions on the content of reviews (beyond a profanity rule). Other use cases for word2vec include:

A. genes, code, likes, playlists, social media graphs

B. other verbal or symbolic series in which patterns may be discerned

Word2vec can also be used for labeled data as well as unlabeled data. Remember that algorithms that are designed to work with supervised data tend to require a large set of examples.

The word2vec Architecture

The word2vec architecture options are skip-gram (default) or continuous bag of words. Unlike deep neural networks, the input layer and the lone hidden layer for the word2vec architecture are not connected with an activation function. The output layers have the same dimensionality as the input layer (which is essentially an autoencoder).

The training algorithm is hierarchical softmax (default) or negative sampling. The following link contains information about backward error propagation in word2vec, with details for CBoW and skip-grams:

http://www.claudiobellei.com/2018/01/06/backprop-word2vec/

Limitations of word2vec

Word2vec provides only one word embedding per word, which is to say that a word embedding can only store one vector for each word. Other limitations of `word2vec` are listed below:

- difficult to train on large datasets
- fine tuning is not possible
- training models is a domain-specific task
- trained on a shallow neural network with one hidden layer

As you will see in Chapter 7, the attention-based mechanism overcomes the deficiencies of word2vec.

THE CBOW ARCHITECTURE

Given a set of words, the CBoW model architecture starts with a set of surrounding words and then attempts to predict the target word (which is the center word). The CBoW model involves a feed forward neural network that determines word embeddings. The neural network consists of the following:

1. an input layer
2. a hidden layer (no activation function)
3. an output layer (softmax activation function)

In addition, the input layer and output layer have the same size. Hence, this neural network resembles an autoencoder, which "squashes" the input values into a smaller vector to obtain a more compact representation of the input data.

Figure 5.1 displays the CBoW architecture and Figure 5.2 in the next section displays the skip-grams architecture, both of which are shallow neural networks.

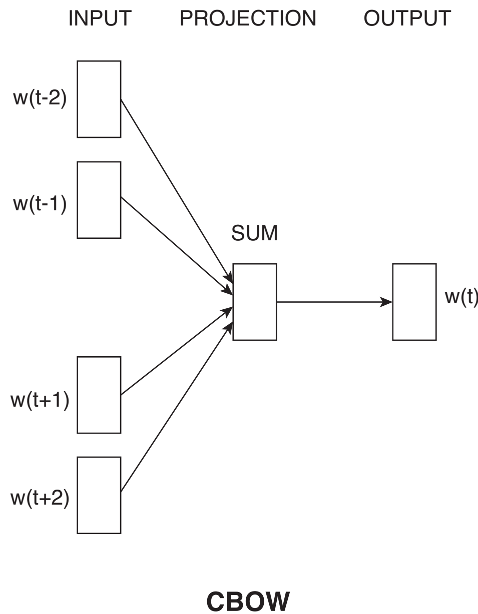

FIGURE 5.1 The CBoW architecture.
SOURCE: "Efficient Estimation of Word Representations in Vector Space."
Tomas Mikolov, Kai Chen, Greg Corrado, Jeffrey Dean. [arXiv:1301.3781v2 [cs.CL] (CC BY 4.0)].

WHAT ARE SKIP-GRAMS?

N-grams infer a missing word from the words that appear on both sides of the word, whereas skip-grams start with the "missing" word and attempt to infer the words that are most likely to appear on both sides of that missing word. In a sense, the key idea of skip-grams is sort of like an "inversion" of n-grams.

Skip-gram models predict the surrounding context words of a target word, and they are based on a neural network architecture that is discussed later in this chapter. In a sense, the skip-gram model works in the opposite manner of the CBoW model: skip-gram attempts to predict the surrounding words of a target word (which is the center word).

In slightly more detailed terms, the following sequence of steps provides a high-level description of the skip-gram algorithm:

- Treat the target word and a neighboring context word as positive examples
- Randomly sample other words in the lexicon to get negative samples
- Use logistic regression to train a classifier to distinguish those two cases
- Use the weights as the embeddings

Later you will see a diagram that displays the skip-gram architecture, right after you see an example of finding skip-grams, which is discussed in the next section.

An Example of Skip-grams

A skip-gram is a tuple that contains words before and after a given word. The size of the type is an integer, which can be as small as 1. In particular, 1-grams, 2-grams, and 3-grams are also called uni-grams, bi-grams, and tri-grams.

Let's consider the following sentence (taken from the previous section):

```
'the big mouse ate the cheese'
```

The set of 1-grams for "ate" is as follows:

```
[mouse, the]
```

The set of 2-grams is as follows:

```
[(ate,the), (ate,big), (ate,mouse), (ate, the), (ate,cheese)]
```

The set of 3-grams is as follows:

```
[(ate,the,big), (ate,big,mouse), (ate,the,cheese)]
```

The Skip-gram Architecture

Figure 5.2 displays the skip-gram architecture that is based on a shallow neural network.

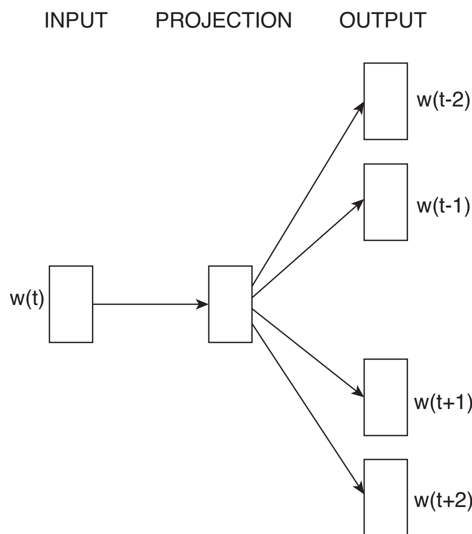

FIGURE 5.2 The skip-gram architecture.
SOURCE: "Efficient Estimation of Word Representations in Vector Space."
Tomas Mikolov, Kai Chen, Greg Corrado, Jeffrey Dean. [arXiv:1301.3781v2 [cs.CL] (CC BY 4.0)].

In order to fully understand this architecture, you need some familiarity with basic neural networks, the softmax activation function, and the concept of backward error propagation. In essence, the skip-gram architecture (along with the n-gram architecture) is based on machine learning concepts. If need be, you can read the relevant appendix that discusses neural networks.

As you can see from Figure 5.2, the skip-gram architecture consists of the following components:

1. the input layer is a single word
2. a hidden layer
3. an output layer (predicted context words)

Each word from the corpus is processed through the neural network, and after the model has been trained, the hidden layer contains the word embeddings. The concept of skip-grams is probably less intuitive than n-grams: how can we guess at the words that surround a single word?

Although the skip-gram model has a larger memory requirement, its word embeddings are better than those generated by an n-gram model.

Keep in mind the following details regarding the shallow network for the skip-gram model:

1. there is no bias term
2. there is no activation function between the input layer and the hidden layer
3. there is a softmax activation function from the hidden layer to the output layer
4. the input layer and the output layer have the same size

If you are familiar with CNNs (Convolutional Neural Networks), then you already know that the softmax activation function is applied between the rightmost hidden layer and the output layer because it generates a set of positive numbers whose sum equals one. Thus, that set of output numbers is a probability distribution, and the index position with the highest probability value is compared with the index of the number 1 in the one-hot encoding of the input data: if the index values are equal, then it's a match (otherwise it's not a match).

Since the input layer and the output layer have the same size, this shallow network is very similar to an autoencoder, whose purpose is to compress the one-hot encoded words of a vocabulary into a smaller representation (similar to the purpose of PCA in machine learning).

For example, suppose we have a vocabulary of 10,000 words (assume they're English words to keep things simple), and we want to find a representation for each word that consists of a 1×300 vector of floating point numbers. Then the weight matrix between the input layer and the hidden layer is a $10{,}000 \times 300$

matrix (let's call it W1), and the matrix between the hidden layer and the output layer is a $300 \times 10{,}000$ matrix (let's call it W2).

The neural network is *trained*, which means that the weights of the edges in the neural network are updated by a process called backward error propagation. When the training process is completed, we discard everything except for the weight matrix W1, which consists of 10,000 rows, each of which is a word in the initial vocabulary. Each row is 300 columns wide, and this 1×300 vector of floating point numbers is the encoding for the current word.

Neural Network Reduction

There are two techniques to reduce the size of the weight matrices in the neural network that is described in the previous section:

1. subsample frequent words (which decreases the number of training examples)
2. modify the optimization objective via Negative Sampling

These two techniques reduce the computational complexity and also improve the quality of the results.

The concept underlying negative sampling is to modify a small portion of the model weights, which involves finding skip-grams for a given word. An earlier section showed you how to find the bi-grams of a simple sentence, and reproduced here:

```
[(ate,the), (ate,big), (ate,mouse), (ate, the), (ate,cheese)]
```

The previous set of bi-grams includes stop words, which you can remove during the cleaning process. Alternatively, there is a formula to calculate the probability of retaining a word that appears in a vocabulary. If `w1` is a word in a vocabulary and `f(w1)` is the frequency of the word in a document, then the probability `P(w1)` that `w1` will be retained is given here:

```
P(w1) = [1 + sqrt(f(w1)*1000] * 0.001/f(w1)
```

Another important Python library for generating distributed word embeddings is GloVe, which is the topic of the next section.

WHAT IS GLOVE?

As you learned earlier in this chapter, word2vec algorithms are based on neural networks. By contrast, GloVe uses *matrix factorization* techniques from linear algebra and word-content matrices. GloVe creates a co-occurrence matrix for a given (local) context, and then decomposes the global matrix.

GloVe is similar to word2vec, with an important difference: GloVe exploits the global co-occurrences of words instead of relying on the local context. GloVe proceeds as follows:

1. construct a co-occurrence matrix of dimensionality words x context
2. factor the matrix into a matrix of dimensionality word x features

In the initial matrix, the rows are words and the columns are word frequencies in a corpus. The factored matrix has a lower dimensionality, and the rows are the vector representations of the initial words.

GloVe can provide 100-dimensional dense vectors as word embeddings. However, there are two important limitations in GloVe. First, GloVe does not support OOV (Out of Vocabulary) words. Second, GloVe does not support *polysemy*, which refers to words that have multiple meanings, and meaning is determined by the context of the words in a sentence. Consider using models that provide support, such as ELMo and USE (Universal Sentence Encoder).

CoVe (McCann, 2017) is based on the GloVe algorithm. CoVe ("contextual vectors") uses machine translation to generate contextual vectors and does not use language modeling.

WORKING WITH GLOVE

GloVe is a Python-based library for word embeddings, and it's an acronym for "Global Vectors [for word representation]". GloVe performs unsupervised learning of word embeddings that is based on co-occurrence matrices. As such, GloVe combines two techniques:

1. Global Matrix Factorization (GMF)
2. Local Context Window (LCW)

In brief, *Global Matrix Factorization* uses matrix factorization methods from linear algebra that perform rank reduction on a large term-frequency matrix. Note that the matrices can represent term-document frequencies, in which case matrix rows are words and the matrix columns are documents (or paragraphs). Alternatively, matrices can represent term-term frequencies, with words on both axes and measure co-occurrence.

GMF applied to term-document frequency matrices is called latent semantic analysis (LSA), and the high-dimensional matrix in LSA is reduced via singular value decomposition (SVD). More details regarding matrix factorization are available online:

https://machinelearningmastery.com/introduction-to-matrix-decompositions-for-machine-learning/

Local context window is a word embedding model that learns semantics by passing a window over the corpus line-by-line. This technique predicts the surroundings of a given word (e.g., skip-gram model) or predicts a word given its surroundings (e.g., CBoW).

The third important Python library for generating distributed word embeddings is fastText, which is the topic of the next section.

WHAT IS FASTTEXT?

Facebook developed the fastText NLP library, and you can install fastest with the following command:

```
pip3 install fasttext
```

The fastText library uses unsupervised learning to perform text clustering of data, which means that fastText uses a clustering algorithm. The method `train_unsupervised()` in fastText uses the skipgram model to generate 100-dimensional vectors. In addition, fastText computes the similarity score between words, along with the `get_nearest_neighbors()` method to display the top 10 words that are the most similar to a given word. Similarity scores between pairs of words that are close to 1 indicates that the pair of words are more similar in meaning.

FastText leverages word2vec by learning vector representations for each word and the n-grams in each word. Next, a vector is created whose values are the average values of the representations during each training step. This step enables word embeddings to encode sub-word information. FastText vectors are more accurate than word2vec vectors based on various criteria. Moreover, fastText can handle OOV words because it uses character n-grams; however, higher accuracy is accompanied by longer training time.

One useful advantage of vector generation techniques such as fastText is that no labeled data is required.

COMPARISON OF WORD EMBEDDINGS

This section contains a summary of the main features of three types of word embeddings. The first group consists of the simplest algorithms for producing word vectors for words: these algorithms were introduced in this chapter and the previous chapter.

The second group consists of the earliest algorithms that use neural networks (i.e., word2vec, GloVe, and fastText) or matrix factorization (such as word2vec) for generating distributional word embeddings.

The third group involves contextual algorithms for creating word embeddings, which are essentially state of the art algorithms. For your convenience, a bullet list for each of the three groups is given below:

- Group 1) Discrete word embeddings (BoW, tf, and tf-idf):
  Word vectors consist of integers, decimals, and decimals, respectively
  Key point: word embedding have zero context
- Group 2) Distributional word embeddings (word2vec, GloVe, and fasttext):
  Based on shallow NN, MF, and NN, respectively
  Two words on the left and the right (bi-grams) for word2vec
  Key point: only one embedding for each word (regardless of its context)

- Group 3) Contextual word representation (BERT et al):
  transformer architecture (no CNNs/RNNs/LSTMs)
  Pays "attention" to ALL words in a sentence
  Key point: words can have multiple embeddings (depending on the context)

The algorithms in Group #1 provide one word embedding per word but no context is captured in the word embedding. Group #2 algorithms are an improvement because they provide context for word embeddings. Finally, Group #3 algorithms generate multiple word embeddings for the same word that appears in multiple sentences. This feature is a significant improvement over Group #2 algorithms, which in turn are a significant improvement over Group #1 algorithms.

WHAT IS TOPIC MODELING?

Topic modeling is a technique for finding topics in one or more documents, and it's also a form of dimensionality reduction. There are two underlying assumptions:

1. each document consists of a mixture of topics
2. each topic consists of a collection of words

Topic models assume that the semantics of a document are governed by so-called *latent variables* that are not immediately observable, which are topics that tend to be more abstract than the actual text. The goal of topic modeling is to uncover these latent variables (topics) that can reveal the primary content of a document or corpus.

Determining the main topics in documents can be performed in various ways, which is the topic of the next section.

Topic Modeling Algorithms

There are several well-known algorithms for topic modeling, some of which are listed below:

- LDA (Latent Dirichlet Analysis)
- LSI (Latent Semantic Indexing)
- LSA (Latent Semantic Analysis)

LDA and Topic Modeling

LDA is a dimensionality reduction technique that is well-suited for topic modeling. LDA is a generative model that assigns topic distributions to documents. Each document is described by a distribution of topics, and each topic is described by a distribution of words. The rest of this section contains a high-level description of LDA, which in turn involves concepts such as KL Divergence and the JS metric, which are discussed in an appendix.

LDA starts with a fixed set of topics, where each topic represents a set of words. Next, LDA maps documents to a set of topics, and document words are mapped to those topics.

LDA is also a clustering method that supports the concept of soft-clustering, which allows different cluster to overlap (so words can belong to multiple clusters). Soft clustering is advantageous because it's simpler to find similar words; however, it's more difficult to determine distinct clusters in LDA.

Note that LDA differs from the kMeans algorithm because the latter is based on hard-clustering, which means that each word belongs to a single cluster.

An LDA model assumes that documents contain several overlapping topics, along with the following:

- topics are based on the words in each document
- the actual topics may not be known in advance
- the actual topics do not need to be specified
- the number of topics must be specified in advance

Recall that LDA supports soft clustering, and therefore the same word can appear in multiple topics (i.e., a topic has the role of a cluster). In addition, the LDA model is called "latent" because LDA generates the following latent (hidden) variables:

- a distribution over topics for each document
- a distribution over words for each topics

LDA uses the JS (Jenson-Shannon) metric, which is based on JS Divergence, and the latter is based on KL Divergence (more information about these topics is in an appendix). Since JS divergence is a metric, it's also symmetric, which means that the similarity of two documents `Doc1` and `Doc2` is the same as the similarity of `Doc2` and `Doc1` (which is obviously a desirable property).

LDA uses the JS metric to determine which documents in a corpus are the most similar to document `D` by comparing the topic distribution of document `D` to the topic distributions of the documents in the corpus. A smaller JS value for a pair of documents indicates greater similarity between the documents.

LDA is related to ANOVA as well as PCA (discussed in an appendix), but there are some differences. For instance, ANOVA uses categorical independent variables and a continuous dependent variable. By contrast, LDA involves the "reverse" of ANOVA: it uses continuous independent variables and a categorical dependent variable. LDA also assumes that the independent variables are normally distributed.

LDA and PCA share one particular aspect: both involve calculating linear combinations of variables. However, LDA tries to model the difference between the classes of data, whereas PCA ignores the difference in class.

Text Classification vs Topic Modeling

Text classification involves *supervised* learning on documents or articles with a known set of labels and also classifies text into a single class. By contrast, topic modeling involves *unsupervised* learning, and it's a process of analyzing documents/articles. Topic modeling finds groups of co-occurring words in text documents, and co-occurring related words are "topics." In cases where the set of possible topics is unknown, topic modeling can be used to solve text classification problems to identify the topics in a document.

LANGUAGE MODELS AND NLP

In brief, a language model is a probability distribution (which is explained in an appendix) for sequences of words. Statistical language modeling refers to the creation of probabilistic models that predict the next word in a sequence based on the words that precede the predicted word. Calculating the probability of word occurrences involves examples of text. Models can be based on individual words, short sequences, sentences, or paragraphs.

Language models are used in machine learning and unsupervised learning (search/IR and clustering/topic modeling). A language model also tries to distinguish between similar-sounding words. However, language models face some challenges, such as data sparsity and determining the likelihood of different phrases. One approach involves the use of n-gram models.

According to Christopher Potts [1], language models learn only from co-occurrence patterns in the streams of symbols that they are trained on. Furthermore, there are at least two issues pertaining to language models:

- Symbols streams lack crucial information
- Language models lack communicative intent

Although pure language models do not have a counterpart to machine learning models that are trained via labeled datasets, Potts is of the opinion that it's possible for language models to achieve language understanding.

How to Create a Language Model

There are three main ways to create a new language model in NLP for a given task:

- Create a new model "from scratch"
- Transfer learning (use a pre-trained model)
- Transfer learning plus vocabulary enhancement

Language models can also be classified into different subtypes. For example, neural language models (also called continuous space language models) are based on neural networks. Such models use continuous representations or

embeddings of words to make their predictions. More details regarding language models are available online:

https://en.wikipedia.org/wiki/Language_model

Language models are the foundation for vector space models, which is the topic of the next section.

VECTOR SPACE MODELS

A *vector space model* (VSM) is based on a mathematical model called a vector space, and represents text documents as vectors of identifiers (for example, using tf-idf weights). If you are unfamiliar with vector spaces, there is a brief introduction to vector spaces in one of the appendices.

A VSM consists of a two-dimensional array of (usually) numeric values that are based on *frequencies*. The latter restriction on the data values creates a "link" between a VSM and the distributional hypothesis. A VSM whose values are based on sophisticated algorithms can overcome the shortcomings of losing semantics and feature sparsity in BoWs (*https://en.wikipedia.org/wiki/Vector_space_model*).

As a point of clarification, the following matrices do *not* represent vector space models:

- an arbitrary matrix
- an adjacency matrix for a tree or graph
- a feature matrix
- a covariance matrix
- a correlation matrix
- a recommender system

Recommender systems are included in the preceding list because they populate a user-item matrix whose cells contain a numeric rating of items; however, the data in such a matrix is *not* derived from event frequencies, which explains why recommender systems are not VSMs.

Now that you have seen examples of matrices that are not VSMs, the following list contains some examples of vector space models:

- a term-document matrix (discussed later)
- a context-document matrix
- a matrix based on word2vec
- the LSA (Latent Semantic Analysis) algorithm
- a pair-pattern matrix

With the preceding in mind, here is a short list of some models that are based on (or extend) the VSM model:

- Generalized vector space model
- Latent semantic analysis (LSA)

- Term Discrimination
- Rocchio Classification
- Random Indexing

Term-Document Matrix

A *term-document* matrix M is an mxn matrix where n is the number of documents and m is the number of unique words in the n documents. The value in a cell (i, j) in a term-document matrix M equals the number of times that the term i appears in document j. Moreover, the value in a cell (i, j) can be based on other calculations, such as tf (term frequency) or tf-idf values.

Note that for a large corpus, the matrix M contains mainly zero values, which means that M is a sparse matrix (and operations are less efficient). Also keep in mind that a tf-idf vector is a vector representation of a document, whereas a word2vec vector is a vector representation of a word.

There are two more points of interest regarding a term-document matrix M. First, if two documents are similar, then the two corresponding columns in M will tend to have similar patterns of numbers, which in turn means that their cosine similarity will be closer to 1. Second, instead of focusing on column vectors, we can examine row vectors in order to measure word similarity.

We can also generalize the concept of a term-document matrix by expanding the meaning of a document to include phrases, sentences, and paragraphs. After doing so, the result is a *word-context* matrix.

Tradeoffs of the VSM

VSMs are not a perfect solution. Some of the advantages and disadvantages of a VSM are related to the advantages and disadvantages of the algorithms that are used to compute the values in the cells of a VSM.

One advantage of a VSM model is because it's based on linear algebra. In addition, it's possible to compute a degree of similarity between queries and documents in a continuous fashion, which then enables you to rank documents according to their possible relevance. Furthermore, VSM models support partial matching.

However, long documents are poorly represented because they have poor similarity values (a small scalar product and a large dimensionality). Word substrings can result in a false positive match, which means that search keywords must match the document terms. Unfortunately, documents with a similar context but contain different term vocabulary won't be associated, which results in a false negative match.

In addition, the order in which the terms appear in the document is not tracked in the vector space representation, along with the assumption that the terms are statistically independent. Even so, some of the disadvantages can be ameliorated by using techniques such as SVD (singular value decomposition).

NLP AND TEXT MINING

In high level terms, text mining performs an analysis of large amounts of unstructured data to find patterns in that data. Text mining tasks involve finding keywords, topics, and patterns. The general sequence of steps (tasks) is as follows:

- pre-processing
- text transformation
- attribute selection
- visualization
- evaluation

Text mining involves document classification whereby similar documents are placed in the same group. Text mining is useful for extracting product-related details, such as customer reviews, product issues, and so forth. Applications of text mining include spam detection, sentiment analysis, e-commerce and customer segmentation. The NLTK library is well-suited for text mining tasks, and you will see code samples in Chapter 6.

Text Extraction Preprocessing and N-Grams

N-grams are one type of language model that assigns numeric probabilities to word sequences. For example, the 3-grams of a sentence is a set of tuples of length 3, where a tuple consists of three consecutive words in that sentence. Note that the terms unigram, bigram, and trigram are often used when n is 1, 2, or 3, respectively.

RELATION EXTRACTION AND INFORMATION EXTRACTION

In simplified terms, *relation extraction* (RE), *information extraction* (IE), and *relation classification* involve various aspects of searching a corpus to find subsets of text that describe relationships between words in those subsets of text. Relation extraction is a key component of NLU, and in general, relation extraction involves extracting relational triplets of text, such as `(founder, steve_jobs, apple)`.

Although these three concepts overlap, they have significant differences. Relation extraction involves finding semantic relationships in a corpus. In addition, relation extraction is a subfield of information extraction (IE), where the latter involves extracting structured information from natural language text. However, relation extraction differs in one important respect from IE: the latter also performs disambiguation. The sense2vec algorithm is one algorithm for word sense disambiguation that can be used with SpaCy:

https://github.com/explosion/sense2vec

As an example, if you have ever summarized a text document, you probably searched for the most important words (typically nouns) and the relationship

between those words: this task is a form of IE. In fact, IE is relevant for multiple NLP tasks, including text summarization and question-answering systems.

Relation classification is the task of identifying the semantic relation holding between two nominal entities in text. There is no one-size-fits-all solution that works for multiple domains (e.g., healthcare, biology, and chemistry).

One more point of interest is the "Never Ending Language Learning" (NELL) semantic machine learning system from Carnegie Mellon University that extracts relationships from the open Web:

https://en.wikipedia.org/wiki/Never-Ending_Language_Learning

WHAT IS A BLEU SCORE?

BLEU is an acronym for "Bilingual Evaluation Understudy,", which is a well-known NLP metric. A BLEU score involves a straightforward calculation, and since a BLEU score is typically published alongside NLP models, its inclusion has become standard practice.

However, BLEU was created `in order` to measure machine translation, and it's most reliable when it's calculated on an entire corpus instead of a sentence-by-sentence calculation. Perhaps the popularity of BLEU scores resulted in a side effect in which BLEU scores are assigned to NLP tasks where other measurement tools produce more accurate results.

BLEU has some significant limitations: it does not take into account sentence structure, which can vary significantly among different languages (see the section on "case endings" in Chapter 3), nor does it take into account the meaning of sentences.

In simplified terms, BLEU scores involve precision, n-grams, and exact matches with reference sentences. BLEU checks how many n-grams in the output also appear in the reference translation. However, BLEU does not recognize synonyms, which means that pairs of sentences that use closely related yet different verbs are not considered similar in BLEU. For example, three sentences that use the verbs "drink," "imbibe," and "consume" would probably be considered equivalent, especially in casual conversation, but BLEU does not recognize them as such.

ROUGE Score: An Alternative to BLEU

In brief, a ROUGE score is a variant of BLEU that involves recall (BLEU uses *precision*) that determines the number of n-grams of the reference translation that also appear in the output (BLEU does the opposite). More information about ROUGE is available online:

https://www.aclweb.org/anthology/N03-1020/

There are also techniques that are unrelated to BLEU, such as perplexity, WER, and F1 score, all of which are discussed in an appendix. Perform an online search with the keywords "BLEU score alternatives" and you will find many articles that discuss the alternatives to BLEU.

SUMMARY

This chapter started with a brief overview of language models, text encoding techniques, and two types of word context. Then, you learned about word embeddings, which are highly useful in NLP. You also got an introduction to distance metrics, such as cosine similarity (for measuring the distance between two vectors) and document similarity.

Then you learned about word2vec, which involves CBoW and skip-grams, both of which are based on a shallow neural network. Furthermore, you learned about GloVE, which is based on matrix factorization instead of neural networks. In addition, you learned about the concepts of VSMs (vector space models) and topic modeling.

CHAPTER 6

NLP IN R

This chapter contains NLP-related code samples in R that perform NLP-related tasks that are described in previous chapters. This chapter contains code samples that involve an R "wrapper" around underlying Python code. Hence, you need to install Python for your machine.

For your convenience, the first section contains an R script that you can launch from the command line in order to install the R libraries that you need for this book. Some of the code samples also contain commented-out code snippets for installing R libraries on your machine. The code snippets contain a URL that references a repository, an example of which is shown here:

```
install.packages("NLP",repos="https://cloud.r-project.org")
```

The second section shows you how to perform data cleaning on text strings, which includes tasks such as normalization (converting text to lowercase), removing stop words, removing punctuation, and removing white spaces. You will also see a similar example in which the text string is retrieved from a plain text file.

The third section contains examples of NER (Named Entity Recognition), as well as the BoW algorithm. The fourth section contains code samples that show you how to implement the tf-idf algorithm, as well as the execution of a code sample in R that involves the word2vec algorithm.

The final section shows you how to use the NLTK and SpaCy modules in R to perform various NLP-related tasks.

Important: Some of the code samples in this chapter require you to install Python, NLTK, gensim, and spaCy your machine, which are available as free downloads on the Internet.

Although there are various errors that you might encounter while launching the R code samples, the good news is that you will most likely find an online solution for those issues. For example, one SpaCy code sample had an issue that was resolved simply by upgrading to the latest version of SpaCy.

LAUNCH R SCRIPTS FROM THE COMMAND LINE

In addition to executing R files from inside RStudio, you can also do so from the command line with the `rscript` utility. If you have a MacBook, this utility is probably in the `/usr/local/bin` directory, which you can verify by typing the following command:

```
which rscript
```

Recall that you can launch an R script (let's call it `abc.R`) from the command line as follows:

```
rscript abc.R
```

In fact, you can invoke multiple R scripts from the command line with the shell script `run_all.sh`, whose contents are displayed in Listing 6.1.

LISTING 6.1: run_all.sh

```
# launch all R scripts in the current directory:
for f in `ls *R`
do
  echo "=> Launching $f..."
  rscript $f
done

# launch all R scripts in all sub-directories:
# for f in 'find . -print| xargs grep "\.R$"'
# do
#  echo "=> Launching $f..."
#  rscript $f
# done

# launch R scripts starting with the letter "L":
# for f in `ls L*R`
# do
#   echo "=> Launching $f..."
#   rscript $f
# done
```

Listing 6.1 contains two sections: the first part executes *all* the R scripts and the second part executes all R scripts that are in the current directory and any subdirectory. The third part executes a *subset* of the R scripts in the current directory. The second and third portions of Listing 6.1 are commented out, so remove the initial #to execute those sections of the shell script.

Launch the shell script in Listing 6.1 by navigating to the directory that contains `run_all.sh` and then typing the following commands (the first command is required only once):

```
chmod +x run_all.sh
./run_all.sh
```

The output of the preceding shell script depends on the contents of the R scripts in the current directory. A sample output might look something like this:

```
=> Launching SentimentAnalysis2017.R...
=> Launching abc.R...
[1] "=> STRING:a sample STRING; MiXed CasE; NUMBERS 1234;
MORE! numbers 5678"
[1] "=> LOWERCASE: a sample string; mixed case; numbers
1234; more! numbers 5678"
[1] "\r"
[1] "=> NO PUNCTUATION: a sample string mixed case numbers
1234 more numbers 5678"
[1] "=> NO WHITE SPACE: a sample string mixed case numbers
1234 more numbers 5678"
```

You can also redirect the standard output to one file and any errors to another file, as shown here:

```
./run_all.sh 1>correct.txt 2>errors.txt
```

For example, the first portion of `correct.txt` might look something like the following:

```
=> Launching SentimentAnalysis2017.R...
=> Launching abc.R...
[1] "=> STRING:a sample STRING; MiXed CasE; NUMBERS 1234;
MORE! numbers 5678"
[1] "=> LOWERCASE: a sample string; mixed case; numbers
1234; more! numbers 5678"
[1] "\r"
```

In addition, the first portion of `errors.txt` might look something like the following:

```
=> Launching SentimentAnalysis2017.R...
Error in file(file, "rt") : cannot open the connection
Calls: read.csv -> read.table -> file
In addition: Warning message:
In file(file, "rt") :
  cannot open file 'str(apple)': No such file or directory
Execution halted
```

If you prefer, you can also launch `run_all.sh` so that it runs as background process, as shown here:

```
./run_all.sh 1>correct.txt 2>errors.txt &
```

Now let's see how to install the R libraries for the R scripts in this chapter, as discussed in the next section.

Installing RStudio Packages

An R package requires a one-time installation before you can reference the package in an R script. For example, the following code snippet installs the NLP package for R and then references the NLP package:

```
install.packages("NLP",repos="https://cloud.r-project.org")
package(NLP)
```

For your convenience, Listing 6.2 shows the content of `package_list.R` that contains an extensive list of R libraries that you will need for the R scripts in this book. Use the `rscript` command line utility to launch `library_list.R` and install the various R libraries.

LISTING 6.2: library_list.R

```
install.packages("cleanNLP",  repos="https://cloud.r-project.org")
install.packages("devtools", repos="http://cran.us.r-project.org")
install.packages("dplyr",     repos="https://cloud.r-project.org")
install.packages("formattable", repos="https://cloud.r-project.org")
install.packages("ggplot2",   repos="https://cloud.r-project.org")
install.packages("githubinstall",repos="http://cran.us.r-project.org")
install.packages("gutenberger", repos="https://cloud.r-project.org")
install.packages("hcandersenr", repos="https://cloud.r-project.org")
install.packages("janeaustenr", repos="https://cloud.r-project.org")
install.packages("lubridate", repos="https://cloud.r-project.org")
install.packages("magrittr",  repos="https://cloud.r-project.org")
install.packages("NLP",       repos="https://cloud.r-project.org")
install.packages("openNLP",   repos="https://cloud.r-project.org")
install.packages("quanteda", repos="http://cran.us.r-project.org")
install.packages("Rcpp",     repos="http://cran.us.r-project.org")
install.packages("readr",     repos="https://cloud.r-project.org")
install.packages("reshape2",  repos="https://cloud.r-project.org")
install.packages("reticulate", repos="https://cloud.r-project.org")
install.packages("rJava",    repos="http://cran.us.r-project.org")
install.packages("rpart",     repos="https://cloud.r-project.org")
install.packages("RTextTools", repos="https://cloud.r-project.org")
install.packages("scales",    repos="https://cloud.r-project.org")
install.packages("SnowballC", repos="https://cloud.r-project.org")
install.packages("spacyr",    repos="https://cloud.r-project.org")
install.packages("stringr",   repos="https://cloud.r-project.org")
install.packages("syuzhet",   repos="https://cloud.r-project.org")
install.packages("textstem", repos="http://cran.us.r-project.org")
install.packages("tidytext",  repos="https://cloud.r-project.org")
install.packages("tidyverse", repos="https://cloud.r-project.org")
install.packages("tm",        repos="https://cloud.r-project.org")
install.packages("topicmodels", repos="https://cloud.r-project.org")
install.packages("udpipe",   repos="http://cran.us.r-project.org")
install.packages("wordcloud", repos="https://cloud.r-project.org")
# now perform the following installation:
```

```
package(devtools)
install_github("jonathanbratt/RBERT")
install_github("jonathanbratt/RBERTviz")
```

Listing 6.2 contains a set of install-related commands for installing more than 30 R libraries on your machine. Navigate to the directory that contains the code in Listing 6.2 and execute the following command:

```
rscript package_list.R
```

Now that we have completed the installation-related steps, let's look at an overview of R packages that you can use for cleaning NLP data, as discussed in the next section.

NLP PACKAGES IN R

The following list of `R` packages provide support for NLP, some of which are discussed in this chapter:

- OpenNLP
- Quanteda
- Spacyr
- Stringr
- Text2vec
- Wordcloud

OpenNLP is an R interface to Apache OpenNLP that provides Java-based NLP tools. OpenNLP handles NLP tasks such as word tokenization, sentence segmentation, POS, NER, and chunking.

Quanteda is a comprehensive framework for performing quantitative text analysis in R. Quanteda enables you to work with tokens and n-grams, as well as sparse matrices of documents by features.

Spacyr is an R wrapper for the Python-based spaCy library. Spacyr provide simple access to spaCy library in a straightforward manner. Install spaCy and spacyr through the spacyr function `spacy_install()`.

Stringr provides wrappers for the string package and simplifies working with character strings in R. Stringr includes functionality for working with sequences of characters surrounded by quotation marks.

Text2vec provides an efficient framework with a concise API for text analysis and natural language processing. Some of its important features include allowing users to easily solve complex tasks, maximize efficiency per single thread, transparently scale to multiple threads on multicore machines, and use streams and iterators.

TM is a package provides a set of predefined sources, such as `DirSource` and `DataframeSource`, which handle a directory, a vector interpreting each component as a document, or data-frame-like structures (such as CSV files), and more.

Wordcloud is package that creates word clouds, which are typically used to visualize text or a corpus of documents.

COMMON TASKS FOR CLEANING NLP DATASETS

Cleaning data in datasets for real estate, the Titanic dataset, and customer churn (among others) involves the following steps:

- detecting and correcting invalid data
- imputing values for missing data
- handling outliers
- handling imbalanced datasets
- selecting the most significant features

By contrast, cleaning NLP data involves a different set of tasks, such as tokenizing data (i.e., determining word tokens), converting text to lowercase, removing punctuation, and removing extra white spaces.

Cleaning numeric data versus text-based data has little more in common than the words "cleaning data." Later in this chapter you will see simple R code samples for performing the following tasks:

- Tokenization
- Convert to Lowercase
- Remove Stop Words
- Stemming
- Lemmatization

Let's examine why the preceding tasks can be performed more easily in some languages and tend to be more complex in other language groups.

Does the Language Make a Difference?

There are various NLP toolkits available that perform the tasks in the preceding section, and results tend to be better for English and European languages. These languages have the following features:

- Specify a simple delimiter for tokens (such as a space character)
- Do not involve declension of articles and adjectives
- Distinguish between singular and plural nouns
- Contain few accent marks (or none at all)

By contrast, languages such as Japanese, Mandarin, and Cantonese tend to be more difficult in terms of cleaning text because of the following reasons:

- An optional word delimiter (Japanese)
- Multiple alphabets (Japanese)

- Declension of articles and adjectives (Slavic languages)
- Multiple tones (Mandarin and Cantonese)
- Same noun for singular and plural (Japanese)

CLEANING NLP DATA IN R

This section contains simple R scripts for performing various data cleaning tasks in NLP. Please refer to the appropriate sections in Chapter 4 if you need to review the various topics (such as tokenization) in this chapter.

Tokenization

Listing 6.3 shows the content of `tokens1.R` that tokenizes a text string.

LISTING 6.3: tokens1.R

```
package(Rcpp)
package(quanteda)

str <- "a STRING? with Mixed Case! with numbers 1234 and 5678"
print(paste0("string:",str))

remove_punct<-tokens(str,remove_punct=TRUE, remove_
symbols=TRUE,remove_numbers=TRUE)
print(paste0("=> tokens without punctuation:"))
print(paste0(remove_punct))
```

Listing 6.3 starts by referencing `Rcpp` and `quanta`, and then initializes the variable `str` as a text string and displays its contents. The next code snippet removes punctuation and then removes digits from the string. Launch the code in Listing 6.3 to see the following output:

```
Package version: 3.1.0
Unicode version: 10.0
ICU version: 61.1
Parallel computing: 8 of 8 threads used.
See https://quanteda.io for tutorials and examples.
[1] "string:a STRING? with Mixed Case! with numbers 1234 and 5678"
[1] "=> tokens without punctuation:"
[1] "a"       "STRING"  "with"    "Mixed"   "Case"    "with"    "numbers"
[8] "and"
```

Remove Punctuation in Strings

Listing 6.4 shows the content of `punctuation1.R` that removes punctuation from a text string.

LISTING 6.4: punctuation1.R

```
package(tm)
str <- "a sample STRING!; MiXed CasE; NUMBERS 1234; number 5678"
```

```
print(paste0("=> initial string:"))
print(paste0(str))

# remove punctuation:
str <- lapply(str, removePunctuation)
print(paste0("=> no punctuation:"))
print(paste0(str))
```

Listing 6.4 starts by referencing the `tm` package and then initializing the variable `str` as a string consisting of uppercase and lowercase letters, punctuation, and digits. Next, the `lapply()` function applies the R function `removePunctuation()` to the string `str` and then the `print()` statement displays the new contents of the variable `str`. Launch the code in Listing 6.4 to see the following output:

```
Loading required package: NLP
[1] "=> initial string:"
[1] "a sample STRING!; MiXed CasE; NUMBERS 1234; number 5678"
[1] "=> no punctuation:"
[1] "a sample STRING MiXed CasE NUMBERS 1234 number 5678"
```

Convert Strings to Lowercase and Uppercase

Listing 6.5 shows the content of `lower_upper_case1.R` that displays a random set of 10 stop words.

LISTING 6.5: lower_upper_case1.R

```
str <- "a STRING with Mixed Case with numbers 1234 and 5678"
print(paste0("string:",str))

# convert to lowercase:
str <- lapply(str, tolower)
print(paste0("lowercase:"))
print(paste0(str))

# convert to uppercase:
str <- lapply(str, toupper)
print(paste0("uppercase:"))
print(paste0(str))
```

Listing 6.5 initializes the variable `str` as a text string and displays its contents. Next, the `lapply()` function applies the R function `tolower()` to the string `str` to convert the contents of `str` to lowercase and then displays the result. In an analogous fashion, the next code snippet invokes the R function `toupper()` to convert the contents of `str` to uppercase letters. Launch the code in Listing 6.5 to see the following output:

```
[1] "string:a STRING with Mixed Case with numbers 1234 and 5678"
[1] "lowercase:"
```

```
[1] "a string with mixed case with numbers 1234 and 5678"
[1] "uppercase:"
[1] "A STRING WITH MIXED CASE WITH NUMBERS 1234 AND 5678"
```

Convert File Data to Lowercase and Uppercase

Listing 6.6 shows the content of `file1.txt` and Listing 6.7 shows the content of `file2.txt`, both of which are referenced in Listing 6.8.

LISTING 6.6: file1.txt

```
this IS A SAMPLE for file #1
its contents are Mixed Case
and it's not just text data
1234 and other numbers 5678
```

LISTING 6.7: file2.txt

```
this is a sample for file #2
Some Uppercase Words
MIXING ALPHANUMERIC characters
!@#$ and 5678 as well
```

Listing 6.8 shows the content of `lower_upper_case2.R` that illustrates how to read text from two text files and convert the text to lowercase and also to uppercase.

LISTING 6.8: lower_upper_case2.R

```
#Load the files:
file1 <- read.delim("file1.txt")
file2 <- read.delim("file2.txt")
text1 <- c(file1,file2)
print(paste0("=> original text1:"))
print(paste0(text1))

#convert to lowercase:
text1 <- lapply(text1, tolower)
print(paste0("=> lowercase text1:"))
print(paste0(text1))

#convert to uppercase:
text1 <- lapply(text1, toupper)
print(paste0("=> uppercase text1:"))
print(paste0(text1))
```

Listing 6.8 starts by initializing the variables `file1` and `file2` with the contents of the text files `file1.txt` and `file2.txt`, respectively. The `print()` statement displays a comment, and the `head()` statement displays 10 randomly selected stop words. Launch the code in Listing 6.8 to see the following output:

```
Loading required package: NLP
[1] "=> original text1:"
[1] "c(\"its contents are Mixed Case\", \"and it's not just
text data\", \"1234 and other numbers 5678\")"
[2] "c(\"Some Uppercase Words \", \"MIXING ALPHANUMERIC
characters\", \"!@#$ and 5678 as well\")"
[1] "=> lowercase text1:"
[1] "c(\"its contents are mixed case\", \"and it's not just
text data\", \"1234 and other numbers 5678\")"
[2] "c(\"some uppercase words \", \"mixing alphanumeric
characters\", \"!@#$ and 5678 as well\")"
[1] "=> uppercase text1:"
[1] "c(\"ITS CONTENTS ARE MIXED CASE\", \"AND IT'S NOT JUST
TEXT DATA\", \"1234 AND OTHER NUMBERS 5678\")"
[2] "c(\"SOME UPPERCASE WORDS \", \"MIXING ALPHANUMERIC
CHARACTERS\", \"!@#$ AND 5678 AS WELL\")"
```

Stop Words

Listing 6.9 shows the content of `stop_words1.R` that displays a random set of 10 stop words.

LISTING 6.9: stop_words1.R

```
package(tidytext)
package(tm)

print(paste0("=> sample of 10 stop words:",collapse=" "))
head(sample(stop_words$word, 10), 10)

str  <- c("this is a sentence and it is short")
str <- c("123", "this","is","a","sentence!?")
str2 <- removeWords(str, stopwords())

print(paste0("=> Contents of str:",collapse=" "))
print(paste0(str))

print(paste0("=> 1Contents of str2:",collapse=" "))
print(paste0(str2))

print(paste0("=> 2Contents of str2:",collapse=" "))
print(paste0(str2, collapse=""))
```

Listing 6.9 starts by referencing the `tidytext` R package that contains a set of stop words. The `print()` statement displays a comment, and the `head()` statement displays 10 randomly selected stop words. Launch the code in Listing 6.9 to see the following output:

```
[1] "=> sample of 10 stop words:"
[1] "lets"  "highest"  "downs"  "upon"   "anything"  "regarding"
[7] "z"     "hers"     "their"  "not"
[1] "=> Contents of str:"
[1] "123"    "this"     "is"      "a"       "sentence!?"
[1] "=> 1Contents of str2:"
```

```
[1] "123"    ""          ""        ""        "sentence!?"
[1] "=> 2Contents of str2:"
[1] "123sentence!?"
```

Stemming in R

Listing 6.10 shows the content of `word_stem.R` that illustrates how to perform stemming in R. If need be, you can read the appropriate section in Chapter 4 that discusses how stemming is performed on text.

LISTING 6.10: word_stem.R

```
package(tm)

# The tm package provides the stemDocument() to stem words,
# which takes in a character vector and returns a character vector,
# or takes in a PlainTextDocument and returns a PlainTextDocument.
# ex: stemDocument(running,runs,ran) returns (run,run,ran)

# a bug in StemDocument:
# https://stackoverflow.com/questions/54197636/how-is-the-correct-
use-of-stemdocument
# A workaround could be using the package quanteda:
#install.packages("quanteda",repos = "http://cran.us.r-project.org")

package(tm)

word = "running"
stemDocument(word, language = "english")

word = "image"
stemDocument(word, language = "english")

word = "poder" # Spanish for "can" or "to be able to"
stemDocument(word, language = "spanish")

word = "potere" # Italian for "can" or "to be able to"
stemDocument(word, language = "italian")
```

Listing 6.10 starts by referencing the `tm` package that can perform stemming on text. The remaining portion of Listing 6.10 invokes the `stemDocument()` method to stem various words. Launch the code in Listing 6.10 to see the following output:

```
Loading required package: NLP
[1] "run"
[1] "imag"
[1] "pod"
[1] "pot"
```

Lemmatization

Listing 6.11 shows the content of `lemmatization.R` that uses the R library textstem and the R library udpipe in order to show you three different blocks of code that perform lemmatization in R.

LISTING 6.11: lemmatization.R

```
#install.packages("textstem",repos="http://cran.us.r-project.org")
#install.packages("udpipe",repos="http://cran.us.r-project.org")
library(textstem)

# first vector of words:
vector1 <- c("eat", "ate", "eaten")
print("vector of words:")
print(vector1)
print("lemmatized vector of words:")
lemmatize_words(vector1)
cat("\n")

# second vector of words:
vector2 <- c("am", "be", "was")
print("vector of words:")
print(vector2)
print("lemmatized vector of words:")
lemmatize_words(vector2)
cat("\n")

# lemmatize a corpus:
library(udpipe)
docs <- c(doc_a = "When ignorance is bliss, 'tis folly to be wise
said the Bard",
          doc_b = "Gambarimasho means let's try our best")
anno <- udpipe(docs, "english")
anno[, c("doc_id", "sentence_id", "token", "lemma", "upos")]
```

Listing 6.11 starts by referencing two libraries, followed by a code block in which the variable `vector1` is initialized with three verb forms of the verb "eat." Next, the method `lemmatize_words()` is invoked with the variable `vector1`, which generates the correct output: three occurrences of the verb "eat."

The second code block initializes the variable `vector2` with three verb forms of the verb "be." Next, the method `lemmatize_words()` is invoked with the variable `vector2`, which generates the correct output: three occurrences of the verb "be."

The third code block initializes the variable `docs` with two sentences, and then invokes the `udpipe()` method to lemmatize each word in `docs`. The final code snippet displays a tabular output that specifies the document, sentence ID, token, and lemmatization of the token, and the POS of the token. Launch the code in Listing 6.11 to see the following output, where the correct lemmatization of the verbs is shown in bold.

```
[1] "vector of words:"
[1] "eat"   "ate"   "eaten"
[1] "lemmatized vector of words:"
[1] "eat" "eat" "eat"

[1] "vector of words:"
[1] "am"  "be"  "was"
```

```
[1] "lemmatized vector of words:"
[1] "be" "be" "be"

   doc_id sentence_id         token         lemma  upos
1   doc_a           1          When          when   ADV
2   doc_a           1     ignorance     ignorance  NOUN
3   doc_a           1            is            be   AUX
4   doc_a           1         bliss         bliss   ADJ
5   doc_a           1             ,             , PUNCT
6   doc_a           1             '             ' PUNCT
7   doc_a           1           tis            ti  NOUN
8   doc_a           1         folly         folly   ADV
9   doc_a           1            to            to  PART
10  doc_a           1            be            be   AUX
11  doc_a           1          wise          wise   ADV
12  doc_a           1          said           say  VERB
13  doc_a           1           the           the   DET
14  doc_a           1          Bard          Bard PROPN
15  doc_b           1 Gambarimasho Gambarimasho PROPN
16  doc_b           1         means          mean  VERB
17  doc_b           1           let           let  VERB
18  doc_b           1            's            's  PRON
19  doc_b           1           try           try  VERB
20  doc_b           1           our            we  PRON
21  doc_b           1          best          best   ADJ
```

Notice that the word `gambarisho`, which is the Romaji-based spelling of the Japanese verb 画MBある (which means "to try one's best") is identified as a proper noun in the preceding output.

POS (PARTS OF SPEECH) WITH SPACY IN R

POS is discussed in chapter 4, and Listing 6.12 shows the content of `spacy1.R` that illustrates how to find the parts of speech in a text string.

LISTING 6.12: spacy1.R

```
# => install spacyr with this command:
#devtools::install_github("kbenoit/spacyr", build_vignettes=FALSE)

# R wrapper for spaCy Python package to extract parts of speech:
library(spacyr)

doc1 <- c("I love Chicago deep dish pizza.")
spacy_parse(doc1, tag = TRUE, entity = FALSE, lemma = FALSE)
```

Listing 6.12 starts by referencing `spacyr`, which is a R-based "wrapper" around the Python library `spaCy`. The next code snippet initializes the variable `doc` as a text string, and then invokes the `spacy_parse()` API to parse the contents of `doc`.

In brief, the `spacy_parse()` API invokes the Python `spaCy` library to tokenize and "tag" the tokens in the variable `doc`. Launch the code in Listing 6.12 to see the following output:

```
[1] "Extract POS and tags: "
Finding a python executable with spaCy installed...
spaCy (language model: en_core_web_sm) is installed in /Library/Frameworks/
Python.framework/Versions/3.7/bin/python3
successfully initialized (spaCy Version: 3.1.3, language model: en_core_web_sm)
(python options: type = "python_executable", value = "/Library/Frameworks/
Python.framework/Versions/3.7/bin/python3")
  doc_id sentence_id token_id   token   pos tag
1  text1           1        1       I  PRON PRP
2  text1           1        2    love  VERB VBP
3  text1           1        3 Chicago PROPN NNP
4  text1           1        4    deep   ADJ  JJ
5  text1           1        5    dish  NOUN  NN
6  text1           1        6   pizza  NOUN  NN
7  text1           1        7       . PUNCT   .
[1] "Extract NER and tags: "
  doc_id sentence_id  entity entity_type
1  text1           1 Chicago         GPE
[1] "Extract noun phrases: "
  doc_id sentence_id              nounphrase
1  text1           1                       I
2  text1           1 Chicago_deep_dish_pizza
[1] "Dependency parsing: "
  doc_id sentence_id token_id   token head_token_id  dep_rel entity
1  text1          1       1        I      2    nsubj
2  text1          1       2     love      2     ROOT
3  text1          1       3  Chicago      6     nmod  GPE_B
4  text1          1       4     deep      5     amod
5  text1          1       5     dish      6 compound
6  text1          1       6    pizza      2     dobj
7  text1          1       7        .      2    punct
Python space is already attached.  If you want to switch to a different Python,
please restart R.
successfully initialized (spaCy Version: 3.1.3, language model: it_core_news_sm)
(python options: type = "python_executable", value = "/Library/Frameworks/
Python.framework/Versions/3.7/bin/python3")
[1] "Parse Italian: "
   doc_id sentence_id token_id       token   pos tag entity
1       R           1        1           R PROPN  SP
2       R           1        2           e CCONJ  CC
3       R           1        3         una   DET  RI
4       R           1        4      lingua  NOUN   S
5       R           1        5      gratis   ADV   B
6       R           1        6         per   ADP   E
7       R           1        7 programmare  VERB   V
8       R           1        8        roba  NOUN   S
9       R           1        9 scientifica   ADJ   A
10      R           1       10           . PUNCT  FS
```

POS IN R

POS is an acronym for parts of speech, which includes nouns, adjectives, verbs, direct and indirect objects, and so forth.

Listing 6.13 shows the content of `pos_tokens1.R` that illustrates how to display the parts of speech in a text string in R.

LISTING 6.13: pos_tokens1.R

```
#install.packages("rJava",repos = "http://cran.us.r-project.org")
#install.packages("NLP",repos = "http://cran.us.r-project.org")
#install.packages("openNLP",repos = "http://cran.us.r-project.org")

package(NLP)
package(openNLP)

sent1 <- paste(c("I love Chicago deep dish pizza!", "Also Pizzeria Uno!"))
str1  <- as.String(sent1)

cat("\n")
print(paste0("contents of str1:"))
str1

cat("str1 tokens:","\n")
sent_annotator <- Maxent_Sent_Token_Annotator()
word_annotator <- Maxent_Word_Token_Annotator()
anntr2 <- annotate(str1, list(sent_annotator, word_annotator))
anntr2

cat("str1 annotations:","\n")
pos_tag_annotator <- Maxent_POS_Tag_Annotator()
anntr3 <- annotate(str1, pos_tag_annotator, anntr2)
anntr3

cat("subset of tokens:","\n")
anntr3_words <- subset(anntr3, type == "word")
anntr3_words

cat("POS of tokens:","\n")
tags <- sapply(anntr3_words$features, '[[', "POS")
tags

cat("table:","\n")
table(tags)
```

Listing 6.13 starts by referencing the `NLP` and `openNLP` packages, followed by initializing the variables `sent1` and `str1` and then displaying the contents of `str1`. The next portion of Listing 6.12 shows the start and end positions of the tokens in the variable `str1`.

The next code snippet also shows the start and end positions of the tokens, along with the type of token (word versus sentence) and the POS of each token. The subsequent code snippet displays the subset of tokens that are of type word, and the final code snippet displays only the POS of the tokens. Launch the code in Listing 6.13 to see the following output:

```
[1] "contents of str1:"
I love Chicago deep dish pizza!
Also Pizzeria Uno!
str1 tokens:
 id type     start end features
  1 sentence     1  31 constituents=<<integer,7>>
  2 sentence    33  50 constituents=<<integer,3>>
```

```
  3 word          1   1
  4 word          3   6
  5 word          8  14
  6 word         16  19
  7 word         21  24
  8 word         26  30
  9 word         31  31
 10 word         33  36
 11 word         38  45
 12 word         47  50
str1 annotations:
 id type     start end features
  1 sentence     1  31 constituents=<<integer,7>>
  2 sentence    33  50 constituents=<<integer,3>>
  3 word         1   1 POS=PRP
  4 word         3   6 POS=VBP
  5 word         8  14 POS=NNP
  6 word        16  19 POS=JJ
  7 word        21  24 POS=NN
  8 word        26  30 POS=NN
  9 word        31  31 POS=.
 10 word        33  36 POS=RB
 11 word        38  45 POS=NNP
 12 word        47  50 POS=NNP
subset of tokens:
 id type start end features
  3 word     1   1 POS=PRP
  4 word     3   6 POS=VBP
  5 word     8  14 POS=NNP
  6 word    16  19 POS=JJ
  7 word    21  24 POS=NN
  8 word    26  30 POS=NN
  9 word    31  31 POS=.
 10 word    33  36 POS=RB
 11 word    38  45 POS=NNP
 12 word    47  50 POS=NNP
POS of tokens:
 [1] "PRP" "VBP" "NNP" "JJ" "NN" "NN" "." "RB" "NNP" "NNP"
table:
tags
  .   JJ   NN NNP PRP  RB VBP
  1   1    2   3   1   1   1
```

NER IN R

NER is an acronym for *Named Entity Recognition*, which was introduced in Chapter 4. Listing 6.14 shows the content of `ner_example1.R` that illustrates how to perform NER in R programs.

LISTING 6.14: ner_example1.R

```
library(magrittr)

# R wrapper for spaCy Python package to extract parts of speech:
```

```
library(spacyr)

str1 <- c("Mr Smith eats Chicago deep dish pizza!", "Also Pizzeria Uno!")
cat("\n")
print(paste0("contents of str1:"))
str1

parsed1 <- spacy_parse(str1, lemma = FALSE, entity = TRUE, nounphrase = TRUE)
entity_extract(parsed1)

entity_extract(parsed1, type = "all")

entity_consolidate(parsed1) %>%
    tail()
```

Listing 6.14 starts by referencing two R libraries, and then initializes and displays the contents of the string variable `str1`. The next code snippet invokes the API `spacy_parse()`, just as you saw in a previous code sample.

The final portion of Listing 6.14 displays three sections of output, starting with the parsed tokens of `str1`. The second and third sections of the output are the same: they display the `doc_id`, `sentence_id`, `entity`, and `entity_type` of the tokens in the first output section. Launch the code in Listing 6.14 to see the following output:

```
[1] "contents of str1:"
[1] "Mr Smith eats Chicago deep dish pizza!"
[2] "Also Pizzeria Uno!"
Finding a python executable with spaCy installed...
spaCy (language model: en_core_web_sm) is installed in /Library/Frameworks/
Python.framework/Versions/3.7/bin/python3
successfully initialized (spaCy Version: 3.1.3, language model: en_core_web_sm)
(python options: type = "python_executable", value = "/Library/Frameworks/
Python.framework/Versions/3.7/bin/python3")
  doc_id sentence_id       entity entity_type
1  text1           1        Smith      PERSON
2  text1           1      Chicago         GPE
3  text2           1 Pizzeria_Uno         ORG

  doc_id sentence_id       entity entity_type
1  text1           1        Smith      PERSON
2  text1           1      Chicago         GPE
3  text2           1 Pizzeria_Uno         ORG

   doc_id sentence_id token_id        token    pos entity_type
6   text1           1        6         dish   NOUN
7   text1           1        7        pizza   NOUN
8   text1           1        8            !  PUNCT
9   text2           1        1         Also    ADV
10  text2           1        2 Pizzeria_Uno ENTITY         ORG
11  text2           1        3            !  PUNCT
```

The following link contains more information about entities:
https://spacy.io/usage/linguistic-features#section-named-entities

THE TF-IDF ALGORITHM

In Chapter 4, you learned about the tf-idf algorithm, and this section contains a code sample. Listing 6.15 shows the content of `tfidf_sample.R` that illustrates how to perform tf-idf in R.

LISTING 6.15: tfidf_sample.R

```
library(tm)
#initialize some short documents:
doc1 <- "I love deep dish pizza."
doc2 <- "Chicago deep dish pizza."
doc3 <- "New York deep dish pizza."
doc4 <- "Good toppings and crust."
doc5 <- "Deep dish with Parmigiano cheese."

# create a document list:
doc.list <- list(doc1, doc2, doc3, doc4, doc5)
N.docs <- length(doc.list)
names(doc.list) <- paste0("doc", c(1:N.docs))
query <- "Good pizza"

# create a corpus from the documents and query:
my.docs <- VectorSource(c(doc.list, query))
my.docs$Names <- c(names(doc.list), "query")
my.corpus <- Corpus(my.docs)

#####################################
# => transform the corpus as follows:
# 1) convert to lowercase
# 2) remove stopwords
# 3) remove punctuation
# 4) remove numbers
# 5) remove multiple whitespaces
# 6) remove plural
#####################################

my.corpus2 <- tm_map(my.corpus,  tolower)
my.corpus3 <- tm_map(my.corpus2, removeWords, stopwords("english"))
my.corpus4 <- tm_map(my.corpus3, removePunctuation)
my.corpus5 <- tm_map(my.corpus4, removeNumbers)
my.corpus6 <- tm_map(my.corpus5, stripWhitespace)
my.corpus6

library(SnowballC)
my.corpus7 <- tm_map(my.corpus6, stemDocument)

# create a document/term matrix:
docTermMatrix <- DocumentTermMatrix(my.corpus7)
cat("\n")
paste("*** Document Term Matrix ***",collapse=" ")
docTermMatrix
inspect(docTermMatrix)

# perform tf-idf operation:
docTermMatrix_tfxidf <- weightTfIdf(docTermMatrix)
cat("\n")
paste("*** TF/IDF Matrix ***",collapse=" ")
docTermMatrix_tfxidf
inspect(docTermMatrix_tfxidf)
```

Listing 6.15 starts by referencing an R library and then initializing 5 variables as documents. In this code sample, a "document" is simply a text string, which makes it easier to understand the output that is displayed later in this

section. In general, though, you would replace each document with a bona fide document instead of using simple text strings.

The next portion of Listing 6.15 initializes `doc.list` as the list of five documents that are defined in the previous code section. Next, the variable `doc.docs` is defined, and eventually, the variable `my.corpus` is defined, as shown here:

```
my.corpus <- Corpus(my.docs)
```

The next sequence of code snippets performs sequential processing on `my.corpus`, as described in the comment block, such as removing stops words and punctuation.

The final portion of Listing 6.15 calculates the document-term matrix that is in the variable `docTermMatrix`, after which the `tfidf` values can be calculated on the entries of this matrix and contained in the variable `docTermMatrix_tfxidf`. Launch the code in Listing 6.13 to see the following output:

```
[1] 1
<<SimpleCorpus>>
Metadata:  corpus specific: 1, document level (indexed): 0
Content:  documents: 6

[1] "*** Document Term Matrix ***"
<<DocumentTermMatrix (documents: 6, terms: 12)>>
Non-/sparse entries: 22/50
Sparsity           : 69%
Maximal term length: 10
Weighting          : term frequency (tf)
<<DocumentTermMatrix (documents: 6, terms: 12)>>
Non-/sparse entries: 22/50
Sparsity           : 69%
Maximal term length: 10
Weighting          : term frequency (tf)
Sample             :
    Terms
Docs chicago crust deep dish good love new pizza top york
   1       0     0    1    1    0    1   0     1   0    0
   2       1     0    1    1    0    0   0     1   0    0
   3       0     0    1    1    0    0   1     1   0    1
   4       0     1    0    0    1    0   0     0   1    0
   5       0     0    1    1    0    0   0     0   0    0
   6       0     0    0    0    1    0   0     1   0    0

[1] "*** TF/IDF Matrix ***"
<<DocumentTermMatrix (documents: 6, terms: 12)>>
Non-/sparse entries: 22/50
Sparsity           : 69%
Maximal term length: 10
Weighting          : term frequency - inverse document frequency
(normalized) (tf-idf)
<<DocumentTermMatrix (documents: 6, terms: 12)>>
Non-/sparse entries: 22/50
Sparsity           : 69%
Maximal term length: 10
Weighting          : term frequency - inverse document frequency
(normalized) (tf-idf)
Sample             :
```

```
    Terms
Docs      chees   chicago     crust      deep      dish      good      love
   1 0.0000000 0.0000000 0.0000000 0.1462406 0.1462406 0.0000000 0.6462406
   2 0.0000000 0.6462406 0.0000000 0.1462406 0.1462406 0.0000000 0.0000000
   3 0.0000000 0.0000000 0.0000000 0.1169925 0.1169925 0.0000000 0.0000000
   4 0.0000000 0.0000000 0.8616542 0.0000000 0.0000000 0.5283208 0.0000000
   5 0.6462406 0.0000000 0.0000000 0.1462406 0.1462406 0.0000000 0.0000000
   6 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.7924813 0.0000000
    Terms
Docs parmigiano     pizza       top
   1  0.0000000 0.1462406 0.0000000
   2  0.0000000 0.1462406 0.0000000
   3  0.0000000 0.1169925 0.0000000
   4  0.0000000 0.0000000 0.8616542
   5  0.6462406 0.0000000 0.0000000
   6  0.0000000 0.2924813 0.0000000
```

WORKING WITH N-GRAMS

In Chapter 5, you learned about n-grams. Listing 6.16 shows the content of `ngrams1.R` that illustrates how to work with a bi-gram in R.

LISTING 6.16: ngrams1.R

```
#install.packages('janeaustenr', repos = "http://cran.us.r-project.org")
library(janeaustenr)
library(magrittr)
library(dplyr)
library(tidytext)

# n-grams are discussed in chapter 5
paste0("Bigrams:",collapse=" ")
austen_bigrams <- austen_books() %>%
  unnest_tokens(bigram, text, token = "ngrams", n = 2)
austen_bigrams

paste0("Count bigrams:",collapse=" ")
austen_bigrams %>%
  count(bigram, sort = TRUE)

library(tidyr)
bigrams_separated <- austen_bigrams %>%
  separate(bigram, c("word1", "word2"), sep = " ")

bigrams_filtered <- bigrams_separated %>%
  filter(!word1 %in% stop_words$word) %>%
  filter(!word2 %in% stop_words$word)
paste0("Filtered by street for word2:",collapse=" ")
bigrams_filtered %>%
  filter(word2 == "street") %>%
  count(book, word1, sort = TRUE)

paste0("Filtered by street for word1:",collapse=" ")
bigrams_filtered %>%
  filter(word1 == "street") %>%
  count(book, word2, sort = TRUE)
```

Listing 6.16 starts by referencing several R libraries, one of which gives us access to Jane Austen's works. The first code block populates the variable

austen_bigrams with the set of bigrams from one of her books (*Sense and Sensibility*) via the function unnest_tokens(). The second code block displays a partial list of bigrams from austen_bigrams, along with the number of occurrences of each bigram.

The third code block initializes the variable bigrams_filtered by extracting the words from austen_bigrams that are *not* stop words. The fourth code block extracts the list of words from austen_bigrams in which the second word of a bigram is the word "street." Similarly, the fifth code block extracts the list of words from austen_bigrams in which the first word of a bigram is the word "street." Launch the code in Listing 6.16 to see the following output:

```
[1] "Bigrams:"
# A tibble: 675,025 × 2
   book                bigram
   <fct>               <chr>
 1 Sense & Sensibility sense and
 2 Sense & Sensibility and sensibility
 3 Sense & Sensibility NA
 4 Sense & Sensibility by jane
 5 Sense & Sensibility jane austen
 6 Sense & Sensibility NA
 7 Sense & Sensibility NA
 8 Sense & Sensibility NA
 9 Sense & Sensibility NA
10 Sense & Sensibility NA
# … with 675,015 more rows
[1] "Count bigrams:"
# A tibble: 193,210 × 2
   bigram      n
   <chr>   <int>
 1 NA      12242
 2 of the   2853
 3 to be    2670
 4 in the   2221
 5 it was   1691
 6 i am     1485
 7 she had  1405
 8 of her   1363
 9 to the   1315
10 she was  1309
# … with 193,200 more rows
[1] "Filtered by street for word2:"
# A tibble: 33 × 3
   book                word1           n
   <fct>               <chr>       <int>
 1 Sense & Sensibility harley         16
 2 Sense & Sensibility berkeley       15
 3 Northanger Abbey    milsom         10
 4 Northanger Abbey    pulteney       10
 5 Mansfield Park      wimpole         9
 6 Pride & Prejudice   gracechurch     8
 7 Persuasion          milsom          5
 8 Sense & Sensibility bond            4
```

```
 9 Sense & Sensibility conduit          4
10 Persuasion          rivers           4
# … with 23 more rows
[1] "Filtered by street for word1:"
# A tibble: 17 × 3
   book                word2         n
   <fct>               <chr>     <int>
 1 Sense & Sensibility january       2
 2 Sense & Sensibility marianne      1
 3 Sense & Sensibility set           1
 4 Sense & Sensibility yesterday     1
 5 Pride & Prejudice   elizabeth     1
 6 Pride & Prejudice   monday        1
 7 Pride & Prejudice   sept          1
 8 Mansfield Park      door          1
 9 Mansfield Park      sir           1
10 Emma                happy         1
11 Emma                till          1
12 Northanger Abbey    door          1
13 Northanger Abbey    overtook      1
14 Northanger Abbey    reached       1
15 Northanger Abbey    walking       1
16 Persuasion          afforded      1
17 Persuasion          perfectly     1
```

TOPIC MODELING IN R

In Chapter 5, you learned about topic modeling. Listing 6.17 shows the content of `topic_modeling.R` that illustrates how to work with topic modeling in R.

LISTING 6.17: topic_modeling.R

```
#install.packages('topicmodels', repos = "http://cran.us.r-project.org")
library(topicmodels)

data("AssociatedPress")
# specify a seed value so the model output is predictable
ap_lda <- LDA(AssociatedPress, k = 2, control = list(seed = 1234))
ap_lda
library(tidytext)
paste0("Word-topic probabilities:",collapse=" ")
ap_topics <- tidy(ap_lda, matrix = "beta")
ap_topics
```

Listing 6.17 starts by referencing an R package and then initializing the variable `ap_lda` with the result of invoking the LDA API, which performs the Latent Dirichlet Analysis to determine a set of topics in the dataset `AssociatedPress`.

The next portion of Listing 6.17 determines the probabilities of the occurrence of the topics that were determined in the previous code section. Launch the code in Listing 6.17 to see the following output:

```
A LDA_VEM topic model with 2 topics.
[1] "Word-topic probabilities:"
# A tibble: 20,946 × 3
   topic term           beta
   <int> <chr>         <dbl>
 1     1 aaron      1.69e-12
 2     2 aaron      3.90e- 5
 3     1 abandon    2.65e- 5
 4     2 abandon    3.99e- 5
 5     1 abandoned  1.39e- 4
 6     2 abandoned  5.88e- 5
 7     1 abandoning 2.45e-33
 8     2 abandoning 2.34e- 5
 9     1 abbott     2.13e- 6
10     2 abbott     2.97e- 5
# … with 20,936 more rows
```

WORKING WITH WORD2VEC IN R

Chapter 5 briefly described `word2vec`, which comprises the CBoW algorithm and the skip-gram algorithm, and provides floating point context vectors for words in a vocabulary. A code sample is available online:

https://gist.github.com/primaryobjects/8038d345aae48ae48988906b0525d175

Download the code from the previous link, navigate to that subdirectory, and launch the following command:

```
rscript 1-word2vec.R
```

After some installation related output, you will see the following output on your screen:

```
trying URL 'http://mattmahoney.net/dc/text8.zip'
Content type 'application/zip' length 31344016 bytes (29.9 MB)
==================================================
downloaded 29.9 MB

Beginning tokenization to text file at temp.prep
Prepping article2.txt
Starting training using file temp.prep

Vocab size (unigrams + bigrams): 192
Words in train file: 224
Starting training using file /Users/staging/Downloads/word2vec-stuff/temp.prep
Vocab size: 4
Words in train file: 25
Filename ends with .bin, so reading in binary format
Reading a word2vec binary file of 4 rows and 200 columns
  |==================================================| 100%
       word similarity to "president"
1 president             1.000000000
2     trump            -0.001098632
3      </s>            -0.071650826
4      said            -0.092222355
Beginning tokenization to text file at temp.prep
Prepping text8
Starting training using file temp.prep
```

```
Words processed: 17000K     Vocab size: 4399K
Vocab size (unigrams + bigrams): 2419827
Words in train file: 17005431
Starting training using file /Users/staging/Downloads/word2vec-stuff/temp.prep
Vocab size: 98330
Words in train file: 15857308
Filename ends with .bin, so reading in binary format
Reading a word2vec binary file of 98330 rows and 200 columns
  |==================================================| 100%
          word similarity to "communism"
1    communism                 1.0000000
2    socialism                 0.8277460
3      marxism                 0.7757172
4      marxist                 0.7737221
5    communist                 0.7617577
6    socialist                 0.7435190
7   capitalism                 0.7417053
8    stalinism                 0.7164418
9   capitalist                 0.7134871
10 proletariat                 0.7124533
```

Additional documentation regarding word2vec in R is available here:

https://www.rdocumentation.org/packages/word2vec/versions/0.3.4/topics/word2vec

SUMMARY

This chapter showed you how to download and install RStudio, as well as how to launch R scripts from the command line via the `rscript` utility.

Then you learned to perform data cleaning on text strings, which includes tasks such as normalization (converting text to lowercase), removing stop words, removing punctuation, and removing white spaces. You will also see a similar example in which the text string is retrieved from a plain text file.

In addition, you saw an R code sample for POS and also a code sample of NER (Named Entity Recognition), as well as the BoW algorithm in R. Furthermore, you learned how to implement the tf-idf algorithm and how to use the NLTK and SpaCy modules in R to perform various NLP-related operations.

CHAPTER 7

TRANSFORMER, BERT, AND GPT

This chapter is devoted to NLP and modern architectures that support NLP-based tasks. Specifically, you will learn about the transformer architecture, the pre-trained BERT model and its variants, and features of GPT-2 and GPT-3 from OpenAI.

Please note that this chapter contains Python-based code samples. The rationale for the inclusion of Python code is simple: you can quickly find a vast set of blog posts, articles, code samples, and Github repositories regarding BERT, the Transformer architecture, and GPT-3 via a simple Internet search. Fortunately, most of the code samples are short and involve rudimentary Python constructs, which you can learn from a plethora of free online resources.

The first part of this chapter contains a brief introduction to the concept of attention, which is a powerful mechanism for generating word embeddings that contain context specific information for words in sentences. The concept of attention is a key aspect of the transformer architecture. This section also contains a summary of the distinguishing characteristics of three types of word embeddings, in which the most powerful technique is the attention-based approach.

The second part of this chapter provides an overview of the transformer architecture that was developed by Google and released in late 2017. This section also discusses the T5 (Text-To-Text Transfer Transformer) model that converts all NLP tasks into a text-to-text format.

The third part of this chapter introduces you to BERT, along with various code samples that illustrate how to invoke some of the BERT APIs. Note that this section relies on the installation of the HuggingFace transformers Python library.

The fourth part of this chapter contains a list of several BERT-based trained models, along with brief description of their functionality. Some of the models that are discussed include DistilledBERT, CamemBERT, and FlauBERT. The final part of this chapter introduces you to the GPT-based models from OpenAI, along with some of the amazing features in GPT-3.

Important: The code samples in this chapter are based on Python, which is also required for various code samples in Chapter 5.

In addition, the code samples currently require Python 3.7, which you can download from the Internet if you haven't already done so.

WHAT IS ATTENTION?

Attention is a mechanism by which contextual word embeddings are determined for words in a corpus. Unlike word2vec or gloVe, the attention mechanism takes into account *all* the words in a sentence during the process of creating a word embedding for a given word. As a result, the same word in different (and distinct) sentences will have a different word embedding in each of those sentences.

Before the attention mechanism was devised, popular architectures used RNNs, LSTMs, or bi-LSTMs. In fact, the attention mechanism was first used in conjunction with RNNs or LSTMs. However, the Google team performed some experiments involving machine translation tasks on models that relied solely on the attention mechanism and the transformer architecture, and discovered that those models achieved higher performance than models that included CNNs, RNNs, or LSTMs. This result led to the expression "attention is all you need." The seminal paper regarding the transformer architecture is available online:

https://papers.nips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

As a quick review, and before delving into details of the attention mechanism, let's look at a summary of the main types of word embeddings that we have encountered, as discussed in the next section.

Types of Word Embeddings

This section contains a summary of the main features of three types of word embeddings. The first group consists of the simplest algorithms for word embeddings, and you have already seen them in previous chapters.

The second group consists of the earliest algorithms that use neural networks (word2vec and fasttext) or matrix factorization (gloVe) for generating word embeddings.

The third group involves contextual algorithms for creating contextual word representations, which are essentially state of the art algorithms. Here is the summary:

1. Discrete word embeddings (BoW, tf, and tf-idf):
   Word vectors consist of integers, decimals, and decimals, respectively
   Key point: word embedding have zero context
2. Distributional word embeddings (word2vec, GloVe, and fasttext):
   Based on shallow NN, MF, and NN, respectively
   Two words on the left and the right (bi-grams) for word2vec
   Key point: only one embedding for each word (regardless of its context)
3. Contextual word representations (such as BERT):
   Transformer architecture (no CNNs/RNNs/LSTMs)
   Pays "attention" to ALL words in a sentence
   Key point: words can have multiple embeddings (depending on the context)

Types of Attention and Algorithms

There are several types of attention mechanisms, three of which are listed below:

1. self-attention
2. global/soft
3. local/hard

Self-attention tries to determine how words in a sentence are interconnected with each other. Multi-headed attention uses a block of multiple self-attention instead of just one self-attention. However, each head processes a different section of the embedding vector.

In addition to the preceding attention mechanisms, there are also several attention algorithms available:

- Additive
- Content-based
- Dot Product
- General
- Location-base
- Scaled Dot Product <= a transformer uses this algorithm

The formulas for attention mechanisms can be divided into two broad types: formulas that involve a dot product of vectors (and sometimes with a scaling factor), and formulas that apply a softmax function or a tanh function to products of matrices and vectors.

The transformer model uses a scaled dot-product mechanism to calculate the attention. If you want more detailed information regarding attention types, the following site contains a list of more than 20 attention types:

https://paperswithcode.com/methods/category/attention-mechanisms-1

AN OVERVIEW OF THE TRANSFORMER ARCHITECTURE

The Transformer architecture differs from other architectures in the following important ways:

- it's primarily based on an attention mechanism
- model training can be parallelized
- no CNNs/RNNs/LSTMs are required

Due to the last point in the preceding list, the encoder-decoder construction differs from a seq2seq model that often contain RNNs or LSTMs.

The Transformer architecture has two main components: an encoder and a decoder. The encoder component has six (sometimes more) concatenated encoder elements. Each encoder element has two layers, and the output of the first layer is the input for the second layer (like a miniature pipeline). The final output of the sixth (or in some cases, the twelfth) encoder component is then passed to every decoder element in the decoder component.

Similarly, the decoder component also has 6 (sometimes more) concatenated decoder elements, where the output of one element in the input for the next element. However, each decoder element consists of three sub-elements, one of which is the output from the encoder.

The overall Transformer architecture consists of an encoder component that contains six "sub" encoders, as well as a decoder component that also contains six "sub" decoders. Each of these structures, which are loosely analogous to filter elements in a CNN.

The input for the encoder is a set of word embeddings that encode the words in a sentence. The word embeddings are constructed via the attention mechanism, which means that every embedding is based on all the words in a given sentence. Hence, a word that appears in two different sentences has two different word embeddings in the two sentences. Given a sentence with n tokens, the construction of each word embedding involves the remaining (n−1) words. Therefore, the attention-based mechanism has order O(N^2), where N is the number of unique tokens in the corpus.

The actual input vector for an encoder is called a *context vector*. This is a crucial detail: by contrast, word2vec constructs a single word embedding for every word, regardless of whether a given word has a different context in different sentences.

The Transformers Library from HuggingFace

HuggingFace created a transformers library and an open-source repository to develop models based on the transformer architecture that you can access online:

https://github.com/huggingface/transformers

The library provides pre-trained models for NLU (Natural Language Understanding) and NLG (Natural Language Generation). In fact, HuggingFace provides more than 30 pre-trained models for more than

100 languages, along with operability between TensorFlow 2 and PyTorch. Furthermore, HuggingFace supports not only BERT-related models, but also GPT-2/GPT-3, XLNet, and others.

HuggingFace supports more than 30 architectures, some of which are listed here:

- BART (from Facebook)
- BERT (from Google)
- Blenderbot (from Facebook)
- CamemBERT (from Inria/Facebook/Sorbonne)
- CTRL (from Salesforce)
- DeBERTa (from Microsoft Research)
- DistilBERT (from HuggingFace)
- ELECTRA (from Google Research/Stanford University)
- FlauBERT (from CNRS)
- GPT-2 (from OpenAI)
- Longformer (from AllenAI)
- LXMERT (from UNC Chapel Hill)
- Pegasus (from Google)
- Reformer (from Google Research)
- RoBERTa (from Facebook)
- SqueezeBert
- T5 (from Google AI)
- Transformer-XL (from Google/CMU)
- XLM-RoBERTa (from Facebook AI)
- XLNet (from Google/CMU)

Check the online documentation for more information regarding these architectures.

Transformers are well-suited for various tasks, such as text generation, text summarization, and language translation. The next several sections contain several short code samples that illustrate how to use the HuggingFace transformer to perform NLP-related tasks. Specifically, you will see how to perform NER, QnA, and sentiment analysis using the HuggingFace transformer.

Transformer and NER Tasks

Listing 7.1 shows the content of `hf_transformer_ner.py` that illustrates how to perform a NER task with the HuggingFace transformer.

LISTING 7.1: hf_transformer_ner.py

```
from transformers import pipeline

nlp = pipeline('ner')
result = nlp("I am a UCSC instructor and my name is Oswald")

print("result:",result)
```

Listing 7.1 starts with an `import` statement and then initializes the variable `nlp` as an instance of the `pipeline` class, with `ner` as a parameter. Next, the variable `nlp` is invoked with a hard-coded sample sentence. The output is assigned to the variable `result`, whose contents are then displayed. Launch the code in Listing 7.1 with the following command:

```
python3 hf_transformer_ner.py
```

The preceding command will launch the version of Python that is installed on your machine. As you learned in the introduction to this chapter, HuggingFace currently supports Python 3.7, but in the future, it's likely that later versions of Python will also be supported. The preceding command will display the following output:

```
result: [{'word': 'UC', 'score': 0.9993938207626343,
'entity': 'I-ORG', 'index': 4}, {'word': '##SC', 'score':
0.9974051713943481, 'entity': 'I-ORG', 'index': 5},
{'word': 'Oswald', 'score': 0.9988114833831787, 'entity':
'I-PER', 'index': 11}]
```

Transformer and QnA Tasks

Listing 7.2 shows the content of `hf_transformer_qa.py` that illustrates how to perform a question-and-answer task with the HuggingFace transformer.

LISTING 7.2: hf_transformer_qa.py

```
from transformers import pipeline

nlp = pipeline('question-answering')

result = nlp({
  'question': "Do you know my name?",
  'context': "My name is Oswald"
})

print("result:",result)
```

Listing 7.2 starts with an `import` statement and then initializes the variable `nlp` as an instance of the `pipeline` class, with `question-answering` as a parameter. Next, the variable `nlp` is invoked with a question/context pair. The output is assigned to the variable `result`, whose contents are then displayed. Launch the code in Listing 7.2 to see the following output:

```
result: [{'word': 'UC', 'score': 0.9993938207626343,
'entity': 'I-ORG', 'index': 4}, {'word': '##SC', 'score':
0.9974051713943481, 'entity': 'I-ORG', 'index': 5},
{'word': 'Oswald', 'score': 0.9988114833831787, 'entity':
'I-PER', 'index': 11}]
```

Transformer and Sentiment Analysis Tasks

Listing 7.3 shows the content of `hf_transformer_sentiment.py` that illustrates how to perform a sentiment analysis task with the HuggingFace transformer.

LISTING 7.3: hf_transformer_sentiment.py

```
from transformers import pipeline

nlp = pipeline('sentiment-analysis')
comment = "Great news that we have pipelines in transformers"

result = nlp(comment)

print("comment:",comment)
print("sentiment:",result)
```

Listing 7.3 starts with an `import` statement and then initializes the variable `nlp` as an instance of the pipeline class, with `sentiment-analysis` as a parameter. Next, the variable comment is initialized with a test string, which is supplied to the variable `nlp`. The output is assigned to the variable `result`, whose contents are displayed. Launch the code in Listing 7.3 to see the following output:

```
comment: Great news that we have pipelines in transformers
sentiment: [{'label': 'POSITIVE', 'score': 0.9985968470573425}]
```

Transformer and Mask Filling Tasks

Listing 7.4 shows the content of `hf_transformer_mask.py` that illustrates how to perform a mask-filling task with the HuggingFace transformer.

LISTING 7.4: hf_transformer_mask.py

```
from transformers import pipeline

nlp = pipeline('fill-mask')
result = nlp("I hope that you <mask> the movie")

print("result:",result)
```

Listing 7.4 starts with an `import` statement and then initializes the variable `nlp` as an instance of the pipeline class, with `fill-mask` as a parameter. Next, the variable `nlp` is invoked with a hard-coded sample sentence. The output is assigned to the variable `result`, whose contents are then displayed. Launch the code in Listing 7.4 to see the following output:

```
result: [{'sequence': '<s>I hope that you enjoyed the
movie</s>', 'score': 0.5466918349266052, 'token': 3776,
'token_str': 'Ġenjoyed'}, {'sequence': '<s>I hope that
you enjoy the movie</s>', 'score': 0.36409610509872437,
'token': 2254, 'token_str': 'Ġenjoy'}, {'sequence':
'<s>I hope that you liked the movie</s>', 'score':
0.06604353338479996, 'token': 6640, 'token_str': 'Ġliked'},
{'sequence': '<s>I hope that you like the movie</s>',
'score': 0.008552208542823792, 'token': 101, 'token_str':
'Ġlike'}, {'sequence': '<s>I hope that you loved the
movie</s>', 'score': 0.003726127091795206, 'token': 2638,
'token_str': 'Ġloved'}]
```

This concludes the section of the chapter pertaining to the HuggingFace transformer code samples. The next section briefly discusses T5, which is another NLP model created by Google.

WHAT IS T5?

T5 is an acronym for Text-To-Text Transfer Transformer. T5 is an encoder-decoder model that converts all NLP tasks into a text-to-text format, and its downloadable code is online:

https://github.com/google-research/text-to-text-transfer-transformer

You can also install T5 by invoking the following command:

```
pip install t5[gcp]
```

T5 is pre-trained on a multi-task mixture of unsupervised and supervised tasks, and it works well on various tasks, such as translation. T5 is trained using a technique called *teacher forcing*, which means that an input sequence and a target sequence are always required for training. The input sequence is designated with `input_ids`, whereas the target sequence is designated with `output_ids` and then passed to the decoder.

Since all tasks (such as classification, question answering, and translation) involve this input/output mechanism, the same model can be used for multiple tasks.

T5 provides several useful classes when working with T5 models. For example, the class `transformers.T5Config` that enables you to specify configuration information, whose default values are similar to the T5-small architecture. Another useful class is `transformers.T5Tokenizer` that enables you to construct a T5 tokenizer.

T5 does differ from BERT in two significant ways that will become clearer after you read the BERT-related material later in this chapter:

- The inclusion of a causal decoder
- The use of pre-training tasks instead of a fill-in-the-blank task

Although you can download code samples for T5, initially it might be simpler to experiment with T5 in a Google Colaboratory notebook (make sure to select a TPU for execution):

https://tiny.cc/t5-colab

More information about T5 and details regarding the preceding T5 classes (and other classes) is available online:

https://huggingface.co/transformers/model_doc/t5.html

WHAT IS BERT?

BERT is a pre-trained model that is based on the transformer architecture developed in 2017 by Google. There are two version of BERT, called BERT Base and BERT Large. BERT Base consists of 12 layers (transformer blocks), 12 attention heads, and 110 million parameters. BERT Large is a larger pre-trained model that consists of 24 layers (transformer blocks), 16 attention heads, and 340 million parameters.

BERT can be used in conjunction with the Transformers library (discussed earlier in this chapter) that provide classes to perform various tasks, such as question answering and sequence classification.

BERT Features

BERT has a set of approximately 30,000 learned raw vectors. Moreover, just under 80% of those raw vectors correspond to "normal" words (i.e., they exist in an English dictionary). The remaining 20% are subwords that are created by WordPiece: these subwords have the form ##s or ##ed. The latter subwords are useful for detecting the past tense of a verb in a sentence. In addition, the BERT vocabulary consists of 45% uppercase and 25% lowercase terms (approximately).

How is BERT Trained?

BERT is trained by performing a pre-training step, followed by a fine-tuning step. The pre-training step involves task-specific data. For example, if you want to perform sentiment analysis using BERT, you need a corpus of labeled data that specifies whether a sentence has positive or negative sentiment. In addition, the dataset is split into a training portion and a test portion, just as you would with linear regression or classification tasks.

The fine-tuning step involves training the model on a large set of sample tasks. For example, if you want to train BERT to perform a question-answering task, then start with the pre-trained model (that was trained on sentiment analysis) and fine-tune that model by training the model on a corpus of question/answer data.

How BERT Differs from Earlier NLP Models

There are several important aspects of BERT that differentiate BERT from models that involve algorithms such as word2vec. First, BERT does not perform a stemming operation: instead, BERT performs subword tokenization via WordPiece (discussed later). Stemming discards suffixes, whereas WordPiece does *not* discard the suffixes.

Second, BERT creates contextual word embeddings whereas word2vec creates distributional word embeddings. Specifically, BERT uses all the words in a sentence in order to generate a word embedding for each word in a given sentence, *and the word embedding is specific to the sentence in which the word appears*. As a result, the same word that appears in distinct sentences will have different word embeddings, whereas word2vec uses bigrams to calculate word embeddings.

Third, BERT does *not* use cosine similarity to determine the extent to which two words are similar to each other. However, it's possible to use BERT with cosine similarities, provided that you fine-tune BERT on suitable data, such as the data and code samples in the following repository:

https://github.com/UKPLab/sentence-transformers

THE INNER WORKINGS OF BERT

BERT implements a number of interesting techniques, some of which are listed below:

- MLM (Masked Language Model)
- NSP (Next Sentence Prediction)
- Special tokens ([CLS] and [SEP])
- Language mask
- Wordpiece (subword tokenization)
- SentencePiece

What is MLM?

MLM is an acronym for *masked language model*, and it's a BERT pre-training task, during which BERT processed the contents of Wikipedia (and also the BookCorpus dataset). In this task, 15% of the words were replaced with the [MASK] token, and BERT then predicted the missing words. Note that this task was performed on "chunks" of data that were submitted to BERT.

Many words in Wikipedia involve dates, names of people, and names of locations, some of which were replaced by the [MASK] token. During the training process, BERT ascertained the missing tokens correctly.

What is NSP?

In addition to MLM, BERT uses NSP, which stands for *next sentence prediction*. NSP combines pairs of sentences in the following way:

- The second sentence is logically related to the first sentence in 50% of the pairs.
- The second sentence is *not* logically related to the first sentence in 50% of the pairs.

The purpose of NSP is to identify which pairs of sentences are correct and which pairs of sentences are incorrect.

Special Tokens

BERT uses two special tokens: `[CLS]` to indicate the start of a text string and `[SEP]` to separate sentences. For example, consider the following phrase:

Pizza with four toppings and trimmings

The `BERT` tokenization of the preceding phrase is as follows:

```
['[CLS]', 'pizza', 'with', 'four', 'topping', '##s', 'and',
'trim', '##ming', '##s', '.', '[SEP]']
```

Listing 7.5 shows the content of `bert_special_tokens.py` that illustrates how to display the special tokens in BERT.

LISTING 7.5: bert_special_tokens.py

```
import transformers
import numpy as np

# instantiate a BERT tokenizer and model:
print("creating tokenizer...")
tokenizer = transformers.BertTokenizer.from_pretrained('bert-base-
uncased', do_lower_case=True)

print("creating model...")
nlp = transformers.TFBertModel.from_pretrained('bert-base-uncased')

# hidden layer with embeddings:
text1      = "cell phone"
input_ids1 = np.array(tokenizer.encode(text1))[None,:]
embedding1 = nlp(input_ids1)

print("input_ids1:")
print(input_ids1)
print()

print("tokenizer.sep_token:   ",tokenizer.sep_token)
print("tokenizer.sep_token_id:",tokenizer.sep_token_id)
print("tokenizer.cls_token:   ",tokenizer.cls_token)
print("tokenizer.cls_token_id:",tokenizer.cls_token_id)
print("tokenizer.pad_token:   ",tokenizer.pad_token)
print("tokenizer.pad_token_id:",tokenizer.pad_token_id)
print("tokenizer.unk_token:   ",tokenizer.unk_token)
print("tokenizer.unk_token_id:",tokenizer.unk_token_id)
print()
```

Listing 7.5 starts with two `import` statements and then initializes the variable `tokenizer` as an instance from a pre-trained model. Next, the variable `nlp` is initialized as an instance of a pre-trained model.

The next portion of Listing 7.5 initializes the variable `text1` as a two-word string, followed by the variable `input-ids1` that consists of the tokens for the two words, along with two special tokens.

The final code block consists of a set of `print()` statements that display several special tokens and their `token_id` values. Launch the code in Listing 7.5 to see the following output:

```
creating tokenizer...
creating model…
input_ids1:
[[ 101 3526 3042  102]]

tokenizer.sep_token:    [SEP]
tokenizer.sep_token_id: 102
tokenizer.cls_token:    [CLS]
tokenizer.cls_token_id: 101
tokenizer.pad_token:    [PAD]
tokenizer.pad_token_id: 0
tokenizer.unk_token:    [UNK]
tokenizer.unk_token_id: 100
```

BERT Encoding: Sequence of Steps

BERT performs the following sequence of steps, all of which have been illustrated via code snippets in previous sections:

Step 1: tokenize the text

Step 2: map the tokens to their IDs

Step 3: add the special [CLS] and [SEP] tokens

As a simple example, the sentence “I got a book” has a total of six tokens (four word tokens, and the start and end tokens), along with the following indices:

```
[CLS]             101
i               1,045
got             2,288
a               1,037
book            2,338
[SEP]             101
```

Listing 7.6 shows the content of `bert_encoding_plus.py` that illustrates how to display the special tokens in BERT.

LISTING 7.6: bert_encoding_plus.py

```
import transformers
import numpy as np
```

```
# instantiate a BERT tokenizer and model:
print("creating tokenizer...")
tokenizer = transformers.BertTokenizer.from_pretrained('bert-base-
uncased', do_lower_case=True)
print("creating model...")
nlp = transformers.TFBertModel.from_pretrained('bert-base-
uncased')

text="When were you last outside? I have been inside for 2 weeks."

encoding = tokenizer.encode_plus(
   text,
   max_length=32,
   add_special_tokens=True, # Add '[CLS]' and '[SEP]'
   return_token_type_ids=False,
   pad_to_max_length=True,
   return_attention_mask=True,
   return_tensors='pt',  # Return PyTorch tensors
)

print("encoding.keys():")
print(encoding.keys())
print()

print("len(encoding['input_ids'][0]):")
print(len(encoding['input_ids'][0]))
print()

print("encoding['input_ids'][0]:")
print(encoding['input_ids'])
print()

print("len(encoding['attention_mask'][0]):")
print(len(encoding['attention_mask'][0]))
print()

print("encoding['attention_mask']:")
print(encoding['attention_mask'])
print()

print("tokenizer.convert_ids_to_tokens(encoding['input_ids'][0]):")
print(tokenizer.convert_ids_to_tokens(encoding['input_ids'][0]))
print()
```

Listing 7.6 starts with two `import` statements and then initializes the variables tokenizer and `nlp` in the same fashion as previous code samples. Next, the variable `text` is initialized as a text string, followed by the variable `encoding` that acts as a configuration-like "holder" of parameters and their values.

The final portion of Listing 7.6 consists of six pairs of `print()` statements, each of which displays a parameter/value pair that is defined in the `encoding` variable. Launch the code in Listing 7.6 to see the following output:

```
creating tokenizer...
creating model…

encoding.keys():
dict_keys(['input_ids', 'attention_mask'])
```

```
len(encoding['input_ids'][0]):
32

encoding['input_ids'][0]:
tensor([[ 101, 2043, 2020, 2017, 2197, 2648, 1029, 1045,
2031, 2042, 2503, 2005,
         1016, 3134, 1012,  102,    0,    0,    0,    0,
0,    0,    0,    0,
            0,    0,    0,    0,    0,    0,    0,    0]])

len(encoding['attention_mask'][0]):
32

encoding['attention_mask']:
tensor([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0,
0, 0, 0, 0, 0, 0, 0,
         0, 0, 0, 0, 0, 0, 0, 0]])

tokenizer.convert_ids_to_tokens(encoding['input_ids'][0]):
['[CLS]', 'when', 'were', 'you', 'last', 'outside', '?',
'i', 'have', 'been', 'inside', 'for', '2', 'weeks', '.',
'[SEP]', '[PAD]', '[PAD]', '[PAD]', '[PAD]', '[PAD]',
'[PAD]', '[PAD]', '[PAD]', '[PAD]', '[PAD]', '[PAD]',
'[PAD]', '[PAD]', '[PAD]', '[PAD]', '[PAD]']
```

SUBWORD TOKENIZATION

OOV is an acronym for *out of vocabulary*, and it refers to words in a corpus that do not belong to a vocabulary. When an OOV word is encountered, BERT splits the word into subwords, which is known as *subword tokenization*. The same process is applied to rare words.

Subword tokenization algorithms are based on a heuristic (something that's intuitive and often produces the correct answer). Specifically, words that appear more frequently are assigned unique IDs. Lower frequency words are split into subwords that retain the meaning of the lower frequency words. The following list contains four important subword tokenization algorithms:

- byte-pair encoding (BPE)
- SentencePiece
- unigram language model
- Wordpiece (used in BERT)

Byte-pair encoding for subwords represents frequent words with fewer symbols and less frequent words with more symbols. BPE is a bottom-up subword tokenization algorithm that learns a subword vocabulary of a certain size (the vocabulary size is a hyper parameter).

The first step in this technique involves splitting every word into Unicode characters, each of which corresponds to a symbol in the final vocabulary. Perform the following sequence of steps repeatedly:

1. Find the most frequent symbol bigram (pair of symbols).
2. Merge those symbols to create a new symbol and add this to the vocabulary.
3. Repeat the preceding steps until a maximum vocabulary size is reached.

GPT-2 views text input as a sequence of bytes instead of unicode characters; in addition, an `id` is allocated to every byte in the sequence.

Wordpiece is a subword tokenization algorithm that is very similar to BPE (discussed later). The main difference pertains to the specific manner in which bigrams are selected for the merging step. Interestingly, RoBERTa (which is based on BERT) also involves the use of Wordpiece. Here are some examples of subword tokenization in BERT:

```
"toppings"   is split into "topping" and "##s"
"trimmings"  is split into "trim", "##ming", and "##s"
"misspelled" is split into "mis", "##spel", and "##led"
```

However, keep in mind that BERT does *not* provide a mechanism to re-construct the original word from its word pieces. Note that ELMo provides word-level (not subword) contextual representations for words, which is different from `BERT`. Later in this chapter you will code samples that create BERT tokens from English sentences (that include toppings and trimmings).

Since word2vec and GloVe do not compute contextual word embeddings, the similarity between two embedded vectors may be of limited value.

BPE is an acronym for Byte Pair Encoding, which is an algorithm that is used in the GPT family of models. BPE (also known as *digram coding*) is a data compression algorithm that uses the following technique: given a text string, the most common pair of consecutive bytes of data is replaced with a byte that does exist in the text string. Each replacement is stored in a look-up table, which means that the table can be used to create the original text string. The models in the GPT family utilize a modified version of BPE.

For example, suppose we wanted to encode the data consisting of the following string:

```
aaabdaaabac
```

Since the byte pair `aa` occurs most often, we replace it with a character that does not appear in the string, such as the letter `Z`. We perform the replacement, which results in the following text string:

```
ZabdZabac (where Z=aa)
```

We repeat the substitution step, this time with the pair `ab`, and replace this pair with the letter `Y`:

```
ZYdZYac (where Y=ab Z=aa)
```

At this point, we can continue the preceding procedure by selecting `ZY` (which appears twice) and replacing this string with the letter `X`, as shown here:

```
XdXac (where X=ZY Y=ab Z=aa)
```

SentencePiece is another subword tokenizer and a detokenizer for NLP that performs subword segmentation. SentencePiece also supports BPE and the unigram language model. The original arxiv paper that describes SentencePiece is available online:

https://arxiv.org/abs/1808.06226v1

SENTENCE SIMILARITY IN BERT

As you learned previously, word2vec and GloVe use word embeddings to find the semantic similarity between two words. However, sentences contain additional information as well as relationships between multiple words.

A well-known example that clearly shows the need for contextual awareness is illustrated in the following pair of sentences:

The dog did not cross the street because it was too narrow.

The dog did not cross the street because it was too tired.

One technique for sentence similarity involves computing the average of the word embeddings of the words in each sentence and then computing the cosine similarity of the resulting pair of word embeddings. Alternatively, you can use tf-idf instead of word embeddings or another technique. In all of these cases, word order is not taken into account, and the word embeddings are determined in an unsupervised fashion.

Word Context in BERT

Listing 7.7 shows the content of `bert_context.py` that illustrates how BERT generates a different word vector for the same word that is used in a different context.

In order to launch the code in Listing 7.7, make sure that you have installed the transformers library by launching the following command in a command shell:

```
pip3 install transformers
```

NOTE *This code downloads a 536M BERT model.*

LISTING 7.7: bert_context.py

```
import transformers

text1 = "cell phone"

# instantiate a BERT tokenizer and model:
tokenizer = transformers.BertTokenizer.from_
pretrained('bert-base-uncased', do_lower_case=True)

nlp = transformers.TFBertModel.from_pretrained('bert-base-
uncased')

# hidden layer with embeddings:
input_ids1 = np.array(tokenizer.encode(text1))[None,:]
embedding1 = nlp(input_ids1)

# display text1 and its context:
print("text1:",text1)
print("embedding1[0][0]:")
print(embedding1[0][0])
print()

text2 = "prison cell"
# hidden layer with embeddings:
input_ids2 = np.array(tokenizer.encode(text2))[None,:]
embedding2 = nlp(input_ids2)

# display text2 and its context:
print("text2:",text2)
print("embedding2[0][0]:")
print(embedding2[0][0])
```

Listing 7.7 starts with `import` statements and then initializes the variables `tokenizer`, `nlp`, `input_ids1`, and `embedding1` in exactly the same manner that you have seen in previous code samples. The next block of code displays the values of `text1` and `embedding1[0][0]`.

The next portion of Listing 7.7 is virtually the same as the previous code block, based on the replacement of `text1` with `text2`. The output of Listing 7.7 is as follows:

```
input sentence #1:
text1: cell phone
embedding1[0][0]:
tf.Tensor(
[[-0.30501425  0.14509355 -0.18064171 ... -0.3127299  -0.12173399
  -0.09033043]
 [ 0.80547976 -0.15233847  0.61319923 ... -0.7498784   0.00167803
  -0.11698578]
 [ 1.0339862  -0.66511637 -0.17642722 ... -0.24407595  0.03978422
  -0.8694502 ]
 [ 0.87851435  0.10932285 -0.27658027 ...  0.18180653 -0.5829581
  -0.34113947]], shape=(4, 768), dtype=float32)
```

```
text2: cell mate
embedding2[0][0]:
tf.Tensor(
[[-0.24141303  0.1146469  -0.13710016 ... -0.2908613  -0.04577148
   0.2965925 ]
 [ 0.05608664 -1.0035615   0.12738925 ... -0.30271983  0.17530476
   0.7245784 ]
 [ 0.2818157  -0.28047347 -0.6547173  ...  0.04996978  0.01698243
   0.03285426]
 [ 1.039136    0.12364347 -0.2661501  ...  0.09439699 -0.7794917
  -0.24966209]], shape=(4, 768), dtype=float32)
```

Listing 7.7 also generates the following informative message:

```
Some weights of the model checkpoint at bert-base-uncased
were not used when initializing TFBertModel: ['nsp___cls',
'mlm___cls']
- This IS expected if you are initializing TFBertModel
from the checkpoint of a model trained on another
task or with another architecture (e.g. initializing
a BertForSequenceClassification model from a
BertForPretraining model).
- This IS NOT expected if you are initializing
TFBertModel from the checkpoint of a model that
you expect to be exactly identical (initializing
a BertForSequenceClassification model from a
BertForSequenceClassification model).
All the weights of TFBertModel were initialized from the
model checkpoint at bert-base-uncased.
If your task is similar to the task the model of the
ckeckpoint was trained on, you can already use TFBertModel
for predictions without further training.
```

Now that you have seen an example where BERT generates a different word vector for a word that is used in a different context, let's look at BERT tokens, which is the topic of the next section.

GENERATING BERT TOKENS (1)

Listing 7.8 shows the content of `bert_tokens1.py` that illustrates how to convert a text string to a BERT-compatible string and then tokenize the latter string into BERT tokens.

LISTING 7.8: bert_tokens1.py

```
from transformers import BertTokenizer, BertModel

tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')

text1 = "Pizza with four toppings and trimmings."
marked_text1 = "[CLS] " + text1 + " [SEP]"
tokenized_text1 = tokenizer.tokenize(marked_text1)
```

```
print("input sentence #1:")
print(text1)
print()

print("Tokens from input sentence #1:")
print(tokenized_text1)
print()

print("Some tokens in BERT:")
print(list(tokenizer.vocab.keys())[1000:1020])
print()
```

Listing 7.8 imports `BertTokenizer` and `BertModel`, and uses the former to initialize the variable `tokenizer`. Next, the variable `text1` is initialized to a text string, and `marked_text1` prepends `[CLS]` to `text1` and then appends `[SEP]` to `text1`. The last variable that is initialized is `tokenized_text1`, which is assigned the result of invoking the `tokenizer()` method on the variable `marked_text1`.

The next three blocks of `print()` statements display the contents of `text1`, `tokenized_text1`, and a range of 20 BERT tokens, respectively. Launch the code in Listing 7.8 to see the following output:

```
input sentence #1:
Pizza with four toppings and trimmings.

Tokens from input sentence #1:
['[CLS]', 'pizza', 'with', 'four', 'topping', '##s', 'and',
'trim', '##ming', '##s', '.', '[SEP]']

Some tokens in BERT:
['"', '#', '$', '%', '&', "'", '(', ')', '*', '+', ',',
'-', '.', '/', '0', '1', '2', '3', '4', '5']
```

GENERATING BERT TOKENS (2)

Listing 7.9 shows the content of `bert_tokens2.py` that illustrates how to convert a text string to a BERT-compatible string and then tokenize the latter string into BERT tokens.

LISTING 7.9: bert_tokens2.py

```
from transformers import BertTokenizer, BertModel

tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')

text2 = "I got a book and after I book for an hour, it's time to book it."
marked_text2 = "[CLS] " + text2 + " [SEP]"
tokenized_text2 = tokenizer.tokenize(marked_text2)

print("input sentence #2:")
print(text2)
print()
```

```
print("Tokens from input sentence #2:")
print(tokenized_text2)
print()

# Map token strings to their vocabulary indices:
indexed_tokens2 = tokenizer.convert_tokens_to_ids(tokenized_text2)

# Display the words with their indices:
for pair in zip(tokenized_text2, indexed_tokens2):
  print('{:<12} {:>6,}'.format(pair[0], pair[1]))
```

The first half of Listing 7.9 is almost identical to the first half of Listing 7.8, and uses the variable `text2` instead of `text1`.

The next portion of Listing 7.9 contains two blocks of `print()` statements that display the contents of `text2` and `tokenized_text2`, respectively. The next code snippet initializes the variable `indexed_tokens2` to the result of converting the tokens in `tokenized_text2` to `id` values.

The final portion of Listing 7.9 contains a loop that displays tokens and their associated `id` values. The output of Listing 7.9 is as follows:

```
input sentence #2:
I got a book and after I book for an hour, it's time to book it.

Tokens from input sentence #2:
['[CLS]', 'i', 'got', 'a', 'book', 'and', 'after', 'i', 'book',
'for', 'an', 'hour', ',', 'it', "'", 's', 'time', 'to', 'book',
'it', '.', '[SEP]']

[CLS]           101
i             1,045
got           2,288
a             1,037
book          2,338
and           1,998
after         2,044
i             1,045
book          2,338
for           2,005
an            2,019
hour          3,178
,             1,010
it            2,009
'             1,005
s             1,055
time          2,051
to            2,000
book          2,338
it            2,009
.             1,012
[SEP]           102
```

THE BERT FAMILY

BERT has spawned a remarkable set of variations of the original BERT model, each of which provides some interesting features. Some of those variations are listed here:

- ALBERT
- DistilBERT
- CamemBERT
- FlauBERT
- RoBERTa
- BIO BERT
- DOC BERT
- Clinical BERT
- German BERT

ALBERT (Google Research and Toyota Technological Institute) is an acronym for "A Lite BERT for Self-Supervised Learning of Language Representations." Like RoBERTa, ALBERT is significantly smaller than BERT, and it's also more capable than BERT.

ALBERT (unlike BERT) shares its parameters in all layers, which reduces the number of parameters, but has no effect on the training and inference time. In addition, ALBERT uses embedding matrix factorization, which further reduces the number of parameters. Furthermore, ALBERT uses SOP (Sentence-Order Prediction), which is an improvement over NSP (Next Sentence Prediction). Finally, ALBERT does not use a dropout rate, which further increases the model capacity.

ALBERT uses both whole-word masking and *n-gram masking*, where the latter refers to masking multiple sequential words. Here is a code snippet for ALBERT:

```
from transformers import AlbertForMaskedLM, AlbertTokenizer

model1 = AlbertForMaskedLM.from_pretrained('albert-xxlarge-v1')
tokenizer = AlbertTokenizer.from_pretrained('albert-xxlarge-v1')

model2 = AlbertForMaskedLM.from_pretrained('albert-xxlarge-v2')
tokenizer = AlbertTokenizer.from_pretrained('albert-xxlarge-v2')
```

DistilBERT is a smaller version of BERT that contains 66 million parameters, which is 40% of the number of parameters of BERT Base (which has 110 million parameters). Even so, DistilBERT achieves 97% of BERT accuracy and is 60% faster than BERT Base, which makes DistilBERT useful for transfer learning.

Knowledge distillation involves a small model (called the "student") that is trained to mimic a larger model or an ensemble of models (called the "teacher"). DistilBERT is an example of a distilled network that is also used in production.

To give you an idea of the type of code required for DistilBERT, here is an example of instantiating a DistilBERT tokenizer:

```
import transformers
tokenizer = transformers.AutoTokenizer.from_
pretrained('distilbert-base-uncased', do_lower_case=True)
```

Here is another example of instantiating a `DistilBERT` tokenizer:

```
from transformers import DistilBertTokenizer:
tokenizer = DistilBertTokenizer.from.
pretrained('distilbert-base-uncased')
```

RoBERTa (from Facebook) leverages BERT's language masking strategy, along with modifications of some of BERT's hyper parameters. Note that RoBERTa was trained on a corpus that is at least 10 times larger than the corpus for BERT.

Unlike BERT, RoBERTa does *not* use an NSP (Next Sentence Prediction) task. Instead, RoBERTa uses *dynamic masking*, whereby a masked token is actually modified during the training process.

Surpassing Human Accuracy: deBERTa

The deBERTa model from Microsoft recently surpassed human accuracy:

http s://www.microsoft.com/en-us /research/blog/microsoft-deberta-surpasses-human-performance-on-the-superglue-benchmark/

The architecture for this model comprises 48 Transformer layers with 1.5 billion parameters. This model has a GLUE score of 90.8, and a SuperGLUE score of 89.9, which exceeds the human performance score of 89.8.

Microsoft intends to integrate DeBERTa with the Turing natural language representation model Turing NLRv4 (also from Microsoft). The Turing models are ubiquitous in the Microsoft ecosystem, including products such as Bing and Azure Cognitive Services.

What is Google Smith?

The SMITH model from Google analyzes documents. Very simply, the SMITH model is trained to understand passages within the context of an entire document. By contrast, BERT is trained to understand words within the context of sentences. However, the SMITH model (which outperforms BERT) supplements BERT by performing major operations that are not possible in BERT.

This concludes the BERT-specific portion of the chapter. The next section introduces GPT, followed by sections that contain details regarding GPT-2 (and code samples) as well as GPT-3.

INTRODUCTION TO GPT

GPT stands for Generative Pre-Training (or sometimes called Generative Pre-Training Transformers), which is a pre-trained NLP-based model that was developed by OpenAI. GPT is trained with unlabeled data via unsupervised pre-training (also known as *self-supervision*).

GPT is based on the transformer architecture and takes advantage of the self-attention mechanism of the transformer. There are several versions of

GPT, which includes GPT-2 (developed in 2019) and GPT-3 that was released in June, 2020. Both GPT-2 and GPT-3 are discussed later in this chapter.

You should keep in mind the Lottery Ticket Hypothesis, which states that in every sufficiently deep neural network, there is a smaller subnetwork that can perform just as well as the whole neural network.

Installing the Transformers Package

The installation process involves the following command:

```
pip3 install transformers
```

You can perform an upgrade of transformers by invoking the following command:

```
pip3 install -U transformers
```

However, you might encounter the following error message:

```
ERROR: After October 2020 you may experience errors when
installing or updating packages. This is because pip will
change the way that it resolves dependency conflicts.

We recommend you use --use-feature=2020-resolver to test
your packages with the new resolver before it becomes the
default.

sentence-transformers 0.3.7.2 requires
transformers<3.4.0,>=3.1.0, but you'll have transformers
4.1.1 which is incompatible.
```

WORKING WITH GPT-2

This section contains Python code samples that use GPT-2 to perform sentiment analysis and the question-and-answer process, also abbreviated as QnA. There are some tasks that you can perform in GPT-2 that are comparable in GPT-3.

NOTE *The Python code samples in this section work with Python 3.7.9 but not with Python 3.6 or Python 3.8 (it's possible that Python 3.7.x will work as well).*

If you need to install Python 3.7.9, you will also need to execute the following commands to install transformers, tensorflow, and scipy:

```
pip3 install transformers
pip3 install tensorflow
pip3 install scipy
```

Listing 7.10 shows the content of `gpt2_sentiment.py` that illustrates how to perform sentiment analysis in GPT-2.

LISTING 7.10: gpt2_sentiment.py

```
# pip3 install transformers
from transformers import pipeline

# pipeline for sentiment-analysis:
cls = pipeline('sentiment-analysis')

text1 = "I love deep dish Chicago pizza."
sentiment1 = cls(text1)
print("sentence: ",text1)
print("sentiment:",sentiment1)
print()

text2 = "I dislike anchovies."
sentiment2 = cls(text2)
print("sentence: ",text2)
print("sentiment:",sentiment2)
print()

text3 = "I dislike anchovies but I like pickled herring."
sentiment3 = cls(text3)
print("sentence: ",text3)
print("sentiment:",sentiment3)
```

Listing 7.10 contains an `import` statement and then initializes the variable `cls` as an instance of the `pipeline` class by specifying `sentiment-analysis` (which is the task for this code sample).

The next three code blocks perform sentiment analysis on the text strings `text1`, `text2`, and `text3`, respectively. Launch the code to see the following output:

```
sentence:  I love deep dish Chicago pizza.
sentiment: [{'label': 'POSITIVE', 'score': 0.9985044598579407}]

sentence:  I dislike anchovies.
sentiment: [{'label': 'NEGATIVE', 'score': 0.9982384443283081}]

sentence:  I dislike anchovies but I like pickled herring.
sentiment: [{'label': 'POSITIVE', 'score': 0.7346124649047852}]
```

Listing 7.11 shows the content of `gpt2_qna.py` that illustrates how to perform question-and-answer in GPT-2. Note that this Python code sample will not work on Python 3.8.x. You must use Python 3.7.

LISTING 7.11: gpt2_qna.py

```
from transformers import pipeline

# pipeline for question-answering:
qna = pipeline('question-answering')

qc_pair = {
    'question': 'What is the name of the repository ?',
```

```
    'context': 'Pipeline have been included in the
huggingface/transformers repository'
}

if __name__ == "__main__":
  result = qna (qc_pair)
  print("result:")
  print(result)
```

Listing 7.11 starts with an `import` statement and then initializes the variable `qna` as an instance of the `pipeline` class from the `transformers` library, with `question-answering` as a parameter. Next, the variable `gc_pair` is initialized as a pair of question/answer strings.

Next, the variable `result` is initialized with the result of invoking `qna` with `gc_pair`, and then the contents of `result` are displayed. Launch the code to see the following output:

```
result:
{'score': 0.5135953426361084, 'start': 35, 'end': 59,
'answer': 'huggingface/transformers'}
```

Listing 7.11 contains an `if` statement; you might see the following error message if you remove the `if` statement:

```
#output:
    raise RuntimeError('''
RuntimeError:
        An attempt has been made to start a new process
before the current process has finished its bootstrapping
phase.

        This probably means that you are not using fork to
start your child processes and you have forgotten to use
the proper idiom
        in the main module:

            if __name__ == '__main__':
                freeze_support()
                ...

        The "freeze_support()" line can be omitted if the
program is not going to be frozen to produce an executable.
```

Listing 7.12 shows the content of `gpt2_text_gen.py` that illustrates how to use generated text from an input string in GPT-2. Note that the default model for the text generation pipeline is GPT-2.

LISTING 7.12: gpt2_text_gen.py

```
from transformers import pipeline

text_gen = pipeline("text-generation")

# specify a max_length of 50 tokens and sampling "off":
```

```
prefix_text = "What a wonderful"
generated_text = text_gen(prefix_text, max_length=50, do_sample=False)[0]

print("=> #1 generated_text['generated_text']:")
print(generated_text['generated_text'])
print("-------------------------------\n")

prefix_text = "Once in a "
generated_text = text_gen(prefix_text, max_length=50, do_sample=False)[0]

print("=> #2 generated_text['generated_text']:")
print(generated_text['generated_text'])
print("-------------------------------\n")

prefix_text = "Once in a blue "
generated_text = text_gen(prefix_text, max_length=50, do_sample=False)[0]

print("=> #3 generated_text['generated_text']:")
print(generated_text['generated_text'])
print("-------------------------------\n")
```

Listing 7.12 starts with an `import` statement and then initializes the variable `text_gen` as an instance of the `pipeline` class by specifying `text-generation` (which is the task for this code sample).

The next three blocks of code display the completion of the text in `prefix_text`, where the latter is assigned three different text strings. Launch the code in Listing 7.12 to see the following output:

```
=> #1 generated_text['generated_text']:
What a wonderful thing about this is that it's a very simple
and simple way to get your hands on a new game.

The game is a simple, simple game. It's a simple game. It's a
simple game. It's
-------------------------------

=> #2 generated_text['generated_text']:
Once in a vernacular, the word "carnage" is used to describe a
large, open, and well-lit place.

The word "carnage" is used to describe a large, open, and well-
-------------------------------

=> #3 generated_text['generated_text']:
Once in a blue urn, you can see the "C" in the center of the
"C" and the "A" in the bottom right corner.

The "C" is the "A" and the "A" are
-------------------------------
```

Listing 7.13 shows the content of `gpt2_auto.py` that illustrates how to perform sentiment analysis in GPT-2. Note that this Python code sample will not work on Python 3.8.x. You must use Python 3.7.

LISTING 7.13: gpt2_auto.py

```
from transformers import AutoTokenizer, TFAutoModel

tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")
mymodel = TFAutoModel.from_pretrained("bert-base-uncased")

inputs = tokenizer("I love deep dish Chicago pizza", return_tensors="tf")
outputs = mymodel(**inputs)

print("inputs:  ",inputs)
print("outputs: ",outputs)
```

Listing 7.13 starts with an `import` statement and then initializes the variable `tokenizer` as a generic tokenizer class from `bert-base-uncased` by invoking the `from_pretrained()` method of the `AutoTokenizer` class that belongs to the `transformers` library.

Similarly, `mymodel` is a general model class from `bert-base-uncased` by invoking the `from_pretrained()` method of the `TFAutoModel` class that belongs to the `transformers` library.

Next, the variable `inputs` is initialized with the result of passing a hard-coded string to the `tokenizer` variable. Then the variable `outputs` is initialized with the result of passing `inputs` to the variable `mymodel`.

The last portion of Listing 7.13 shows the contents of inputs and outputs. Launch the code to see the following output:

```
inputs:   {'input_ids': <tf.Tensor: shape=(1, 8), dtype=int32,
numpy=array([[  101,  1045,  2293,  2784,  9841,  3190, 10733,   102]],
      dtype=int32)>, 'token_type_ids': <tf.Tensor: shape=(1, 8),
dtype=int32, numpy=array([[0, 0, 0, 0, 0, 0, 0, 0]], dtype=int32)>,
'attention_mask': <tf.Tensor: shape=(1, 8), dtype=int32,
numpy=array([[1, 1, 1, 1, 1, 1, 1, 1]], dtype=int32)>}
outputs:  TFBaseModelOutputWithPooling(last_hidden_state=<tf.Tensor:
shape=(1, 8, 768), dtype=float32, numpy=
array([[[-0.00286604,  0.22725284,  0.0192489 , ..., -0.16997483,
          0.22732456,  0.2084062 ],
        [ 0.37293857,  0.18514417, -0.1804212 , ..., -0.02841423,
          0.92029154,  0.08076832],
        [ 1.0605763 ,  0.68393016,  0.3488946 , ...,  0.23068337,
          0.57474136, -0.2725499 ],
        ...,
        [ 0.36834046,  0.09277615, -0.49751407, ..., -0.21702018,
         -0.15317607, -0.17662546],
        [ 0.2218363 , -0.1452129 , -0.6224062 , ...,  0.19659105,
          0.0055675 ,  0.05520308],
        [ 0.38959947,  0.1536812 , -0.2523777 , ...,  0.3461408 ,
         -0.5905776 , -0.2758692 ]]],
dtype=float32)>, pooler_output=<tf.Tensor: shape=(1, 768), dtype=float32,
...
numpy=array([[-8.23929489e-01, -2.69686729e-01,  2.79440969e-01,
         5.52639008e-01, -5.11318594e-02, -8.98018852e-02,
         7.92447925e-01,  1.49121523e-01,  4.11069989e-02,
        -9.99752760e-01,  2.84106694e-02,  4.69654143e-01,
```

```
         9.74410057e-01, -2.57081628e-01,  9.02504683e-01,
        -4.83381122e-01,  3.19796950e-02, -5.14692605e-01,
...
         3.13184172e-01,  3.45878363e-01,  7.98233569e-01,
         4.64420468e-01,  6.13458335e-01,  4.65085119e-01,
         2.03554392e-01, -5.93035281e-01,  8.85935843e-01]],
dtype=float32)>, hidden_states=None, attentions=None)
```

Listing 7.14 shows the content of `pytorch_gpt_next_word.py` that illustrates how to predict the next word in a sentence.

LISTING 7.14: pytorch_gpt_next_word.py

```
import torch
from pytorch_transformers import GPT2Tokenizer, GPT2LMHeadModel

# Load pre-trained GPT-2 tokenizer model:
tokenizer = GPT2Tokenizer.from_pretrained('gpt2')

# encode the words in a sentence:
text = "What is the fastest car in the"
indexed_tokens = tokenizer.encode(text)

# convert tokens to a PyTorch tensor:
tokens_tensor = torch.tensor([indexed_tokens])

# load pre-trained model (weights)
model = GPT2LMHeadModel.from_pretrained('gpt2')

# "eval" mode deactivates the DropOut modules:
model.eval()

# Predict each token:
with torch.no_grad():
  outputs = model(tokens_tensor)
  predictions = outputs[0]

print("=> list of predictions:")
print(predictions[0, -1, :])
print()

print("=> argmax of predictions:")
print(torch.argmax(predictions[0, -1, :]).item())
print()

# Get the predicted next sub-word
predicted_index = torch.argmax(predictions[0, -1, :]).item()
predicted_text = tokenizer.decode(indexed_tokens + [predicted_index])

# Print the predicted word
print("=> initial text:")
print(text)
print()

print("=> Predicted next word:")
print(predicted_text)
```

Listing 7.14 starts with two `import` statements, the second of which is sort of like the counterpart to the `import` statement in Listing 7.13:

```
from transformers import AutoTokenizer, TFAutoModel
```

Next, the variable `tokenizer` is created as an instance of a generic `gpt2` model. Then, the variable `text` is initialized as a text string and passed as a parameter to the `encode()` method of the variable `tokenizer`, with the result assigned to the variable `indexed_tokens`.

The next portion of Listing 7.14 creates the variable `tokens_tensor`, which is a Torch-based tensor that is created from `indexed_tokens`. Now we can instantiate the variable `model` as a generic instance of the `gpt2` model.

The second half of Listing 7.14 starts by initializing the variables `outputs` and `predictions`, followed by blocks of `print()` statements that display the values of `predictions` and the index position with the maximum value. Then, the initial text is displayed, followed by the initial text concatenated with the predicted word **`world`** (that is shown in bold below).

The output of Listing 7.14 is here (this might take a minute or two when you launch it the first time due to a file download):

```
=> list of predictions:
tensor([ -96.1219,  -94.2472,  -96.9560,  ..., -103.5570, -100.5182,
         -95.6672])

=> argmax of predictions:
995

=> initial text:
What is the fastest car in the

=> Predicted next word:
 What is the fastest car in the world
```

GPT-2 versus BERT

There are some important differences between GPT-2 and BERT. Specifically, GPT-2 is *not* bidirectional and has no concept of masking. GPT-2 is based on transformer decoder blocks. Moreover, GPT-2 involves supervised fine-tuning and outputs only one token at a time.

By contrast, BERT adds the NSP task during training and also has a segment embedding. BERT uses transformer encoder blocks (not the decoder blocks) and also requires pre-training. The fine-tuning process necessitates task-specific sample data.

WHAT IS GPT-3?

GPT-3 is an extension of the GPT-2 model, contains more layers and data, and is 100 times larger than GPT-2. The largest GPT-3 model has 96 attention layers, each of which contains 96 × 128 dimension heads. In addition, GPT-3 consists of 175 billion parameters and was trained on hundreds of gigabytes of text to learn how to predict the next word in a user-supplied text string.

GPT-3 is a statistical model that determines the probability distribution of words in order to generate the appropriate text in response to an input string.

Unlike other models that are trained on specific tasks, GPT-3 is designed to detect *patterns* in text strings, which provides the following benefits:

- GPT-3 is a general purpose (not task-specific) model
- GPT-3 does not require re-training to handle new prompts
- GPT-3 achieves SOTA (state of the art) performance on multiple NLP tasks

The performance of GPT-3 is probably due to the fact that GPT-3 was trained on a dataset consisting of more than 50 billion words. Although the size and cost of GPT-3 is prohibitive for the vast majority of companies, GPT-3 is accessible via a simple and cost-effective API. GPT-3 is based on point-in-time data that can be several months old instead of continuous training.

GPT-3 Task Strengths and Mistakes

GPT-3 has the ability to perform text generation that is close to human-level quality. For example, suppose that GPT-3 is given a title and a subtitle, along with the word "article" that serves as a prompt word. GPT-3 can then write brief articles that often seem to be written by humans.

However, any trained model has limitations, including GPT-3. Moreover, bias exists in the corpus that was used to train GPT-3. According to the following article, one way in which GPT-3 can misclassify results is to include bias toward women and minorities:

https://techcrunch.com/2020/08/07/here-are-a-few-ways-gpt-3-can-go-wrong/

A more significant example is the use of GPT-3 in a medical chatbot, which suggested to a fake patient (who expressed suicidal thoughts) that he kill himself:

https://artificialintelligence-news.com/2020/10/28/medical-chatbot-openai-gpt3-patient-kill-themselves/

GPT-3 Architecture

GPT-3 has eight different model sizes (from 125M to 175B parameters), and the smallest GPT-3 model is about the size of BERT-Base and RoBERTa-Base, with 12 attention layers that in turn have 12 × 64 dimensional heads. However, the largest GPT-3 model is ten times larger than T5-11B (the previous record holder), and has 96 attention layers, which in turn have 96 × 128 dimension heads.

The GPT-3 Playground

OpenAI provides the Playground, which is a Web-based tool for entering prompts in a text field and receiving completions from GPT-3. The Playground supports most of the functionality that is available directly through the GPT-3 API. Moreover, the Playground enables you to interact with GPT-3 without writing any code.

Accessing the GPT-3 Playground

Log in at *https://openai.com*, after which you can access the Playground if you are a registered user. Next, enter a text string, click the submit button, and then GPT-3 will display its response text.

Here is an interesting feature: each time that you click the submit button, GPT-3 uses the existing text as a prompt to generate a new completion. Try clicking the submit button repeatedly, and watch the response text that GPT-3 generates for you.

WHAT IS THE GOAL OF GPT-3?

The aim of the GPT-3 pre-trained model is to directly evaluate the model on the test-related data of new tasks; i.e., GPT-3 essentially skips the training-related data of new tasks and focuses directly on the test-related data, in its capacity as a "few shot" learner (discussed later).

By way of comparison, GPT-3 has 175 billion parameters, whereas GPT-2 has 1.5 billion parameters, and BERT Large has 340 million parameters. GPT-3 was trained entirely on publicly available datasets, on nearly 500,000,000,000 words (some of which might contain offensive content).

GPT-3 achieved state-of-the-art performance on several NLP tasks without fine-tuning, at the cost of over $10,000,000. Some of the datasets that were used to train GPT-3 are downloadable from this read-only Github repository:

https://github.com/openai/gpt-3

GPT-3 has caught the attention of many people because of various tasks that it has performed, including automatic code generation. For example, one user typed a paragraph of text describing the following Web application:

1. a button that increments a total by USD 3
2. a button that decrements a total by USD 5
3. a button that displays the current total

GPT-3 then created a React application with the preceding functionality, which prompted a variety of reactions: some people were amused by such a simplistic application, whereas others contemplated their future job security.

Give GPT-3 an initial sequence of words and GPT-3 will generate various responses, such as code generation, news articles, poems, and even make jokes.

GPT-3 generated an interesting poem about Elon Musk ("your tweets are a blight"), part of which you can read online:

https://www.businessinsider.com/elon-musk-poem-tweets-gpt-3-openai-2020-8

The key differentiator of GPT-3 is its ability to perform specific tasks without the need for fine-tuning, whereas other models tend to require task-specific datasets, and they generally do not perform as well on other tasks.

One of the distinguishing characteristics of GPT-3 is its ability to solve unseen NLP tasks. This is due to the fact that GPT-3 was trained on a very large corpus. GPT-3 also uses “few shot” learning (discussed later in this chapter) and can perform the following tasks:

- translate natural language into code for websites
- solve complex medical question-and-answer problems
- create tabular financial reports
- write code to train machine learning models

The GPT-3 API involves setting the temperature parameter as well as the response length parameter. The temperature parameter (whose default value is 0.7) affects how much randomness the system uses in generating its replies. The response length parameter yields an approximate number of “words” the system generates in its response.

GPT-3 has surprised people with its capacity to generate prose as well as poetry. Elon Musk is one of the founding members of OpenAI that created GPT-3, which generated the following poem about Elon Musk[1]:

“The SEC said, “Musk,/your tweets are a blight./They really could cost you your job,/if you don’t stop/all this tweetingat night.”/…Then Musk cried, “Why?/The tweets I wrote are not mean,/I don’t use all-caps/and I’m sure that my tweets are clean.”/”But your tweets can move markets/and that’s why we’re sore./You may be a genius/and a billionaire,/but that doesn’t give you the right to be a bore!”

Zero-Shot, One-Shot, and Few Shot Learners

These three types of learners differ in the number of task examples that they are given and also the number of gradient updates that they perform.

Specifically, a *zero-shot* learner is a model that predicts an answer based solely on an NLP description of the task: *no gradient updates are performed*.

A *one-shot* learner is a model that 1) sees a description of the task and 2) one example of the task: *no gradient updates are performed*.

A *few-shot* learner is a model that 1) sees a description of the task and 2) a few examples of the task: *no gradient updates are performed*, and a “few” examples can involve between 10 and 100 examples of the task.

With the preceding points in mind, GPT-3 is a “few shot” learner because GPT-3 is fine-tuned on a small set of samples. By contrast, most other models (including BERT) require an elaborate fine-tuning step.

GPT-3 TASK PERFORMANCE

For most models, the task of translating sentences from English to Italian involves thousands of sentence pairs in order for those models to learn how

[1] *https://www.businessinsider.com/elon-musk-poem-tweets-gpt-3-openai-2020-8*

to perform translation. By comparison, GPT-3 does not require a fine-tuning step: it can handle custom language tasks without training data.

Thus, GPT-3 has the ability to perform specific tasks without any special tuning, which is something that other models cannot do well. For example, GPT-3 can be trained to translate text, generate code, or even write poetry. Moreover, GPT-3 can do so with no more than 10 training examples.

One other point to keep in mind: GPT-3 is not just a few-shot learner. It can also perform as a zero-shot learner and a one-shot learner. By way of comparison, GPT-3 as a zero-shot learner has higher accuracy than a fine-tuned RoBERTa model (which previously had SOTA performance).

In terms of reading comprehension, GPT-3 performs best on free-form conversational datasets, and performs its worst on datasets that involve modeling structured dialog. However, as a few-shot learner for this task, GPT-3 outperforms the fine-tuned baseline of BERT. In addition, GPT-3 performs well on the SQuAD 2.0 dataset from Stanford, but underperforms on multiple-choice test questions.

GPT-3 treats each input string as a so-called "prompt" in order to determine the most suitable response: higher quality prompts generate higher quality responses. A completion is another term for the response string that is generated by `GPT-3`. Examples of `GPT-3` are available online:

https://beta.openai.com/examples

THE SWITCH TRANSFORMER: ONE TRILLION PARAMETERS

Recently Google researchers announced an NLP model with one trillion parameters, which is almost six times as larges at GPT-3 (175 billion parameters). This model is one of the largest models ever created, and it is as much as four times faster than T5-XXL (the previous largest language model from Google).

Instead of using complicated algorithms, the researchers combined a simple architecture in conjunction with large datasets and parameter counts. Since large-scale training is computationally intensive, they adopted a Switch Transformer, which is a technique that uses only a subset of the parameters of a model. In addition to the model's sparseness, the Switch Transformer adroitly takes advantage of GPUs and TPUs for intense matrix multiplications operations.

LOOKING AHEAD

Several important topics are not discussed in this chapter. For example, the topic of ethics is much more visible than it was even just a few years ago. Various questions have become more prominent in AI, such as the ethical concerns associated with large-scale deployment of AI system, how algorithms contribute decision-making processes, the source of data and the extent of biases in that data.

In health care, questions arise regarding AI-controlled robots prescribing medicine and performing surgery. Moreover, there are legal issues and accountability when robots make mistakes, such as who is responsible (the owner or the robot manufacturer?) and determining the type of penalty to impose (deactivate one robot or every robot in the same series?).

In parallel with the preceding issues, recent developments in AI are creating a sense of optimism that breakthroughs may well be on the event horizon. Recently OpenAI created DALL-E (derived from Salvador Dali and Pixar's WALL-E), which is 12-billion parameter variation of GPT-3: *https://openai.com/blog/dall-e/*

In addition, DeepMind developed AlphaFold, which made a significant contribution toward solving the protein folding problem (which has been called a "50-year-old problem in biology"). AlphaFold handily won the competition (by a substantial margin).

To give you an idea of the impact of AlphaFold, Andrei Lupas, who is an evolutionary biologist at the Max Planck Institute for Developmental Biology in Tübingen, Germany, stated the following: "The [AlphaFold] model from group 427 gave us our structure in half an hour, after we had spent a decade trying everything."

Indeed, the future of NLP and AI in general looks both challenging and promising, guided by ethical principles that may lead us to a more mindful way of life.

SUMMARY

This chapter started with an introduction to the concept of attention, followed by the transformer architecture that was developed by Google and released in late 2017. You also learned how to use the transformer model from HuggingFace to perform tasks such as NER, QnA, Sentiment Analysis, and mask-filling tasks.

Next, you learned about BERT, which is a pre-trained NLP model that is based on the transformer architecture, along with some of its features. You also saw how to perform sentence similarity in BERT, and how to generate BERT tokens. Then you learned about several BERT-based trained models, including DistilledBERT, CamemBERT, and FlauBERT.

In the final portion of this chapter, you learned about GPT-3 and some of its remarkable features, and its strengths as well as its weaknesses. You also learned about various types of learners and how GPT-3 was trained.

Congratulations! You have reached the end of a fast-paced introduction to R and NLP in R. You are in a good position to use the knowledge that you acquired in this book as a stepping stone to further your understanding of NLP.

INTRO TO PROBABILITY AND STATISTICS

This appendix introduces you to concepts in probability, as well as a wide assortment of statistical terms and algorithms.

The first section of this appendix starts with a discussion of probability, how to calculate the expected value of a set of numbers (with associated probabilities), and the concept of a random variable (discrete and continuous), and a short list of some well-known probability distributions.

The second section of this appendix introduces basic statistical concepts, such as mean, median, mode, variance, and standard deviation, along with simple examples that illustrate how to calculate these terms. You will also learn about the terms RSS, TSS, R^2, and F1 score.

The third section of this appendix introduces Gini Impurity, Entropy, Perplexity, Cross-Entropy, and KL Divergence. You will also learn about skewness and kurtosis.

The fourth section explains covariance and correlation matrices and how to calculate eigenvalues and eigenvectors.

The fifth section explains PCA (Principal Component Analysis), which is a well-known dimensionality reduction technique. The final section introduces you to Bayes' Theorem.

WHAT IS A PROBABILITY?

All measurements have some uncertainty. In general, we assume that there is a correct value, and we endeavor to find the best estimate of that value.

When we work with an event that can have multiple outcomes, we try to define the probability of an outcome as the chance that it will occur, which is calculated as follows:

p(outcome) = # of times outcome occurs/(total number of outcomes)

For example, in the case of a single balanced coin, the probability of tossing a head H equals the probability of tossing a tail T:

p(H) = 1/2 = p(T)

The set of probabilities associated with the outcomes {H, T} is shown in the set P:

P = {1/2, 1/2}

Some experiments involve replacement while others involve non-replacement. For example, suppose that an urn contains 10 red balls and 10 green balls. What is the probability that a randomly selected ball is red? The answer is 10/(10 + 10) = 1/2. What is the probability that the second ball is also red?

There are two scenarios with two different answers. If each ball is selected with replacement, then each ball is returned to the urn after selection, which means that the urn always contains 10 red balls and 10 green balls. In this case, the answer is 1/2 * 1/2 = 1/4. In fact, the probability of any event is independent of all previous events (with replacement).

However, if balls are selected without replacement, then the answer is 10/20 * 9/19. As you undoubtedly know, card games are also examples of selecting cards without replacement.

Another concept is *conditional probability*, which refers to the likelihood of the occurrence of event E1 given that event E2 has occurred. A simple example is the following statement:

"If it rains (E2), then I will carry an umbrella (E1)."

Calculating the Expected Value

Consider the following scenario involving a well-balanced coin: whenever a head appears, you earn $1 and whenever a tail appears, you earn $1. If you toss the coin 100 times, how much money do you expect to earn? Since you will earn $1 regardless of the outcome, the expected value (in fact, the guaranteed value) is 100.

Now consider this scenario: whenever a head appears, you earn $1 and whenever a tail appears, you earn 0 dollars. If you toss the coin 100 times, how much money do you expect to earn? You probably determined the value $50 (which is the correct answer) by making a quick mental calculation. The more formal derivation of the value of E (the expected earning) is here:

E = 100 *[1 * 0.5 + 0 * 0.5] = 100 * 0.5 = 50

The quantity **1** * 0.5 + **0** * 0.5 is the amount of money you expected to earn during each coin toss (half the time you earn $1 and half the time you earn 0 dollars), and multiplying this number by 100 is the expected earning after 100 coin tosses. Also note that you might never earn $50: the actual amount that you earn can be any integer between 1 and 100 inclusive.

As another example, suppose that you earn \$3 whenever a head appears, and you *lose* \$1.50 dollars whenever a tail appears. Then the expected earning E after 100 coin tosses is shown here:

E = 100 *[3 * 0.5 – 1.5 * 0.5] = 100 * 1.5 = 150

We can generalize the preceding calculations as follows. Let P = {p1, . . . , pn} be a probability distribution, which means that the values in P are non-negative and their sum equals 1. In addition, let R = {R1, . . . , Rn} be a set of rewards, where reward Ri is received with probability pi. Then the expected value E after N trials is as follows:

E = N * [SUM pi*Ri]

In the case of a single balanced die, we have the following probabilities:

p(1) = 1/6
p(2) = 1/6
p(3) = 1/6
p(4) = 1/6
p(5) = 1/6
p(6) = 1/6
P = { 1/6, 1/6, 1/6, 1/6, 1/6, 1/6}

As a simple example, suppose that the earnings are {3, 0, −1, 2, 4, −1} when the values 1, 2, 3, 4, 5, 6, respectively, appear when tossing the single die. Then after 100 trials our expected earnings are calculated as follows:

E = 100 * [3 + 0 + −1 + 2 + 4 + −1]/6 = 100 * 3/6 = 50

In the case of two balanced dice, we have the following probabilities of rolling 2, 3, . . . , or 12:

p(2) = 1/36
p(3) = 2/36
...
p(12) = 1/36
P = {1/36, 2/36, 3/36, 4/36, 5/36, 6/36, 5/36, 4/36, 3/36, 2/36, 1/36}

RANDOM VARIABLES

A random variable is a variable that can have multiple values, and each value has an associated probability of occurrence. For example, if we let X be a random variable whose values are the outcomes of tossing a well-balanced die, then the values of X are the numbers in the set {1, 2, 3, 4, 5, 6}. Moreover, each of those values can occur with equal probability (which is 1/6).

In the case of two well-balanced dice, let X be a random variable whose values can be any of the numbers in the set {2, 3, 4, . . . , 12}. Then the associated probabilities for the different values for X are listed in the previous section.

Discrete versus Continuous Random Variables

The preceding section contains examples of *discrete* random variables because the list of possible values is either finite or countably infinite (such as the set of integers). As an aside, the set of rational numbers and the set of algebraic numbers are also countably infinite, but the set of non-algebraic irrational numbers and the set of real numbers are both uncountably infinite (proofs are available online). As pointed out earlier, the associated set of probabilities must form a probability distribution, which means that the probability values are non-negative and their sum equals 1.

A *continuous* random variable is a variable whose values can be *any* number in an interval, which can be an uncountably infinite number of values. For example, the amount of time required to perform a task is represented by a continuous random variable.

A continuous random variable also has a probability distribution that is represented as a continuous function. The constraint for such a variable is that the area under the curve (which is sometimes calculated via a mathematical integral) equals 1.

Well-Known Probability Distributions

There are many probability distributions, and some of the well-known probability distributions are listed here:

- Gaussian distribution
- Poisson distribution
- Chi-squared distribution
- Binomial distribution

The Gaussian distribution is named after Karl F. Gauss, and it is sometimes called the *normal distribution* or the *Bell curve*. The Gaussian distribution is symmetric: the shape of the curve on the left of the mean is identical to the shape of the curve on the right side of the mean. As an example, the distribution of IQ scores follows a curve that is similar to a Gaussian distribution.

However, the frequency of traffic at a given point in a road follows a Poisson distribution (which is not symmetric). If you count the number of people who go to a public pool based on five-degree (Fahrenheit) increments of the temperature, followed by five-degree decrements in temperature, that set of numbers follows a Poisson distribution.

Perform an Internet search for each of the bullet items in the preceding list and you will find numerous articles that contain images and technical details about these (and other) probability distributions.

FUNDAMENTAL CONCEPTS IN STATISTICS

This section contains several subsections that discuss the mean, median, mode, variance, and standard deviation. Feel free to skim (or skip) this section if you are already familiar with these concepts. As a start point, let's suppose that we have a set of numbers X ={x1, . . . , xn} that can be positive, negative, integer-valued or decimal values.

The Mean

The *mean* of the numbers in the set X is the average of the values. For example, if the set X consists of {–10, 35, 75, 100}, then the mean equals (–10 + 35 + 75 + 100)/4 = 50. If the set X consists of {2, 2, 2, 2}, then the mean equals (2 + 2 + 2 + 2)/4 = 2. The mean value is not necessarily one of the values in the set.

The mean is sensitive to outliers. For example, the mean of the set of numbers {1, 2, 3, 4} is 2.5, whereas the mean of the set of number {1, 2, 3, 4, 1000} is 202. Since the formulas for the variance and standard deviation involve the mean of a set of numbers, both of these terms are also more sensitive to outliers.

The Median

The *median* of the numbers (sorted in increasing or decreasing order) in the set X is the middle value in the set of values, which means that half the numbers in the set are less than the median and half the numbers in the set are greater than the median. For example, if the set X consists of {–10, 35, 75, 100}, then the median equals 55 because 55 is the average of the two numbers 35 and 75. As you can see, half the numbers are less than 55 and half the numbers are greater than 55. If the set X consists of {2, 2, 2, 2}, then the median equals 2.

By contrast, the median is much less sensitive to outliers than the mean. For example, the median of the set of numbers {1, 2, 3, 4} is 2.5, and the median of the set of numbers {1, 2, 3, 4, 1000} is 3.

The Mode

The *mode* of the numbers (sorted in increasing or decreasing order) in the set X is the most frequently occurring value, which means that there can be more than one such value. If the set X consists of {2, 2, 2, 2}, then the mode equals 2.

If X is the set of numbers {2, 4, 5, 5, 6, 8}, then the number 5 occurs twice and the other numbers occur only once, so the mode equals 5.

If X is the set of numbers {2, 2, 4, 5, 5, 6, 8}, then the numbers 2 and 5 occur twice and the other numbers occur only once, so the mode equals 2 and 5. A set that has two modes is called bimodal, and a set that has more than two modes is called *multi-modal*.

One other scenario involves sets that have numbers with the same frequency and they are all different. In this case, the mode does not provide meaningful information, and one alternative is to partition the numbers into

subsets and then select the largest subset. For example, if set X has the values {1, 2, 15, 16, 17, 25, 35, 50}, we can partition the set into subsets whose elements are in range that are multiples of ten, which results in the subsets {1, 2}, {15, 16, 17}, {25}, {35}, and {50}. The largest subset is {15, 16, 17}, so we could select the number 16 as the mode.

As another example, if set X has the values {–10, 35, 75, 100}, then partitioning this set does not provide any additional information, so it's probably better to work with either the mean or the median.

The Variance and Standard Deviation

The *variance* is the sum of the squares of the difference between the numbers in X and the mean mu of the set X, divided by the number of values in X, as shown here:

```
variance = [SUM (xi – mu)**2 ] / n
```

For example, if the set X consists of {–10, 35, 75, 100}, then the mean equals (–10 + 35 + 75 + 100)/4 = 50, and the variance is computed as follows:

```
variance = [(–10–50)**2 + (35–50)**2 + (75–50)**2 + (100–50)**2]/4
         = [60**2 + 15**2 + 25**2 + 50**2]/4
         = [3600 + 225 + 625 + 2500]/4
         = 6950/4 = 1,737.50
```

The standard deviation std is the square root of the variance:

```
std = sqrt(1737) = 41.677
```

If the set X consists of {2, 2, 2, 2}, then the mean equals (2 + 2 + 2 + 2)/4 = 2, and the variance is computed as follows:

```
variance = [(2–2)**2 + (2–2)**2 + (2–2)**2 + (2–2)**2]/4
         = [0**2 + 0**2 + 0**2 + 0**2]/4
         = 0
```

The standard deviation std is the square root of the variance:

```
std = sqrt(0) = 0
```

Population, Sample, and Population Variance

The *population* specifically refers to the entire set of entities in a given group, such as the population of a country, the people over 65 in the USA, or the number of first year students in a university.

However, in many cases statistical quantities are calculated on samples instead of an entire population. Thus, a sample is (a much smaller) subset of the given population. See the Central Limit Theorem regarding the distribution of

the mean of a set of samples of a population (which need not be a population with a Gaussian distribution).

If you want to learn about techniques for sampling data, here is a list of three different techniques that you can investigate:

- Stratified sampling
- Cluster sampling
- Quota sampling

The population variance is calculated by multiplying the sample variance by n/(n−1), as shown here:

population variance = [n/(n−1)]*variance

Chebyshev's Inequality

Chebyshev's inequality provides a simple way to determine the minimum percentage of data that lies within k standard deviations. Specifically, this inequality states that for any positive integer k greater than 1, the amount of data in a sample that lies within k standard deviations is at least 1 – 1/k**2. For example, if k = 2, then at least 1 – 1/2**2 = 3/4 of the data must lie within 2 standard deviations.

The interesting part of this inequality is that it has been mathematically proven to be true, i.e., it's not an empirical or heuristic-based result. An extensive description regarding Chebyshev's inequality (including some advanced mathematical explanations) is available online:

https://en.wikipedia.org/wiki/Chebyshev%27s_inequality

What is a p-value?

The null hypothesis states that there is no correlation between a dependent variable (such as y) and an independent variable (such as x). The p-value is used to reject the null hypothesis if the *p-value* is small enough (< 0.005) which indicates a higher significance. The threshold value for p is typically 1% or 5%.

There is no straightforward formula for calculating p-values, which are values that are always between 0 and 1. In fact, p-values are statistical quantities to evaluate the null hypothesis, and they are calculated by means of p-value tables or via spreadsheet/statistical software.

THE MOMENTS OF A FUNCTION (OPTIONAL)

The previous sections describe several statistical terms that is sufficient for the material in this book. However, several of those terms can be viewed from the perspective of different moments of a function.

In brief, the moments of a function are measures that provide information regarding the shape of the graph of a function. In the case of a probability distribution, the first four moments are defined as follows:

- The mean is the first central moment
- The variance is the second central moment
- The skewness (discussed later) is the third central moment
- The kurtosis (discussed later) is the fourth central moment

More detailed information (including the relevant integrals) regarding moments of a function is available here:

https://en.wikipedia.org/wiki/Moment_(mathematics)#Variance

What is Skewness?

Skewness is a measure of the asymmetry of a probability distribution. A Gaussian distribution is symmetric, which means that its skew value is zero (it's not exactly zero, but close enough for our purposes). In addition, the skewness of a distribution is the *third* moment of the distribution.

A distribution can be skewed on the left side or on the right side. A *left-sided skew* means that the long tail is on the left side of the curve, with the following relationships:

mean < median < mode

A *right-sided skew* means that the long tail is on the right side of the curve, with the following relationships (compare with the left-sided skew):

mode < median < mean

If need be, you can transform skewed data to a normally distributed dataset using one of the following techniques (which depends on the specific use-case):

- Exponential transform
- Log transform
- Power transform

Perform an online search for more information regarding the preceding transforms and when to use each of these transforms.

What is Kurtosis?

Kurtosis is related to the skewness of a probability distribution, in the sense that both of them assess the asymmetry of a probability distribution. The kurtosis of a distribution is a scaled version of the *fourth* moment of the distribution, whereas its skewness is the *third* moment of the distribution. Note that the kurtosis of a univariate distribution equals 3.

If you are interested in learning about additional kurtosis-related concepts, you can perform an online search for information regarding mesokurtic, leptokurtic, and platykurtic types of "excess kurtosis."

DATA AND STATISTICS

This section contains various subsections that briefly discuss some of the challenges and obstacles that you might encounter when working with datasets. This section and subsequent sections introduce you to the following concepts:

- Correlation versus Causation
- The bias-variance tradeoff
- Types of bias
- The Central Limit Theorem
- Statistical inferences

Statistics typically involves data *samples*, which are subsets of observations of a population. The goal is to find well-balanced samples that provide a good representation of the entire population.

Although this goal can be very difficult to achieve, it's also possible to achieve highly accurate results with a very small sample size. For example, the Harris poll in the USA has been used for decades to analyze political trends. This poll computes percentages that indicate the favorability rating of political candidates, and it's usually within 3.5% of the correct percentage values. What's remarkable about the Harris poll is that its sample size is a mere 4,000 people that are from the US population that is greater than 325,000,000 people.

Another aspect to consider is that each sample has a mean and variance, which do not necessarily equal the mean and variance of the actual population. However, the expected value of the sample mean and variance equal the mean and variance, respectively, of the population.

The Central Limit Theorem

Samples of a population have an interesting property. Suppose that you take a set of samples {S1, S3, . . . , Sn} of a population and you calculate the mean of those samples, which is {m1, m2, . . . , mn}. The Central Limit Theorem is a remarkable result: given a set of samples of a population and the mean value of those samples, the distribution of the mean values can be approximated by a Gaussian distribution. Moreover, as the number of samples increases, the approximation becomes more accurate.

Correlation versus Causation

In general, datasets have some features (columns) that are more significant in terms of their set of values, and some features only provide additional information that does not contribute to potential trends in the dataset. For example, the passenger names in the list of passengers on the Titanic are unlikely to affect the survival rate of those passengers, whereas the gender of the passengers is likely to be an important factor.

In addition, a pair of significant features may also be "closely coupled" in terms of their values. For example, a real estate dataset for a set of houses will contain the number of bedrooms and the number of bathrooms for each house in the dataset. As you know, these values tend to increase together and also decrease together. Have you ever seen a house that has 10 bedrooms and 1 bathroom, or a house that has 10 bathrooms and 1 bedroom? If you did find such a house, would you purchase that house as your primary residence?

The extent to which the values of two features change is called their correlation, which is a number between −1 and 1. Two "perfectly" correlated features have a correlation of 1, and two features that are not correlated have a correlation of 0. In addition, if the values of one feature decrease when the values of another feature increase, and vice versa, then their correlation is closer to −1 (and might also equal −1).

However, causation between two features means that the values of one feature can be used to calculate the values of the second feature (within some margin of error).

Keep in mind this fundamental point about machine learning models: they can provide correlation but they cannot provide causation.

Statistical Inferences

Statistical thinking related processes and statistics, whereas statistical inference refers to the process by which the inferences that you make regarding a population. Those inferences are based on statistics that are derived from samples of the population. The validity and reliability of those inferences depend on random sampling in order to reduce bias. There are various metrics that you can calculate to help you assess the validity of a model that has been trained on a particular dataset.

STATISTICAL TERMS RSS, TSS, R^2, AND F1 SCORE

Statistics is extremely important in machine learning, so it's not surprising that many concepts are common to both fields. Machine learning relies on a number of statistical quantities in order to assess the validity of a model, some of which are listed here:

- RSS
- TSS
- R^2

The term `RSS` is the "residual sum of squares" and the term `TSS` is the "total sum of squares." Moreover, these terms are used in regression models.

As a starting point so we can simplify the explanation of the preceding terms, suppose that we have a set of points {(x1, y1), . . . , (xn, yn)} in the Euclidean plane. In addition, let's define the following quantities:

- (x, y) is any point in the dataset
- y is the y-coordinate of a point in the dataset
- y_ is the mean of the y-values of the points in the dataset
- y_hat is the y-coordinate of a point on a best-fitting line

Just to be clear, (x, y) is a point in the dataset, whereas (x, y_hat) is the corresponding point that lies on the *best fitting line*. With these definitions in mind, the definitions of RSS, TSS, and R^2 are listed here (n equals the number of points in the dataset):

- RSS = (y – **y_hat**)**2/n
- TSS = (y – **y_bar**)**2/n
- R^2 = 1 – RSS/TSS

We also have the following inequalities involving RSS, TSS, and R^2:

- 0 <= RSS
- RSS <= TSS
- 0 <= RSS/TSS <= 1
- 0 <= 1 – RSS/TSS <= 1
- 0 <= R^2 <= 1

When RSS is close to 0, then RSS/TSS is also close to zero, which means that R^2 is close to 1. Conversely, when RSS is close to TSS, then RSS/TSS is close to 1, and R^2 is close to 0. In general, a larger R^2 is preferred (i.e., the model is closer to the data points), but a lower value of R^2 is not necessarily a bad score.

What is an F1 score?

In machine learning, an F1 score is for models that are evaluated on a feature that contains categorical data, and the p-value is useful for machine learning in general. An F1 score is a measure of the accuracy of a test, and it's defined as the harmonic mean of precision and recall. Here are the relevant formulas, where p is the precision and r is the recall:

p = (# of correct positive results)/(# of all positive results)
r = (# of correct positive results)/(# of all relevant samples)

F1-score = 1/[((1/r) + (1/p))/2]
= 2*[p*r]/[p+r]

The best value of an F1 score is 0 and the worse value is 0. An F1 score is for categorical classification problems, whereas the R^2 value is typically for regression tasks (such as linear regression).

GINI IMPURITY, ENTROPY, AND PERPLEXITY

These concepts are useful for assessing the quality of a machine learning model and the latter pair are useful for dimensionality reduction algorithms.

Before we discuss the details of Gini impurity, suppose that P is a set of non-negative numbers {p1, p2, . . . , pn} such that the sum of all the numbers in the set P equals 1. Under these two assumptions, the values in the set P comprise a probability distribution, which we can represent with the letter p.

Now suppose that the set K contains a total of M elements, with k1 elements from class S1, k2 elements from class S2, . . . , and kn elements from class Sn. Compute the fractional representation for each class as follows:

$$p1 = k1/M, p2 = k2/M, \ldots, pn = kn/M$$

As you can surmise, the values in the set {p1, p2, . . . , pn} form a probability distribution. We're going to use the preceding values in the following subsections.

What is Gini Impurity?

The Gini impurity is defined as follows, where {p1, p2, . . . , pn} is a probability distribution:

$$\begin{aligned} \text{Gini} &= 1 - [p1*p1 + p2*p2 + \cdots + pn*pn] \\ &= 1 - \text{SUM } pi*pi \text{ (for all i, where } 1<=i<=n) \end{aligned}$$

Since each pi is between 0 and 1, then pi*pi <= pi, which means that

$$\begin{aligned} 1 &= p1 + p2 + \cdots + pn \\ &>= p1*p1 + p2*p2 + \cdots + pn*pn \\ &= \text{Gini impurity} \end{aligned}$$

Since the Gini impurity is the sum of the squared values of a set of probabilities, the Gini impurity cannot be negative. Hence, we have derived the following result:

$$0 <= \text{Gini impurity} <= 1$$

What is Entropy?

Entropy is a measure of the expected ("average") number of bits required to encode the outcome of a random variable. The calculation for the entropy H (the letter E is reserved for Einstein's formula) as defined via the following formula:

$$\begin{aligned} H &= (-1)*[p1*\log p1 + p2 * \log p2 + \cdots + pn * \log pn] \\ &= (-1)* \text{SUM } [pi * \log(pi)] \text{ (for all i, where } 1<=i<=n) \end{aligned}$$

Calculating Gini Impurity and Entropy Values

For our first example, suppose that we have three classes A and B and a cluster of 10 elements with 8 elements from class A and 2 elements from class B. Therefore, p1 and p2 are 8/10 and 2/10, respectively. We can compute the Gini score as follows:

$$\begin{aligned} \text{Gini} &= 1 - [p1*p1 + p2*p2] \\ &= 1 - [64/100 + 04/100] \\ &= 1 - 68/100 \\ &= 32/100 \\ &= 0.32 \end{aligned}$$

We can also calculate the entropy for this example as follows:

$$\begin{aligned} \text{Entropy} &= (-1)*[p1 * \log p1 + p2 * \log p2] \\ &= (-1)*[0.8 * \log 0.8 + 0.2 * \log 0.2] \\ &= (-1)*[0.8 * (-0.322) + 0.2 * (-2.322)] \\ &= 0.8 * 0.322 + 0.2 * 2.322 \\ &= 0.7220 \end{aligned}$$

For our second example, suppose that we have three classes A, B, C and a cluster of 10 elements with 5 elements from class A, 3 elements from class B, and 2 elements from class C. Therefore p1, p2, and p3 are 5/10, 3/10, and 2/10, respectively. We can compute the Gini score as follows:

$$\begin{aligned} \text{Gini} &= 1 - [p1*p1 + p2*p2 + p3*p3] \\ &= 1 - [25/100 + 9/100 + 04/100] \\ &= 1 - 38/100 \\ &= 62/100 \\ &= 0.62 \end{aligned}$$

We can also calculate the entropy for this example as follows:

$$\begin{aligned} \text{Entropy} &= (-1)*[p1 * \log p1 + p2 * \log p2] \\ &= (-1)*[0.5*\log0.5 + 0.3*\log0.3 + 0.2*\log0.2] \\ &= (-1)*[-1 + 0.3*(-1.737) + 0.2*(-2.322)] \\ &= 1 + 0.3*1.737 + 0.2*2.322 \\ &= 1.9855 \end{aligned}$$

In both examples, the Gini impurity is between 0 and 1. However, while the entropy is between 0 and 1 in the first example, it's greater than 1 in the second example (which was the rationale for showing you two examples).

A set whose elements belong to the same class has Gini impurity equal to 0 and also its entropy equal to 0. For example, if a set has 10 elements that belong to class S1, then

Gini = 1 – SUM pi*pi
= 1 – p1*p1
= 1 – (10/10)*(10/10)
= 1 – 1 = 0

Entropy = (–1)*SUM pi*log pi
= (–1) * p1*log pi
= (–1) * (10/10) * log(10/10)
= (–1)*1*0 = 0

Multi-Dimensional Gini Index

The Gini index is a one-dimensional index that works well because the value is uniquely defined. However, when working with multiple factors, we need a multidimensional index. Unfortunately, the multi-dimensional Gini index (MGI) is not uniquely defined. While there have been various attempts to define an MGI that has unique values, they tend to be non-intuitive and mathematically much more complex. More information about MGI is available online:

https://link.springer.com/appendix/10.1007/978-981-13-1727-9_5

What is Perplexity?

Suppose that q and p are two probability distributions, and {x1, x2, . . . , xN} is a set of sample values that is drawn from a model whose probability distribution is p. In addition, suppose that b is a positive integer (it's usually equal to 2). Now define the variable S as the following sum (logarithms are in base b not 10):

S = (–1/N) * [log q(x1) + log q(x2) + · · · + log q(xN)]
= (–1/N) * SUM log q(xi)

The formula for the perplexity PERP of the model q is b raised to the power S, as shown here:

PERP = b^S

If you compare the formula for entropy with the formula for S, you can see that the formulas as similar, so the perplexity of a model is somewhat related to the entropy of a model.

CROSS-ENTROPY AND KL DIVERGENCE

Cross-entropy is useful for understanding machine learning algorithms, and frameworks such as TensorFlow, which supports multiple APIs that involve cross entropy. KL divergence is relevant in machine learning, deep learning, and reinforcement learning.

As an example, consider the credit assignment problem, which involves assigning credit to different elements or steps in a sequence. For example,

suppose that users arrive at a Web page by clicking on a previous page, which was also reached by clicking on yet another Web page. Then on the final Web page users click on an ad. How much credit is given to the first and second Web pages for the selected ad? You might be surprised to discover that one solution to this problem involves KL Divergence.

What is Cross Entropy?

The following formulas for logarithms are presented here because they are useful for the derivation of cross entropy in this section:

- log (a * b) = log a + log b
- log (a/b) = log a – log b
- log (1/b) = (–1) * log b

In a previous section you learned that for a probability distribution P with values {p1, p2, . . . , pn}, its entropy is H defined as follows:

H(P) = (–1)*SUM pi*log(pi)

Now let's introduce another probability distribution Q whose values are {q1, q2, . . . , qn}, which means that the entropy H of Q is defined as follows:

H(Q) = (–1)*SUM qi*log(qi)

Now we can define the cross entropy CE of Q and P as follows (notice the log qi and log pi terms and recall the formulas for logarithms in the previous section):

CE(Q, P) = SUM (pi*log **qi**) – SUM (pi*log **pi**)
= SUM (pi*log **qi** – pi*log **pi**)
= SUM pi*(log **qi** – log **pi**)
= SUM pi*(log **qi/pi**)

What is KL Divergence?

Now that entropy and cross-entropy have been discussed, we can easily define the KL Divergence of the probability distributions Q and P as follows:

KL(P||Q) = CE(P, Q) – H(P)

The definitions of entropy H, cross entropy CE, and KL Divergence in this appendix involve discrete probability distributions P and Q. However, these concepts have counterparts in continuous probability density functions. The mathematics involves the concept of a Lebesgue measure on Borel sets (which is beyond the scope of this book):

https://en.wikipedia.org/wiki/Lebesgue_measure
https://en.wikipedia.org/wiki/Borel_set

In addition to KL Divergence, there is also the JS Divergence (also called the Jenson-Shannon Divergence), which was developed by Johan Jensen and Claude Shannon (who defined the formula for entropy). The *JS Divergence* is based on the KL Divergence, and it has some differences. The JS Divergence is symmetric and a true metric, whereas the KL Divergence is neither. More information regarding JS Divergence is available online:

https://en.wikipedia.org/wiki/Jensen–Shannon_divergence

What's Their Purpose?

The Gini impurity is often used to obtain a measure of the homogeneity of a set of elements in a decision tree. The entropy of that set is an alternative to its Gini impurity, and you will see both of these quantities used in machine learning models.

The *perplexity* value in NLP is one way to evaluate language models, which are probability distributions over sentences or texts. This value provides an estimate for the encoding size of a set of sentences.

Cross entropy is used in various methods in the TensorFlow framework, and the KL Divergence is used in various algorithms, such as the dimensionality reduction algorithm t-SNE.

COVARIANCE AND CORRELATION MATRICES

This section explains two important matrices: the covariance matrix and the correlation matrix. Although these are relevant for PCA (Principal Component Analysis) that is discussed later in this appendix, these matrices are not specific to PCA, which is the rationale for discussing them in a separate section. If you are familiar with these matrices, feel free to skim through this section.

The Covariance Matrix

As a reminder, the statistical quantity called the variance of a random variable X is defined as follows:

variance(x) = [SUM (x – xbar)*(x – xbar)]/n

A covariance matrix C is an n × n matrix whose values on the main diagonal are the variance of the variables X1, X2, . . . , Xn. The other values of C are the covariance values of each pair of variables Xi and Xj.

The formula for the covariance of the variables X and Y is a generalization of the variance of a variable, and the formula is shown here:

covariance(X, Y) = [SUM (x – xbar)*(y – ybar)]/n

Notice that you can reverse the order of the product of terms (multiplication is commutative), and therefore the covariance matrix C is a symmetric matrix:

covariance(X, Y) = covariance(Y, X)

Suppose that a CSV file contains four numeric features, all of which have been scaled appropriately, and let's call them x1, x2, x3, and x4. Then the covariance matrix C is a 4 × 4 square matrix that is defined with the following entries (pretend that there are outer brackets on the left side and the right side to indicate a matrix):

```
cov(x1, x1) cov(x1, x2) cov(x1, x3) cov(x1, x4)
cov(x2, x1) cov(x2, x2) cov(x2, x3) cov(x2, x4)
cov(x3, x1) cov(x3, x2) cov(x3, x3) cov(x3, x4)
cov(x4, x1) cov(x4, x2) cov(x4, x3) cov(x4, x4)
```

Note that the following is true for the diagonal entries in the preceding covariance matrix C:

```
var(x1, x1) = cov(x1, x1)
var(x2, x2) = cov(x2, x2)
var(x3, x3) = cov(x3, x3)
var(x4, x4) = cov(x4, x4)
```

In addition, C is a symmetric matrix, which is to say that the transpose of matrix C (rows become columns and columns become rows) is identical to the matrix C. The latter is true because (as you saw in the previous section) cov(x, y) = cov(y, x) for any feature x and any feature y.

Covariance Matrix: an Example

Suppose we have the two-column matrix A defined as follows:

```
      x  y
A = | 1  1 | <= 6 × 2 matrix
    | 2  1 |
    | 3  2 |
    | 4  2 |
    | 5  3 |
    | 6  3 |
```

The mean x_bar of column x is (1 + 2 + 3 + 4 + 5 + 6)/6 = 3.5, and the mean y_bar of column y is (1 + 1 + 2 + 2 + 3 + 3)/6 = 2. Now subtract x_bar from column x and subtract y_bar from column y and we get matrix B, as shown here:

```
B = | –2.5  –1 | <= 6 × 2 matrix
    | –1.5  –1 |
    | –0.5   0 |
    |  0.5   0 |
    |  1.5   1 |
    |  2.5   1 |
```

Let Bt indicate the transpose of the matrix B (i.e., switch columns with rows and rows with columns) which means that Bt is a 2 × 6 matrix, as shown here:

```
Bt = |-2.5 -1.5 -0.5 0.5, 1.5, 2.5|
     |-1   -1    0   0    1    1  |
```

The covariance matrix C is the product of Bt and B, as shown here:

```
C = Bt * B = | 15.25 4 |
             | 4     8 |
```

Note that if the units of measure of features x and y do not have a similar scale, then the covariance matrix is adversely affected. In this case, the solution is simple: use the correlation matrix, which defined in the next section.

The Correlation Matrix

As you learned in the preceding section, if the units of measure of features x and y do not have a similar scale, then the covariance matrix is adversely affected. The solution involves the correlation matrix, which equals the covariance values cov(x, y) divided by the standard deviation stdx and stdy of x and y, respectively, as shown here:

```
corr(x, y) = cov(x, y)/[stdx * stdy]
```

The correlation matrix no longer has units of measure, and we can use this matrix to find the eigenvalues and eigenvectors.

Now that you understand how to calculate the covariance matrix and the correlation matrix, you are ready for an example of calculating eigenvalues and eigenvectors, which are the topic of the next section.

Eigenvalues and Eigenvectors

According to a well-known theorem in mathematics (whose proof you can find online), the eigenvalues of a symmetric matrix are real numbers. Consequently, the eigenvectors of C are vectors in a Euclidean vector space (not a complex vector space).

Before we continue, a non-zero vector x' is an eigenvector of the matrix C if there is a non-zero scalar lambda such that C*x' = lambda * x'.

Now suppose that the eigenvalues of C are b1, b2, b3, and b4, in decreasing numeric order from left-to-right, and that the corresponding eigenvectors of C are the vectors w1, w2, w3, and w4. Then, the matrix M that consists of the column vectors w1, w2, w3, and w4 represents the principal components.

CALCULATING EIGENVECTORS: A SIMPLE EXAMPLE

As a simple illustration of calculating eigenvalues and eigenvectors, suppose that the square matrix C is defined as follows:

C = | 1 3 |
      | 3 1 |

Let I denote the 2 × 2 identity matrix, and let b' be an eigenvalue of C, which means that there is an eigenvector x' such that

C* x' = b' * x', or
(C−b*I)*x' = 0 (the right side is a 2 × 1 vector)

Since x' is non-zero, that means the following is true (where det refers to the determinant of a matrix):

det(C−b*I) = det |1−b 3 | = (1−b)*(1−b)−9 = 0
                          |3 1−b|

We can expand the quadratic equation in the preceding line to obtain the following:

det(C−b*I) = (1−b)*(1−b) − 9
                  = 1 − 2*b + b*b − 9
                  = −8 − 2*b + b*b
                  = b*b − 2*b − 8

Use the quadratic formula (or perform factorization by visual inspection) to determine that the solution for det(C−b*I) = 0 is b = −2 or b = 4. Next, substitute b = −2 into (C−b*I)x' = 0 and we obtain the following result:

|1−(−2) 3 | |x1| = |0|
|3   1−(−2)| |x2| |0|

The preceding reduces to the following identical equations:

3*x1 + 3*x2 = 0
3*x1 + 3*x2 = 0

The general solution is x1 = −x2, and we can choose any non-zero value for x2, so let's set x2 = 1 (any non-zero value will do just fine), which yields x1 = −1. Therefore, the eigenvector [−1, 1] is associated with the eigenvalue −2. In a similar fashion, if x' is an eigenvector whose eigenvalue is 4, then [1, 1] is an eigenvector.

Notice that the eigenvectors [−1, 1] and [1, 1] are orthogonal because their inner product is zero, as shown here:

[−1, 1] * [1, 1] = (−1)*1 + (1)*1 = 0

In fact, the set of eigenvectors of a square matrix (whose eigenvalues are real) are always orthogonal, regardless of the dimensionality of the matrix.

Gauss Jordan Elimination (optional)

This simple technique enables you to find the solution to systems of linear equations "in place," which involves a sequence of arithmetic operations to transform a given matrix to an identity matrix.

The following example combines the Gauss-Jordan elimination technique (which finds the solution to a set of linear equations) with the "bookkeeper's method," which determines the inverse of an invertible matrix (its determinant is non-zero).

This technique involves two adjacent matrices: the left-side matrix is the initial matrix and the right-side matrix is an identity matrix. Next, perform various linear operations on the left-side matrix to reduce it to an identity matrix: the matrix on the right side equals its inverse. For example, consider the following pair of linear equations whose solution is x = 1 and y = 2:

```
2*x + 2*y = 6
4*x – 1*y = 2
```

Step 1: Create a 2 × 2 matrix with the coefficients of x in column 1 and the coefficients of y in column two, followed by the 2 × 2 identity matrix, and a column from the numbers on the right of the equals sign:

```
|2  2| 1 0 |6|
|4 –1| 0 1 |2|
```

Step 2: Add (–2) times the first row to the second row:

```
|2  2|  1 0 |6    |
|0 –5| –2 1 |–10|
```

Step 3: Divide the second row by 5:

```
|2  2|    1    0  |   6   |
|0 –1| –2/5 1/5 |–10/5|
```

Step 4: Add 2 times the second row to the first row:

```
|2  0|   1/5 2/5 | 2 |
|0 –1| –2/5 1/5 |–2|
```

Step 5: Divide the first row by 2:

```
|1  0| –2/10 2/10 | 1 |
|0 –1|  –2/5   1/5  |–2|
```

Step 6: Multiply the second row by (–1):

```
|1 0| –2/10 2/10 |1|
|0 1|    2/5  –1/5 |2|
```

As you can see, the left-side matrix is the 2 × 2 identity matrix, the right-side matrix is the inverse of the original matrix, and the right-most column is the solution to the original pair of linear equations (x = 1 and y = 2).

PCA (PRINCIPAL COMPONENT ANALYSIS)

PCA is a linear dimensionality reduction technique for determining the most important features in a dataset. This section discusses PCA because it's a very popular technique that you will encounter frequently. Other techniques are more efficient than PCA, so it's worthwhile to learn other dimensionality reduction techniques as well.

Keep in mind the following points regarding the PCA technique:

- PCA is a variance-based algorithm.
- PCA creates variables that are linear combinations of the original variables.
- The new variables are all pair-wise orthogonal.
- PCA can be a useful pre-processing step before clustering.
- PCA is generally preferred for data reduction.

PCA can be useful for variables that are strongly correlated. If most of the coefficients in the correlation matrix are smaller than 0.3, PCA is not helpful. PCA provides some advantages: less computation time for training a model (for example, using only five features instead of 100 features), a simpler model, and the ability to render the data visually when two or three features are selected. *PCA calculates the eigenvalues and the eigenvectors of the covariance (or correlation) matrix C.*

If you have four or five components, you won't be able to display them visually, but you could select subsets of three components for visualization, and perhaps gain some additional insight into the dataset.

The PCA algorithm involves the following sequence of steps:

1. Calculate the correlation matrix (from the covariance matrix) C of a dataset.
2. Find the eigenvalues of C.
3. Find the eigenvectors of C.
4. Construct a new matrix that comprises the eigenvectors.

The covariance matrix and correlation matrix were explained in a previous section. You also saw the definition of eigenvalues and eigenvectors, along with an example of calculating eigenvalues and eigenvectors.

The eigenvectors are treated as column vectors that are placed adjacent to each other in decreasing order (from left-to-right) with respect to their associated eigenvectors.

PCA uses the variance as a measure of information: the higher the variance, the more important the component. In fact, PCA determines the eigenvalues and eigenvectors of a covariance matrix (discussed in a previous section), and constructs a new matrix whose columns are eigenvectors, ordered from left-to-right in a sequence that matches the corresponding sequence of eigenvalues.

The left-most eigenvector has the largest eigenvalue, and the next eigenvector has the second-largest eigenvalue. This process continues until it reaches the right-most eigenvector (which has the smallest eigenvalue).

Alternatively, there is an interesting theorem in linear algebra: if C is a symmetric matrix, then there is a diagonal matrix D and an orthogonal matrix P (the columns are pair-wise orthogonal, which means their pair-wise inner product is zero), such that the following is true:

C = P * D * Pt (where Pt is the transpose of matrix P)

In fact, the diagonal values of D are eigenvalues, and the columns of P are the corresponding eigenvectors of the matrix C.

Fortunately, we can use NumPy and Pandas to calculate the mean, standard deviation, covariance matrix, correlation matrix, as well as the matrices D and P in order to determine the eigenvalues and eigenvectors.

Any positive definite square matrix has real-valued eigenvectors, which also applies to the covariance matrix C because it is a real-valued symmetric matrix.

The New Matrix of Eigenvectors

The previous section described how the matrices D and P are determined. The left-most eigenvector of D has the largest eigenvalue, the next eigenvector has the second-largest eigenvalue, and so forth. This fact is very convenient: the eigenvector with the highest eigenvalue is the principal component of the dataset. The eigenvector with the second-highest eigenvalue is the second principal component, and so forth. You specify the number of principal components that you want via the `n_components` hyper parameter in the `PCA` class of Sklearn.

As a simple and minimalistic example, consider the following code block that uses PCA for a (somewhat contrived) dataset:

```
import numpy as np
from sklearn.decomposition import PCA
data = np.array([[-1, -1], [-2, -1], [-3, -2], [1, 1], [2, 1], [3, 2]])
pca = PCA(n_components=2)
pca.fit(X)
```

Note that there is a trade-off here: we greatly reduce the number of components, which reduces the computation time and the complexity of the model, but we also lose some accuracy. However, if the unselected eigenvalues are small, we lose only a small amount of accuracy.

Now let's use the following notation:

- NM denotes the matrix with the new principal components.
- NMt is the transpose of NM.
- PC is the matrix of the subset of selected principal components.
- SD is the matrix of scaled data from the original dataset.
- SDt is the transpose of SD.

Then the matrix NM is calculated via the following formula:

```
NM = PCt * SDtz
```

Although PCA is a nice technique for dimensionality reduction, keep in mind the following limitations of PCA:

- less suitable for data with non-linear relationships
- less suitable for special classification problems

A related algorithm is called Kernel PCA, which is an extension of PCA that introduces a non-linear transformation so you can still use the PCA approach.

WELL-KNOWN DISTANCE METRICS

There are several similarity metrics available, such as item similarity metrics, Jaccard (user-based) similarity, and cosine similarity (which is used to compare vectors of numbers). The following subsections introduce you to these similarity metrics.

Another well-known distance metric is the *taxicab metric*, which is also called the *Manhattan distance metric*. Given two points A and B in a rectangular grid, the taxicab metric calculates the distance between two points by counting the number of “blocks” that must be traversed in order to reach B from A (the other direction has the same taxicab metric value). For example, if you need to travel two blocks north and then three blocks east in a rectangular grid, then the Manhattan distance is 5.

There are various other metrics available, which you can learn about by searching Wikipedia. In the case of NLP, the most commonly used distance metric is calculated via the cosine similarity of two vectors, and it’s derived from the formula for the inner (“dot”) product of two vectors.

Pearson Correlation Coefficient

The *Pearson similarity* is the Pearson coefficient between two vectors. Given random variables X and Y, and the following terms:

std(X) = standard deviation of X
std(Y) = standard deviation of Y
cov(X, Y) = covariance of X and Y

Then the Pearson correlation coefficient rho(X, Y) is defined as follows:

$$\text{rho}(X, Y) = \frac{\text{cov}(X, Y)}{\text{std}(X)*\text{std}(Y)}$$

The Pearson coefficient is limited to items of the same type. More information about the Pearson correlation coefficient is available online:

https://en.wikipedia.org/wiki/Pearson_correlation_coefficient

Jaccard Index (or Similarity)

The *Jaccard similarity* is based on the number of users which have rated item A and B divided by the number of users who have rated either A or B. The Jaccard similarity is based on unique words in a sentence and is unaffected by duplicates, whereas cosine similarity is based on the length of all word vectors (which changes when duplicates are added). The choice between the cosine similarity and Jaccard similarity depends on whether word duplicates are important.

The following Python method illustrates how to compute the Jaccard similarity of two sentences.

```
def get_jaccard_sim(str1, str2):
  set1 = set(str1.split())
  set2 = set(str2.split())
  set3 = set1.intersection(set2)
  # (size of intersection) / (size of union):
  return float(len(set3)) / (len(set1) + len(set2) - len(set3))
```

The Jaccard similarity can be used in situations involving Boolean values, such as product purchases (true/false), instead of numeric values. More information is available online:

https://en.wikipedia.org/wiki/Jaccard_index

Local Sensitivity Hashing (optional)

If you are familiar with hash algorithms, you know that they are algorithms that create a hash table that associate items with a value. The advantage of hash tables is that the lookup time to determine whether an item exists in the hash table is constant.

Of course, it's possible for two items to *collide*, which means that they both occupy the same bucket in the hash table. In this case, a bucket can consist of a list of items that can be searched in more or less a constant amount of time. If there are too many items in the same bucket, then a different hashing function can be selected to reduce the number of collisions. The goal of a hash table is to minimize the number of collisions.

The Local Sensitivity Hashing (LSH) algorithm hashes similar input items into the same "buckets." In fact, the goal of LSH is to maximize the number of collisions, whereas traditional hashing algorithms attempt to minimize the number of collisions.

Since similar items end up in the same buckets, LSH is useful for data clustering and nearest neighbor searches. Moreover, LSH is a dimensionality reduction technique that places data points of high dimensionality closer together in a lower-dimensional space, while simultaneously preserving the relative distances between those data points. More details about LSH are available online:

https://en.wikipedia.org/wiki/Locality-sensitive_hashing

TYPES OF DISTANCE METRICS

Non-linear dimensionality reduction techniques can also have different distance metrics. For example, linear reduction techniques can use the Euclidean distance metric (based on the Pythagorean theorem).

However, you need to use a different distance metric to measure the distance between two points on a sphere (or some other curved surface). In the case of NLP, the cosine similarity metric is often used to measure the distance between word embeddings (which are vectors of floating point numbers that represent words or tokens).

Distance metrics are used for measuring physical distances, and some well-known distance metrics are listed here:

- Euclidean distance
- Manhattan distance
- Chebyshev distance

The Euclidean algorithm also obeys the *triangle inequality*, which states that for any triangle in the Euclidean plane, the sum of the lengths of any pair of sides must be greater than the length of the third side.

In spherical geometry, you can define the distance between two points as the arc of a great circle that passes through the two points (always selecting the smaller of the two arcs when they are different).

In addition to physical metrics, there are algorithms that implement the concept of the *edit distance* (the distance between strings), as listed here:

- Hamming distance
- Jaro–Winkler distance
- Lee distance
- Levenshtein distance
- Mahalanobis distance metric
- Wasserstein metric

The Mahalanobis metric is based on an interesting idea: given a point P and a probability distribution D, this metric measures the number of standard deviations that separate point P from distribution D. More information about Mahalanobis is available online:

https://en.wikipedia.org/wiki/Mahalanobis_distance

In the branch of mathematics called topology, a metric space is a set for which distances between all members of the set are defined. Various metrics are available (such as the Hausdorff metric), depending on the type of topology.

The Wasserstein metric measures the distance between two probability distributions over a metric space X. This metric is also called the "earth mover's metric" for the following reason: given two unit piles of dirt, it's the measure of the minimum cost of moving one pile on top of the other pile.

The KL Divergence bears some superficial resemblance to the Wasserstein metric. However, there are some important differences between them. Specifically, the Wasserstein metric has the following properties:

1. It is a metric.
2. It is symmetric.
3. It satisfies the triangle inequality.

The KL Divergence has the following properties:

1. It is not a metric (it's a divergence).
2. It is not symmetric: KL(P, Q) != KL(Q, P).
3. It does not satisfy the triangle inequality.

Note that the JS (Jenson-Shannon) Divergence (which is based on the KL Divergence) is a true metric, which would enable a more meaningful comparison with other metrics (such as the Wasserstein metric).

https://stats.stackexchange.com/questions/295617/what-is-the-advantages-of-wasserstein-metric-compared-to-kullback-leibler-diverg

More information is available online:

https://en.wikipedia.org/wiki/Wasserstein_metric

WHAT IS BAYESIAN INFERENCE?

Bayesian inference is an important technique in statistics that involves statistical inference and Bayes' theorem to update the probability for a hypothesis as more information becomes available. *Bayesian inference* is often called Bayesian probability, and it's important in dynamic analysis of sequential data.

Bayes' Theorem

Given two sets A and B, let's define the following numeric values (all of them are between 0 and 1):

P(A) = probability of being in set A
P(B) = probability of being in set B
P(Both) = probability of being in A intersect B
P(A|B) = probability of being in A (given you're in B)
P(B|A) = probability of being in B (given you're in A)

Then the following formulas are also true:

P(A|B) = P(Both)/P(B) (#1)
P(B|A) = P(Both)/P(A) (#2)

Multiply the preceding pair of equations by the term that appears in the denominator to obtain these equations:

P(B)*P(A|B) = P(Both) (#3)
P(A)*P(B|A) = P(Both) (#4)

Now set the left-side of equations #3 and #4 equal to each another and that gives us this equation:

P(B)*P(A|B) = P(A)*P(B|A) (#5)

Divide both sides of #5 by P(B) to obtain this well-known equation:

P(A|B) = P(A)*P(A|B)/P(B) (#6)

Some Bayesian Terminology

In the previous section we derived the following relationship:

P(h|d) = (P(d|h) * P(h)) / P(d)

There is a name for each of the four terms in the preceding equation, as discussed below.

First, the posterior probability is P(h|d), which is the probability of hypothesis `h` given the data `d`.

Second, P(d|h) is the probability of data d given that the hypothesis h was true.

Third, the *prior probability* of h is P(h), which is the probability of hypothesis h being true (regardless of the data).

Finally, P(d) is the probability of the data (regardless of the hypothesis)

We are interested in calculating the posterior probability of P(h|d) from the prior probability p(h) with P(D) and P(d|h).

What is MAP?

The maximum a posteriori (MAP) hypothesis is the hypothesis with the highest probability, which is the maximum probable hypothesis. This can be written as follows:

MAP(h) = max(P(h|d))
or
MAP(h) = max((P(d|h) * P(h)) / P(d))
or
MAP(h) = max(P(d|h) * P(h))

Why Use Bayes' Theorem?

Bayes' Theorem describes the probability of an event based on the prior knowledge of the conditions that might be related to the event. If we know the conditional probability, we can use Bayes rule to find out the reverse probabilities. The previous statement is the general representation of the Bayes rule.

SUMMARY

This appendix started with a discussion of probability, expected values, and the concept of a random variable. Then you learned about some basic statistical concepts, such as mean, median, mode, variance, and standard deviation. Next, you learned about the terms RSS, TSS, R^2, and F1 score. In addition, you were introduced to the concepts of skewness, kurtosis, Gini Impurity, Entropy, Perplexity, Cross-Entropy, and KL Divergence.

Next, you learned about covariance and correlation matrices and how to calculate eigenvalues and eigenvectors. Then you were introduced to the dimensionality reduction technique known as PCA (Principal Component Analysis), after which you learned about Bayes' Theorem.

INDEX

A
A Lite BERT (ALBERT), 199
AlphaFold, 212
Attention mechanism
  algorithms, 181
  description, 180
  formulas, 181
  types, 181
  word embedding types, 180–181

B
Bag of Words (BoW) algorithm, 99
  advantages of, 123
  `CountVectorizer` class, 123
  word/index pairs, 124
Bayesian inference, 238–239
Bayes' Theorem, 238–239
BERT model
  ALBERT, 199
  deBERTa, 200
  description, 187
  DistilBERT, 199
  features, 187
  MLM, 188
  vs. NLP models, 188
  NSP, 188–189
  RoBERTa, 200
  sentence similarity, 194
  sequence of steps, 190–192
  SMITH model, 200
  special tokens, 189–190
  tokens, 196–198
  training, 187
  word vector, 194–196
Bilingual Evaluation Understudy (BLEU)
    score, 153
bm25 algorithm, 131
Brown Corpus, 94
Byte-pair encoding (BPE), 192

C
CBoW architecture, 140–141
Chunking, 110
Context for words
  contextual word representations, 133
  discrete text representations, 133
  distributed text representations, 133
  semantic context, 132
  textual entailment, 133
Contextual word representations, 133
Correlation matrix, 230
Cosine similarity, 133–135
Covariance matrix, 228–230
Cross-entropy, 227

D
DALL-E, 212
Data frames in R
  add a new attribute, 43–45
  append a new row, 45
  column-related and row-related
    operations, 39

display portions of, 42–43
with heterogeneous values, 41
variable `mydf`, 40
work with, 46–47
deBERTa, 200
Decoding task, 108
Deep Learning Methods, POS, 109
Digram coding. *See* Byte-pair encoding (BPE)
Discrete text representations, 133
Distance metrics, 237–238
DistilBERT, 199
Distributed text representations, 133
Document classification, 97
Document similarity, 118

E
Eigenvalues and eigenvectors, 230
Emission probabilities, 108
Entropy, 224–226

F
FastText library, 146
Functions
apply family, 70–72
custom function, 85–86
file-related functions, 68
`gsub()` function, 67–68
math-related functions and trigonometric functions, 65–66
miscellaneous functions, 68
`pipe` operator, 75–77
recursive function, 86
set functions, 69–70
string-related functions, 66–67
summary functions, 84–85

G
Gauss-Jordan elimination technique, 232
Generative Pre-Training (GPT) transformers
GPT-2, 201–207
vs. BERT, 207
GPT-3
AlphaFold, 212
architecture, 208
benefits, 208
DALL-E, 212
few-shot learner, 210
goals of, 209–210
one-shot learner, 210
Playground, 208–209
strengths and mistakes, 208
switch transformer, 211
task performance, 210–211
zero-shot learner, 210
installation process, 201
versions, 200–201
Gini impurity, 224
Global Matrix Factorization (GMF), 145
GloVe, 144–145
Grammatical ambiguity, 98
`gsub()` function, 67–68

H
HuggingFace transformer, 182–183
mask-filling task, 185–186
NER task, 183–184
question-and-answer task, 184
sentiment analysis task, 185

I
Information extraction (IE), 152
Inverse Document Frequency (IDF), 128–129
ISRIStemmer and RSLPSStemmer, 105

J
Jaccard similarity, 236

K
KL Divergence, 227–228
Kurtosis, 220

L
Lancaster stemmer, 106
Latent Dirichlet Allocation (LDA) algorithm, 111
Lexical ambiguity, 98
Lexical Based Methods, POS, 109
LexRank algorithm, 112
Local context window (LCW), 145
Local Sensitivity Hashing (LSH) algorithm, 236
Loops
for, 31–32
compound conditional logic, 34
conditional logic, 33–34
nested loop, 32
while loops, 32–33

M

Manhattan distance metric. *See* Taxicab metric
Masked language model (MLM), 188
Math-related functions and trigonometric functions, 65–66
Maximum a posteriori (MAP) hypothesis, 239
Multi-dimensional Gini index (MGI), 226

N

Named Entity Recognition (NER), 102, 170–171
  abbreviations and acronyms, 110
  challenges, 110
  deep learning techniques, 111
  feature-based supervised learning, 111
  incorrect results, 110
  rule-based techniques, 111
  types, 109
  unsupervised learning techniques, 111
NaN values, 63–64
Natural Language Generation (NLG), 98
Natural Language Processing (NLP)
  applications and use cases, 96–97
  challenges, 94
  corpus, 94
  data cleaning
    convert to lowercase, 162–164
    lemmatization, 165–167
    remove punctuation in strings, 161–162
    stemming, 165
    stop words, 164–165
    tokenization, 161
  document classification, 117
  document similarity, 118
  evolution of, 95–96
  information extraction and retrieval, 99
  keyword extraction, 112
  language models and
    challenges, 149
    creation of, 149–150
  lemmatization
    limitations, 107–108
    warnings, 107
  M2M model, 94
  named entity recognition (NER)
    abbreviations and acronyms, 110
    challenges, 110
    deep learning techniques, 111
    feature-based supervised learning, 111
    incorrect results, 110
    rule-based techniques, 111
    types, 109
    unsupervised learning techniques, 111
  neural networks, 94
  NLU and NLG, 97–98
  parts of speech (POS)
    challenges of, 108
    HMMs, 108
    purpose of, 108
    tagging, 108–109
  rule-based approaches, 94
  sentence similarity, 117
    sentence encoders, 117
  sentiment analysis, 112
  stemmers
    ISRIStemmer and RSLPSStemmer, 105
    Lancaster stemmer, 106
    limitations, 107–108
    over stemming, 106–107
    Porter stemmer, 105
    singular *vs*. plural word endings, 105
    Snowball stemmer, 105
    under stemming, 107
    warnings, 107
    word prefixes, 106
  steps for training a model, 101
  stop words, 104–105
  text classification, 98–99
  text encoding
    BoW and n-grams, 119–120
    document vectorization, 120–121
    index-based encoding, 122
    OHE, 121–122
    other encoding techniques, 122–123
    tf-idf, 120
  text mining, 152
  text normalization, 101
  text similarity, 116–117
    similarity query, 119
    tf-idf, 118
  text summarization, 112
  tokenization
    in Japanese, 102–103
    rule-based tokenization, 103
    with Unix commands, 104
  topic modeling, 95, 111–112
  traditional machine learning, 94

transformer architecture, 94
types and techniques, 100
word relevance, 115–116
word sense disambiguation, 99–100
Natural Language Understanding (NLU), 97
Next sentence prediction (NSP), 188–189
n-grams, 152
with bi-gram, 174–176
2-grams and 3-grams, 125
probability calculation, 125–127
types, 124
NLTK tokenizer and SpaCY tokenizer, 103

O
OpenNLP, 159
Out of vocabulary (OOV), 192

P
Parts of speech (POS)
challenges of, 108
HMMs, 108
purpose of, 108
`spacy_parse()`, 168
tagging, 108–109
in a text string, 167–170
Pearson correlation coefficient, 235
Perplexity, 226
Pointwise Mutual Information (PMI), 132
Porter stemmer, 105
Positive PMI (PPMI), 132
Principal component analysis (PCA), 233–235
Probabilistic Methods, POS, 109
Probability
conditional probability, 214
description, 213–214
distributions, 216
expected value calculation, 214–215
random variable, 215–216
discrete *vs*. continuous, 216

Q
Quanteda, 159

R
`R`
assign values to variables, 3
built-in chart-related functions
bar charts, 52–53
box plots, 59–60
histograms, 56–57
line graphs, 53–55
multi-line graphs, 55–56
pie charts, 60–62
scatter plots, 57–59
check for leap years, 37
`convert_to_binary()` function, 90–91
CSV files, 77–78
custom function, 85–86
data frames
add a new attribute, 43–45
append a new row, 45
column-related and row-related operations, 39
display portions of, 42–43
with heterogeneous values, 41
variable `mydf`, 40
work with, 46–47
data types, 3–4
dates, 27–28
`dplyr` package, 72–74
factorial value calculation
with recursion, 87–88
without recursion, 87
factors in, 38–39
features, 2
Fibonacci number calculation
with recursion, 89–90
without recursion, 88–89
GCD of two integers, 91
installation, 2
JSON file, 81–82
LCM of two integers, 92
lists, 12–15
matrices, 16–27
named entity recognition, 170–171
NLP
data cleaning, 160–167
packages, 159–160
operators, 3
packages, 74–75
prime numbers
numbers in an array, 36–37
`PrimeNumber.R`, 35
reading excel files in, 47–48
reading sqlite tables in, 48–49
reading text files in, 49–50
recursion, 86
RStudio package installation, 158–159
save and restore objects, 50–51
scripts from the command line, 156–158

seq() function, 28–30
statistical functions, 83
strings
blank_count and non_blank initialization, 6
blank_count initialization, 6
number of digits and non-digits calculation, 6
print() statement, 4
string_tasks.R, 5–6
str_length2 initialization, 6
uppercase and lowercase strings, 4–5
sum of angles, 37–38
variable names, 2
vector related functions, 15–16
vectors
built-in variable letters, 11
finding NULL values, 9
sorting, 10–11
updating NA values, 10
VectorStuff.R, 7
VectorStuff2.R, 8
XML files, 78–80
read the contents of, 80–81
Referential ambiguity, 98
Relation extraction (RE), 97, 152
RoBERTa, 200
ROUGE score, 153
RStudio, 2
RStudio Cloud, 2
Rule-Based Methods, POS, 109

S

Semantic context, 132
Sentence embedding models, 117
SentencePiece, 194
Sentence similarity, 194
Sentiment analysis, 96
Set functions, 69–70
Skewness, 220
Skip-grams
architecture, 142
backward error propagation, 144
high-level description, 141
neural network reduction, 144
shallow network, 143
SMITH model, 200
Snowball stemmer, 105
Spacyr, 159
Statistics
Central Limit Theorem, 221
Chebyshev's inequality, 219
correlation *vs*. causation, 221–222
F1 score, 223
mean, 217
median, 217
mode, 217–218
moments of a function, 219–220
population, sample, and population variance, 218–219
p-values, 219
RSS, TSS and R^2, 222–223
statistical inferences, 222
variance and standard deviation, 218
Stemming
ISRIStemmer and RSLPSStemmer, 105
Lancaster stemmer, 106
limitations, 107–108
over stemming, 106–107
Porter stemmer, 105
singular *vs*. plural word endings, 105
Snowball stemmer, 105
under stemming, 107
warnings, 107
word prefixes, 106
Stop words, 104–105
Stringr, 159
String-related functions, 66–67
Subword tokenization
byte-pair encoding, 192
SentencePiece, 194
unigram language model, 194
Wordpiece, 193
Syntactical ambiguity, 98

T

Taxicab metric, 235
Teacher forcing, 186
Term frequency (TF), 127–128
Text classification, 98–99
Text encoding
BoW and n-grams, 119–120
document vectorization, 120–121
index-based encoding, 122
OHE, 121–122
other encoding techniques, 122–123
tf-idf, 120
TextRank algorithm, 112
Text-To-Text Transfer Transformer (T5), 186–187
Textual entailment, 133
Text2vec, 159

Text vectorization, 135–136
tf-idf algorithm, 129, 171–174
  vs. BoW, 130
  limitations of, 130–131
Topic modeling, 95, 111–112, 147, 176–177
  LDA, 147–148
  vs. text classification, 149
Transformer architecture
  encoder and decoder component, 182
  HuggingFace, 182–183
    mask-filling task, 185–186
    NER task, 183–184
    question-and-answer task, 184
    sentiment analysis task, 185
Transition probabilities, 108

V
Vector space model (VSM), 150
  advantages and disadvantages of, 151
  term-document matrix, 151

W
Wordcloud, 160
Word embeddings
  comparison of, 146–147
  definition, 137
  document classification and
    clustering, 137
  fastText algorithm, 138
  GloVe algorithm, 138
  word2vec algorithm, 137
Wordpiece, 193
Word sense disambiguation, 99–100
Word2vec, 177–178
  architecture, 140
  CBoW and skip-grams, 139
  cosine similarity, 138
  limitations of, 140
  making predictions, 138
  use cases, 139
Word vectorization, 117