Natural Language Processing Fundamentals for Developers

LICENSE, DISCLAIMER OF LIABILITY, AND LIMITED WARRANTY

By purchasing or using this book and its companion files (the "Work"), you agree that this license grants permission to use the contents contained herein, but does not give you the right of ownership to any of the textual content in the book or ownership to any of the information, files, 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, production, 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 insure 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 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 also available for downloading from the publisher by writing to info@merclearning.com.

Natural Language Processing Fundamentals for Developers

Oswald Campesato

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

Publisher: David Pallai
Mercury Learning and Information
22841 Quicksilver Drive
Dulles, VA 20166
info@merclearning.com
www.merclearning.com
1-800-232-0223

O. Campesato. *Natural Language Processing Fundamentals for Developers.*
ISBN: 978-1-68392-657-3

Library of Congress Control Number: 2021939603

212223321 Printed on acid-free paper in the United States of America.

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

CONTENTS

PREFACE

WHAT IS THE PRIMARY VALUE PROPOSITION FOR THIS BOOK?

This book contains a fast-paced introduction to as much relevant information about NLP as possible that can be reasonably included in a book of this size. Some chapters contain topics that are discussed in great detail (such as the first half of Chapter 2), and other chapters contain advanced statistical concepts that you can safely omit during your first pass through this book. This book casts a wide net to help 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 keep in mind the following point: *you will probably need to read some of the content in this book multiple times.*

However, you will be exposed to *many* NLP topics, and many topics 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 will decide whether or not 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 quadruple the size of this book, and few people are interested in reading 1,000-page technical books. Hence, this book provides a broad view of the NLP landscape, based on the belief that this approach will be more beneficial for readers who are experienced developers, who want to learn about NLP.

However, it's important for you to decide if this approach is suitable for your needs and learning style: if not, you can select one or more of the plethora of NLP books that are available.

THE TARGET AUDIENCE

This book is intended primarily for people who have a solid background as software developers. Specifically, it 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. While many readers know how to read English, their native spoken language is not English. Consequently, this book uses standard English rather than colloquial expressions that might be confusing to those readers. As you know, many people learn by different types of imitation, which includes reading, writing, or hearing new material. This book takes these points into consideration in order to provide a comfortable and meaningful learning experience for the intended readers.

WHY SUCH A MASSIVE NUMBER OF TOPICS IN THIS BOOK?

As mentioned in the response to the previous question, this book is intended for developers who want to learn NLP concepts. Because this encompasses people with vastly different technical backgrounds, there are readers who "don't know what they don't know" regarding NLP. Therefore, this book exposes people to a plethora of NLP-related concepts, after which they can decide those topics to select for greater study. Consequently, the book does *not* have a "zero-to-hero" approach, nor is it necessary to master all the topics that are discussed in the chapters and the appendices; rather, they are a go-to source of information to help you decide where you want to invest your time and effort.

As you might already know, learning often takes place through an iterative and repetitive approach whereby the cumulative exposure leads to a greater level of comfort and understanding of technical concepts. For some readers, this will be the first step in their journey toward mastering NLP.

HOW IS THE BOOK ORGANIZED AND WHAT WILL I LEARN?

The first chapter shows you various details of managing data that are relevant for NLP. The next pair of chapters contain NLP concepts, followed by another pair of chapters that contain Python code samples which illustrate the NLP concepts.

Chapter 6 explores sentiment analysis, recommender systems, COVID-19 analysis, spam detection, and a short discussion regarding chatbots. The final chapter presents the Transformer architecture, BERT-based models, and the GPT family of models, all of which have been developed during the past three years and to varying degrees they are considered SOTA ("state of the art").

The appendices contain introductory material (including Python code samples) for various topics, including Regular Expressions and statistical concepts.

WHY ARE THE CODE SAMPLES PRIMARILY IN PYTHON?

Most of the code samples are short (usually less than one page and sometimes less than half a page), and if need be, you can easily and quickly copy/paste the code into a new Jupyter notebook.

If you do decide to use Google Colaboratory, you can easily copy/paste the Python code into a notebook, and also use the upload feature to upload existing Jupyter notebooks. Keep in mind the following point: if the Python code references a CSV file, make sure that you include the appropriate code snippet (as explained in Chapter 1) to access the CSV file in the corresponding Jupyter notebook in Google Colaboratory.

HOW WERE THE CODE SAMPLES CREATED?

The code samples in this book were created and tested using Python 3 on a MacBook Pro with OS X 10.15.15 (macOS Catalina). Regarding their content: the code samples are derived primarily from the author for his Natural Language Processing graduate course. In some cases, there are code samples that incorporate short sections of code from discussions in online forums. The key point to remember is the code samples follow the "Four

Cs": they must be Clear, Concise, Complete, and Correct to the extent that it's possible to do so, given the size of this book.

GETTING THE MOST FROM THIS BOOK

Some programmers learn well from prose, 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" from earlier code samples.

WHAT DO I NEED TO KNOW FOR THIS BOOK?

Current knowledge of Python 3.x is the most helpful skill. 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 in order 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 primary purpose of the code samples in this book is to show you Python-based libraries for solving a variety of NLP-related tasks. Clarity has 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 ought to 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 a strong desire to learn about NLP, along with the motivation and discipline to read and understand the code samples.

Even simple APIs can be a challenge to understand them 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 *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, then 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
May 2021

CHAPTER 1

WORKING WITH DATA

This chapter introduces you to the data types (along with their differences), how to scale data values, and various techniques for handling missing data values. If most of the material in this chapter is new to you, be assured that it's not necessary to understand everything in this chapter. It's still a good idea to read as much material as you can, and perhaps return to this chapter again after you have completed some of the other chapters in this book.

The first part of this chapter contains an overview of different types of data and an explanation of how to normalize and standardize a set of numeric values by calculating the mean and standard deviation of a set of numbers. You will see how to map categorical data to a set of integers and how to perform one-hot encoding.

The second part of this chapter discusses missing data, outliers, and anomalies, and also some techniques for handling these scenarios. The third section discusses imbalanced data and the use of SMOTE (Synthetic Minority Oversampling Technique) to deal with imbalanced classes in a dataset.

The fourth section discusses ways to evaluate classifiers such as LIME and ANOVA. This section also contains details regarding the bias-variance trade-off and various types of statistical bias.

WHAT ARE DATASETS?

In simple terms, a dataset is a source of data (such as a text file) that contains rows and columns of data. Each row is typically called a "data point," and each column is called a "feature." A dataset can be in any form: CSV (comma separated values), TSV (tab separated values), Excel spreadsheet, a table in an RDMBS (Relational Database Management System), a document

in a NoSQL database, or the output from a Web service. Someone needs to analyze the dataset to determine which features are the most important and which features can be safely ignored in order to train a model with the given dataset.

A dataset can vary from very small (a couple of features and 100 rows) to very large (more than 1,000 features and more than one million rows). If you are unfamiliar with the problem domain, then you might struggle to determine the most important features in a large dataset. In this situation, you might need a domain expert who understands the importance of the features, their interdependencies (if any), and whether the data values for the features are valid. In addition, there are algorithms (called dimensionality reduction algorithms) that can help you determine the most important features. For example, PCA (Principal Component Analysis) is one such algorithm, which is discussed in more detail later in this chapter.

Data Preprocessing

Data preprocessing is the initial step that involves validating the contents of a dataset, which involves making decisions about missing and incorrect data values such as

- dealing with missing data values
- cleaning "noisy" text-based data
- removing HTML tags
- removing emoticons
- dealing with emojis/emoticons
- filtering data
- grouping data
- handling currency and date formats (i18n)

Cleaning data is an important initial task that involves removing unwanted data as well as handling missing data. In the case of text-based data, you might need to remove HTML tags, punctuation, and so forth. In the case of numeric data, it's less likely (though still possible) that alphabetic characters are mixed together with numeric data. However, a dataset with numeric features might have incorrect values or missing values (discussed later). In addition, calculating the minimum, maximum, mean, median, and standard deviation of the values of a feature obviously pertain only to numeric values.

After the preprocessing step is completed, *data wrangling* is performed, which refers to transforming data into a new format. You might have to combine data from multiple sources into a single dataset. For example, you might

need to convert between different units of measurement (such as date formats or currency values) so that the data values can be represented in a consistent manner in a dataset.

Currency and date values are part of *i18n* (internationalization), whereas *l10n* (localization) targets a specific nationality, language, or region. Hard-coded values (such as text strings) can be stored as resource strings in a file that's often called a *resource bundle*, where each string is referenced via a code. Each language has its own resource bundle.

DATA TYPES

Explicit data types exist in many programming languages such as C, C++, Java, and TypeScript. Some programming languages, such as JavaScript and awk, do not require initializing variables with an explicit type: the type of a variable is inferred dynamically via an implicit type system (i.e., one that is not directly exposed to a developer).

In machine learning, datasets can contain features that have different types of data, such as a combination of one or more of the following:

- numeric data (integer/floating point and discrete/continuous)
- character/categorical data (different languages)
- date-related data (different formats)
- currency data (different formats)
- binary data (yes/no, 0/1, and so forth)
- nominal data (multiple unrelated values)
- ordinal data (multiple and related values)

Consider a dataset that contains real estate data, which can have as many as thirty columns (or even more), often with the following features:

- the number of bedrooms in a house: numeric value and a discrete value
- the number of square feet: a numeric value and (probably) a continuous value
- the name of the city: character data
- the construction date: a date value
- the selling price: a currency value and probably a continuous value
- the "for sale" status: binary data (either "yes" or "no")

An example of nominal data is the seasons in a year: although many countries have four distinct seasons, some countries have only two distinct seasons.

However, seasons can be associated with different temperature ranges (summer versus winter). An example of ordinal data is an employee pay grade: 1=entry level, 2=one year of experience, and so forth. Another example of nominal data is a set of colors, such as {Red, Green, Blue}.

An example of binary data is the pair {Male, Female}, and some datasets contain a feature with these two values. If such a feature is required for training a model, first convert {Male, Female} to a numeric counterpart, such as {0, 1}. Similarly, if you need to include a feature whose values are the previous set of colors, you can replace {Red, Green, Blue} with the values {0, 1, 2}.

PREPARING DATASETS

If you have the good fortune to inherit a dataset that is in pristine condition, then data cleaning tasks (discussed later) are vastly simplified: in fact, it might not be necessary to perform *any* data cleaning for the dataset. On the other hand, if you need to create a dataset that combines data from multiple datasets that contain different formats for dates and currency, then you need to perform a conversion to a common format.

If you need to train a model that includes features that have categorical data, then you need to convert that categorical data to numeric data. For instance, the Titanic dataset contains a feature called "gender," which is either male or female. Later in this chapter, we show how to "map" male to 0 and female to 1 using Pandas.

Discrete Data Versus Continuous Data

As a simple rule of thumb: discrete data is a set of values that can be counted, whereas continuous data must be measured. Discrete data can reasonably fit in a drop-down list of values, but there is no exact value for making such a determination. One person might think that a list of 500 values is discrete, whereas another person might think it's continuous.

For example, the list of provinces of Canada and the list of states of the United States are discrete data values, but is the same true for the number of countries in the world (roughly 200) or for the number of languages in the world (more than 7,000)?

Values for temperature, humidity, and barometric pressure are considered continuous. Currency is also treated as continuous, even though there is a measurable difference between two consecutive values. The smallest

unit of currency for U.S. currency is one penny, which is 1/100th of a dollar (accounting-based measurements use the "mil," which is 1/1,000th of a dollar).

Continuous data types can have subtle differences. For example, someone who is 200 centimeters tall is twice as tall as someone who is 100 centimeters tall; the same is true for 100 kilograms versus 50 kilograms. However, temperature is different: 80 degrees Fahrenheit is not twice as hot as 40 degrees Fahrenheit.

Furthermore, keep in mind that the meaning of the word "continuous" in mathematics is not necessarily the same as continuous in machine learning. In the former, a continuous variable (let's say in the 2D Euclidean plane) can have an uncountably infinite number of values. A feature in a dataset that can have more values than can be reasonably displayed in a drop-down list is treated *as though* it's a continuous variable.

For instance, values for stock prices are discrete: they must differ by at least a penny (or some other minimal unit of currency), which is to say, it's meaningless to say that the stock price changes by one-millionth of a penny. However, since there are so many possible stock values, it's treated as a continuous variable. The same comments apply to car mileage, ambient temperature, and barometric pressure.

"Binning" Continuous Data

Binning refers to subdividing a set of values into multiple intervals, and then treating all the numbers in the same interval as though they had the same value.

As a simple example, suppose that a feature in a dataset contains the age of people in a dataset. The range of values is approximately between 0 and 120, and we could bin them into 12 equal intervals, where each consists of 10 values: 0 through 9, 10 through 19, 20 through 29, and so forth.

However, partitioning the values of people's ages as described in the preceding paragraph can be problematic. Suppose that person A, person B, and person C are 29, 30, and 39, respectively. Then person A and person B are probably more similar to each other than person B and person C, but because of the way in which the ages are partitioned, B is classified as closer to C than to A. In fact, binning can increase Type I errors (false positive) and Type II errors (false negative), as discussed in this blog post (along with some alternatives to binning):

https://medium.com/@peterflom/why-binning-continuous-data-is-almost-always-a-mistake-ad0b3a1d141f.

As another example, using quartiles is even more coarse-grained than the earlier age-related binning example. The issue with binning pertains to the consequences of classifying people in different bins, even though they are in close proximity to each other. For instance, some people struggle financially because they earn a meager wage, and they are disqualified from financial assistance because their salary is higher than the cutoff point for receiving any assistance.

Scaling Numeric Data via Normalization

A range of values can vary significantly, and it's important to note that they often need to be scaled to a smaller range, such as values in the range [–1, 1] or [0, 1], which you can do via the `tanh` function or the `sigmoid` function, respectively.

For example, measuring a person's height in terms of meters involves a range of values between 0.50 meters and 2.5 meters (in the vast majority of cases), whereas measuring height in terms of centimeters ranges between 50 centimeters and 250 centimeters: these two units differ by a factor of 100. A person's weight in kilograms generally varies between 5 kilograms and 200 kilograms, whereas measuring weight in grams differs by a factor of 1,000. Distances between objects can be measured in meters or in kilometers, which also differ by a factor of 1,000.

In general, use units of measure so that the data values in multiple features belong to a similar range of values. In fact, some machine learning algorithms require scaled data, often in the range of [0, 1] or [–1, 1]. In addition to the `tanh` and `sigmoid` function, there are other techniques for scaling data, such as standardizing data (think Gaussian distribution) and normalizing data (linearly scaled so that the new range of values is in [0, 1]).

The following examples involve a floating point variable `X` with different ranges of values that will be scaled so that the new values are in the interval [0, 1].

- Example 1: If the values of `X` are in the range [0, 2], then `X/2` is in the range [0, 1].
- Example 2: If the values of `X` are in the range [3, 6], then `X – 3` is in the range [0, 3], and `(X – 3)/3` is in the range [0, 1].
- Example 3: If the values of `X` are in the range [–10, 20], then `X + 10` is in the range [0, 30], and `(X + 10)/30` is in the range of [0, 1].

In general, suppose that `X` is a random variable whose values are in the range `[a, b]`, where a < b. You can scale the data values by performing two steps:

```
Step 1: X-a is in the range [0,b-a]
Step 2: (X-a)/(b-a) is in the range [0,1]
```

If X is a random variable that has the values {x1, x2, x3, ..., xn}, then the formula for normalization involves mapping each xi value to (xi – min)/(max – min), where min is the minimum value of X and max is the maximum value of X.

As a simple example, suppose that the random variable X has the values {-1, 0, 1}. Then min and max are 1 and –1, respectively, and the normalization of {-1, 0, 1} is the set of values {(-1-(-1))/2, (0-(-1))/2, (1-(-1))/2}, which equals {0, 1/2, 1}.

Scaling Numeric Data via Standardization

The standardization technique involves finding the mean mu and the standard deviation sigma, and then mapping each xi value to (xi – mu)/sigma. Recall the following formulas:

```
          mu = [SUM (x) ]/n
variance(x) = [SUM (x - xbar)*(x-xbar)]/n
      sigma = sqrt(variance)
```

As a simple illustration of standardization, suppose that the random variable X has the values {–1, 0, 1}. Then mu and sigma are calculated as follows:

```
mu       = (SUM xi)/n = (-1 + 0 + 1)/3 = 0
variance = [SUM (xi- mu)^2]/n
         = [(-1-0)^2 + (0-0)^2 + (1-0)^2]/3
         = 2/3
sigma    = sqrt(2/3) = 0.816 (approximate value)
```

Hence, the standardization of {-1, 0, 1} is {-1/0.816, 0/0.816, 1/0.816}, which in turn equals the set of values {-1.2254, 0, 1.2254}.

As another example, suppose that the random variable X has the values {-6, 0, 6}. Then mu and sigma are calculated as follows:

```
mu       = (SUM xi)/n = (-6 + 0 + 6)/3 = 0
variance = [SUM (xi- mu)^2]/n
         = [(-6-0)^2 + (0-0)^2 + (6-0)^2]/3
         = 72/3
         = 24
sigma    = sqrt(24) = 4.899 (approximate value)
```

Hence, the standardization of `{-6, 0, 6}` is `{-6/4.899, 0/4.899, 6/4.899}`, which in turn equals the set of values `{-1.2247, 0, 1.2247}`.

In the preceding two examples, the mean equals 0 in both cases, but the variance and standard deviation are significantly different. The normalization of a set of values *always* produces a set of numbers between 0 and 1.

However, the standardization of a set of values can generate numbers that are less than –1 and greater than 1; this will occur when `sigma` is less than the minimum value of every term `|mu – xi|`, where the latter is the absolute value of the difference between `mu` and each `xi` value. In the preceding example, the minimum difference equals 1, whereas `sigma` is 0.816, and therefore the largest standardized value is greater than 1.

What to Look for in Categorical Data

This section contains various suggestions for handling inconsistent data values, and you can determine which ones to adopt based on any additional factors that are relevant to your particular task. For example, consider dropping columns that have very low cardinality (equal to or close to 1), as well as numeric columns with zero or very low variance.

Next, check the contents of categorical columns for inconsistent spellings or errors. A good example pertains to the gender category, which can consist of a combination of the following values:

```
male
Male
female
Female
m
f
M
F
```

The preceding categorical values for gender can be replaced with two categorical values (unless you have a valid reason to retain some of the other values). Moreover, if you are training a model whose analysis involves a single gender, then you need to determine which rows (if any) of a dataset must be excluded. Also check categorical data columns for redundant or missing white spaces.

Check for data values that have multiple data types, such as a numerical column with numbers as numerals and some numbers as strings or objects.

Ensure consistent data formats (numbers as integers or floating numbers), and ensure that dates have the same format (for example, do not mix `mm/dd/yyyy` date formats with another date format, such as `dd/mm/yyyy`).

Mapping Categorical Data to Numeric Values

Character data is often called *categorical data*, examples of which include people's names, home or work addresses, and email addresses. Many types of categorical data involve short lists of values. For example, the days of the week and the months in a year involve seven and twelve distinct values, respectively. Notice that the days of the week have a relationship: For example, each day has a previous day and a next day. However, the colors of an automobile are independent of each other: the color red is not "better" or "worse" than the color blue.

There are several well-known techniques for mapping categorical values to a set of numeric values. A simple example where you need to perform this conversion involves the gender feature in the Titanic dataset. This feature is one of the relevant features for training a machine learning model. The gender feature has {M, F} as its set of possible values. As you will see later in this chapter, Pandas makes it very easy to convert the set of values {M, F} to the set of values {0, 1}.

Another mapping technique involves mapping a set of categorical values to a set of consecutive integer values. For example, the set {Red, Green, Blue} can be mapped to the set of integers {0, 1, 2}. The set {Male, Female} can be mapped to the set of integers {0, 1}. The days of the week can be mapped to {0, 1, 2, 3, 4, 5, 6}. Note that the first day of the week depends on the country: In some cases it's Sunday, and in other cases it's Monday.

Another technique is called *one-hot encoding*, which converts each value to a *vector* (check Wikipedia if you need a refresher regarding vectors). Thus, {Male, Female} can be represented by the vectors `[1, 0]` and `[0, 1]`, and the colors {Red, Green, Blue} can be represented by the vectors `[1, 0, 0]`, `[0, 1, 0]`, and `[0, 0, 1]`. If you vertically "line up" the two vectors for gender, they form a 2 × 2 identity matrix, and doing the same for the colors will form a 3 × 3 identity matrix.

If you vertically "line up" the two vectors for gender, they form a 2 × 2 identity matrix, and doing the same for the colors will form a 3 × 3 identity matrix, as shown here:

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

If you are familiar with matrices, you probably noticed that the preceding set of vectors looks like the 3 × 3 identity matrix. In fact, this technique generalizes in a straightforward manner. Specifically, if you have *n* distinct categorical values, you can map each of those values to one of the vectors in an *n* × *n* identity matrix.

As another example, the set of titles `{"Intern", "Junior", "Mid-Range", "Senior", "Project Leader", "Dev Manager"}` have a hierarchical relationship in terms of their salaries. Another set of categorical data involves the season of the year: `{"Spring", "Summer", "Autumn", "Winter"}`, and while these values are generally independent of each other, there are cases in which the season is significant. For example, the values for the monthly rainfall, average temperature, crime rate, or foreclosure rate can depend on the season, month, week, or even the day of the year.

If a feature has a large number of categorical values, then one-hot encoding will produce many additional columns for each data point. Since the majority of the values in the new columns equal 0, this can increase the sparsity of the dataset, which in turn can result in more overfitting and hence adversely affect the accuracy of machine learning algorithms that you adopt during the training process.

Another solution is to use a sequence-based solution in which *N* categories are mapped to the integers 1, 2, . . . , *N*. Another solution involves examining the row frequency of each categorical value. For example, suppose that *N* equals 20, and there are three categorical values that occur in 95% of the values for a given feature. You can try the following:

1. Assign the values 1, 2, and 3 to those three categorical values.
2. Assign numeric values that reflect the relative frequency of those categorical values.
3. Assign the category "OTHER" to the remaining categorical values.
4. Delete the rows whose categorical values belong to the 5%.

Working with Dates

The format for a calendar date varies among different countries, and this belongs to something called *localization* of data (not to be confused with `i18n`, which is data internationalization). Some examples of date formats are shown as follows (and the first four are probably the most common):

```
MM/DD/YY
MM/DD/YYYY
DD/MM/YY
DD/MM/YYYY
YY/MM/DD
M/D/YY
D/M/YY
YY/M/D
MMDDYY
DDMMYY
YYMMDD
```

If you need to combine data from datasets that contain different date formats, then converting the disparate date formats to a single common date format will ensure consistency.

Working with Currency

The format for currency depends on the country, which includes different interpretations for a "," and "." in currency (and decimal values in general). For example, 1,124.78 equals "one thousand one hundred twenty-four point seven eight" in the United States, whereas 1.124,78 has the same meaning in Europe (i.e., the "." symbol and the "," symbol are interchanged).

If you need to combine data from datasets that contain different currency formats, then you probably need to convert all the disparate currency formats to a single common currency format. There is another detail to consider: currency exchange rates can fluctuate on a daily basis, which in turn can affect the calculation of taxes, late fees, and so forth. Although you might be fortunate enough where you won't have to deal with these issues, it's still worth being aware of them.

MISSING DATA, ANOMALIES, AND OUTLIERS

Although missing data is not directly related to checking for anomalies and outliers, in general you will perform all three of these tasks. Each task involves a set of techniques to help you perform an analysis of the data in a dataset, and the following subsections describe some of those techniques.

Missing Data

How you decide to handle missing data depends on the specific dataset. Here are some ways to handle missing data (the first three techniques are manual techniques, and the other techniques are algorithms):

1. replace missing data with the mean/median/mode value
2. infer ("impute") the value for missing data
3. delete rows with missing data
4. isolation forest (tree-based algorithm)
5. minimum covariance determinant
6. local outlier factor
7. one-class SVM (Support Vector Machines)

In general, replacing a missing numeric value with zero is a risky choice: this value is obviously incorrect if the values of a feature are between 1,000 and 5,000. For a feature that has numeric values, replacing a missing value with the average value is better than the value zero (unless the average equals zero); also consider using the median value. For categorical data, consider using the mode to replace a missing value.

If you are not confident that you can impute a "reasonable" value, consider dropping the row with a missing value, and then train a model with the imputed value and also with the deleted row.

One problem that can arise after removing rows with missing values is that the resulting dataset is too small. In this case, consider using SMOTE, which is discussed later in this chapter, in order to generate synthetic data.

Anomalies and Outliers

In simplified terms, an outlier is an abnormal data value that is outside the range of "normal" values. For example, a person's height in centimeters is typically between 30 centimeters and 250 centimeters. Hence, a data point (e.g., a row of data in a spreadsheet) with a height of 5 centimeters or a height of 500 centimeters is an outlier. The consequences of these outlier values are unlikely to involve a significant financial or physical loss (though they could adversely affect the accuracy of a trained model).

Anomalies are also outside the "normal" range of values (just like outliers), and they are typically more problematic than outliers: anomalies can have more severe consequences than outliers. For example, consider the scenario in which someone who lives in California suddenly makes a credit

card purchase in New York. If the person is on vacation (or a business trip), then the purchase is an outlier (it's outside the typical purchasing pattern), but it's not an issue. However, if that person was in California when the credit card purchase was made, then it's most likely to be credit card fraud, as well as an anomaly.

Unfortunately, there is no simple way to *decide* how to deal with anomalies and outliers in a dataset. Although you can drop rows that contain outliers, keep in mind that doing so might deprive the dataset—and therefore the trained model—of valuable information. You can try modifying the data values (described as follows), but again, this might lead to erroneous inferences in the trained model. Another possibility is to train a model with the dataset that contains anomalies and outliers, and then train a model with a dataset from which the anomalies and outliers have been removed. Compare the two results and see if you can infer anything meaningful regarding the anomalies and outliers.

Outlier Detection

Although the decision to keep or drop outliers is your decision to make, there are some techniques available that help you detect outliers in a dataset. This section contains a short list of some techniques, along with a very brief description and links for additional information.

Perhaps *trimming* is the simplest technique (apart from dropping outliers), which involves removing rows whose feature value is in the upper 5% range or the lower 5% range. *Winsorizing* the data is an improvement over trimming: set the values in the top 5% range equal to the maximum value in the 95th percentile, and set the values in the bottom 5% range equal to the minimum in the 5th percentile.

The *Minimum Covariance Determinant* is a covariance-based technique, and a Python-based code sample that uses this technique is available online:

https://scikit-learn.org/stable/modules/outlier_detection.html.

The *Local Outlier Factor* (LOF) technique is an unsupervised technique that calculates a local anomaly score via the kNN (k Nearest Neighbor) algorithm. Documentation and short code samples that use LOF are available online:

https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.LocalOutlierFactor.html.

Two other techniques involve the Huber and the Ridge classes, both of which are included as part of Sklearn. The Huber error is less sensitive to

outliers because it's calculated via the linear loss, similar to the MAE (Mean Absolute Error). A code sample that compares Huber and Ridge is available online:

https://scikit-learn.org/stable/auto_examples/linear_model/plot_huber_vs_ridge.html.

You can also explore the Theil-Sen estimator and RANSAC, which are "robust" against outliers:

https://scikit-learn.org/stable/auto_examples/linear_model/plot_theilsen.html and

https://en.wikipedia.org/wiki/Random_sample_consensus.

Four algorithms for outlier detection are discussed at the following site:

https://www.kdnuggets.com/2018/12/four-techniques-outlier-detection.html.

One other scenario involves "local" outliers. For example, suppose that you use kMeans (or some other clustering algorithm) and determine that a value is an outlier with respect to one of the clusters. While this value is not necessarily an "absolute" outlier, detecting such a value might be important for your use case.

What is Data Drift?

The value of data is based on its accuracy, its relevance, and its age. Data drift refers to data that has become less relevant over time. For example, online purchasing patterns in 2010 are probably not as relevant as data from 2020 because of various factors (such as the profile of different types of customers). Keep in mind that there might be multiple factors that can influence data drift in a specific dataset.

Two techniques are domain classifier and the black-box shift detector, both of which are discussed online:

https://blog.dataiku.com/towards-reliable-mlops-with-drift-detectors.

WHAT IS IMBALANCED CLASSIFICATION?

Imbalanced classification involves datasets with imbalanced classes. For example, suppose that class A has 99% of the data and class B has 1%. Which classification algorithm would you use? Unfortunately, classification algorithms

don't work well with this type of imbalanced dataset. Here is a list of several well-known techniques for handling imbalanced datasets:

- Random resampling rebalances the class distribution.
- Random oversampling duplicates data in the minority class.
- Random undersampling deletes examples from the majority class.
- SMOTE

Random resampling transforms the training dataset into a new dataset, which is effective for imbalanced classification problems.

The *random undersampling* technique removes samples from the dataset, and involves the following:

- randomly remove samples from majority class
- can be performed with or without replacement
- alleviates imbalance in the dataset
- may increase the variance of the classifier
- may discard useful or important samples

However, random undersampling does not work well with a dataset that has a 99%/1% split into two classes. Moreover, undersampling can result in losing information that is useful for a model.

Instead of random undersampling, another approach involves generating new samples from a minority class. The first technique involves oversampling examples in the minority class and duplicate examples from the minority class.

There is another technique that is better than the preceding technique, which involves the following:

- synthesize new examples from minority class
- a type of data augmentation for tabular data
- this technique can be very effective
- generate new samples from minority class

Another well-known technique is called SMOTE, which involves data augmentation (i.e., synthesizing new data samples) well before you use a classification algorithm. SMOTE was initially developed by means of the kNN algorithm (other options are available), and it can be an effective technique for handling imbalanced classes.

Yet another option to consider is the Python package `imbalanced-learn` in the `scikit-learn-contrib` project. This project provides various re-sampling techniques for datasets that exhibit class imbalance. More details are available online:

https://github.com/scikit-learn-contrib/imbalanced-learn.

WHAT IS SMOTE?

SMOTE is a technique for synthesizing new samples for a dataset. This technique is based on linear interpolation:

- Step 1: Select samples that are close in the feature space.
- Step 2: Draw a line between the samples in the feature space.
- Step 3: Draw a new sample at a point along that line.

A more detailed explanation of the SMOTE algorithm is as follows:

- Select a random sample "a" from the minority class.
- Find k nearest neighbors for that example.
- Select a random neighbor "b" from the nearest neighbors.
- Create a line "L" that connects "a" and "b."
- Randomly select one or more points "c" on line L.

If need be, you can repeat this process for the other (k-1) nearest neighbors to distribute the synthetic values more evenly among the nearest neighbors.

SMOTE Extensions

The initial SMOTE algorithm is based on the kNN classification algorithm, which has been extended in various ways, such as replacing kNN with SVM. A list of SMOTE extensions is shown as follows:

- selective synthetic sample generation
- Borderline-SMOTE (kNN)
- Borderline-SMOTE (SVM)
- Adaptive Synthetic Sampling (ADASYN)

ANALYZING CLASSIFIERS (OPTIONAL)

This section is marked "optional" because its contents pertain to machine learning classifiers, which are not the focus of this book. However, it's still worthwhile to glance through the material, or perhaps return to this section after you have a basic understanding of machine learning classifiers.

Several well-known techniques are available for analyzing the quality of machine learning classifiers. Two techniques are LIME and ANOVA, both of which are discussed in the following subsections.

What is LIME?

LIME is an acronym for Local Interpretable Model-Agnostic Explanations. LIME is a model-agnostic technique that can be used with machine learning models. In LIME, you make small random changes to data samples and then observe the manner in which predictions change (or not). The approach involves changing the output (slightly) and then observing what happens to the output.

By way of analogy, consider food inspectors who test for bacteria in truckloads of perishable food. Clearly, it's infeasible to test every food item in a truck (or a train car), so inspectors perform "spot checks" that involve testing randomly selected items. In an analogous fashion, LIME makes small changes to input data in random locations and then analyzes the changes in the associated output values.

However, there are two caveats to keep in mind when you use LIME with input data for a given model:

1. The actual changes to input values are model-specific.
2. This technique works on input that is interpretable.

Examples of interpretable input include machine learning classifiers (such as trees and random forests) and NLP techniques such as BoW (Bag of Words). Non-interpretable input involves "dense" data, such as a word embedding (which is a vector of floating point numbers).

You could also substitute your model with another model that involves interpretable data, but then you need to evaluate how accurate the approximation is to the original model.

What is ANOVA?

ANOVA is an acronym for *analysis of variance*, which attempts to analyze the differences among the mean values of a sample that's taken from a population. ANOVA enables you to test if multiple mean values are equal. More importantly, ANOVA can assist in reducing Type I (false positive) errors and Type II errors (false negative) errors. For example, suppose that person A is diagnosed with cancer and person B is diagnosed as healthy, and that both diagnoses are incorrect. Then the result for person A is a false positive whereas the result for person B is a false negative. In general, a test result of false positive is much preferable to a test result of false negative.

ANOVA pertains to the design of experiments and hypothesis testing, which can produce meaningful results in various situations. For example,

suppose that a dataset contains a feature that can be partitioned into several "reasonably" homogenous groups. Next, analyze the variance in each group and perform comparisons with the goal of determining different sources of variance for the values of a given feature.

THE BIAS-VARIANCE TRADE-OFF

This section is presented from the viewpoint of machine learning, but the concepts of bias and variance are highly relevant outside of machine learning.

Bias in machine learning can be due to an error from wrong assumptions in a learning algorithm. High bias might cause an algorithm to miss relevant relations between features and target outputs (underfitting). Prediction bias can occur because of "noisy" data, an incomplete feature set, or a biased training sample.

Error due to bias is the difference between the expected (or average) prediction of your model and the correct value that you want to predict. Repeat the model building process multiple times, and gather new data each time, and also perform an analysis to produce a new model. The resulting models have a range of predictions because the underlying datasets have a degree of randomness. Bias measures the extent to which the predictions for these models deviate from the correct value.

Variance in machine learning is the expected value of the squared deviation from the mean. High variance can/might cause an algorithm to model the random noise in the training data, rather than the intended outputs (aka overfitting). Moreover, adding parameters to a model increases its complexity, increases the variance, and decreases the bias.

Dealing with bias and variance involves addressing underfitting and overfitting.

Error due to variance is the variability of a model prediction for a given data point. As before, repeat the entire model building process, and the variance is the extent to which predictions for a given point vary among different "instances" of the model.

If you have worked with datasets and performed data analysis, you already know that finding well-balanced samples can be difficult or highly impractical. Moreover, performing an analysis of the data in a dataset is vitally important, yet there is no guarantee that you can produce a dataset that is 100% "clean."

A *biased statistic* is a statistic that is systematically different from the entity in the population that is being estimated. In more casual terminology,

if a data sample "favors" or "leans" toward one aspect of the population, then the sample has bias. For example, if you prefer movies that are comedies, then clearly you are more likely to select a comedy instead of a dramatic movie or a science fiction movie. Thus, a frequency graph of the movie types in a sample of your movie selections will be more closely clustered around comedies.

However, if you have a wide-ranging set of preferences for movies, then the corresponding frequency graph will be more varied, and therefore have a larger spread of values. As a simple example, suppose that you are given an assignment that involves writing a term paper on a controversial subject that has many opposing viewpoints. Since you want a bibliography that supports your well-balanced term paper that takes into account multiple viewpoints, your bibliography will contain a wide variety of sources. In other words, your bibliography will have a larger variance and a smaller bias. However, if most (or all) the references in your bibliography espouse the same point of view, then you will have a smaller variance and a larger bias (it's just an analogy, so it's not a perfect counterpart to bias vs. variance).

The bias-variance trade-off can be stated in simple terms: in general, reducing the bias in samples can increase the variance, whereas reducing the variance tends to increase the bias.

Types of Bias in Data

In addition to the bias-variance trade-off that is discussed in the previous section, there are several types of bias, some of which are listed as follows:

- Availability Bias
- Confirmation Bias
- False Causality
- Sunk Cost Fallacy
- Survivorship Bias

Availability bias is akin to making a "rule" based on an exception. For example, there is a known link between smoking cigarettes and cancer, but there are exceptions. If you find someone who has smoked three packs of cigarettes on a daily basis for four decades and is still healthy, can you assert that smoking does not lead to cancer?

Confirmation bias refers to the tendency to focus on data that confirms one's beliefs and simultaneously ignore data that contradicts a belief.

False causality occurs when you incorrectly assert that the occurrence of a particular event causes another event to occur as well. One of the most well-known examples involves ice cream consumption and violent crime in New

York during the summer. Since more people eat ice cream in the summer, that "causes" more violent crime, which is a false causality. Other factors, such as the increase in temperature, may be linked to the increase in crime. However, it's important to distinguish between correlation and causality: the latter is a much stronger link than the former, and it's also more difficult to establish causality instead of correlation.

Sunk cost refers to something (often money) that has been spent or incurred that cannot be recouped. A common example pertains to gambling at a casino: People fall into the pattern of spending more money in order to recoup a substantial amount of money that has already been lost. While there are situations in which people do recover their money, in many cases, people simply incur an even greater loss because they continue to spend their money.

Survivorship bias refers to analyzing a particular subset of "positive" data while ignoring the "negative" data. This bias occurs in various situations, such as being influenced by individuals who recount their rags-to-riches success story ("positive" data) while ignoring the fate of the people (which is often a very high percentage) who did not succeed (the "negative" data) in a similar quest. So, while it's certainly possible for an individual to overcome many difficult obstacles in order to succeed, is the success rate one in one thousand (or even lower)?

SUMMARY

This chapter started with an explanation of datasets, a description of data wrangling, and details regarding various types of data. Then you learned about techniques for scaling numeric data, such as normalization and standardization. You saw how to convert categorical data to numeric values, and how to handle dates and currency.

Then you learned some of the nuances of missing data, anomalies, and outliers, and techniques for handling these scenarios. You also learned about imbalanced data and evaluating the use of SMOTE to deal with imbalanced classes in a dataset. In addition, you learned about classifiers using two techniques, LIME and ANOVA. Finally, you learned about the bias-variance trade-off and various types of statistical bias.

CHAPTER 2

NLP Concepts (I)

This chapter is the first chapter that contains NLP-related material, starting with a high-level introduction to some major language groups and the substantive grammatical differences among languages. Then we discuss some basic concepts in NLP, such as text normalization, the concepts of stop words, stemming, and lemmatization (the dictionary form of words), POS (Parts Of Speech) tagging, and NER (Named Entity Recognition).

This chapter focuses on NLP concepts, and while some NLP algorithms are mentioned in this Chapter, the relevant code samples are provided Chapter 5 and Chapter 6. Depending on your NLP background, you might decide to read the sections in a nonsequential fashion. If your goal is to proceed quickly to code samples, you can skip some sections in this chapter, and later you can return to read those omitted sections.

The first part of the chapter provides an abbreviated tour of several languages that belong to major human language groups, illustrating some of the facets of human languages that can make NLP a truly challenging endeavor. However, please keep in mind that this section contains many details that appeal primarily to language aficionados. This section contains grammatical details that differentiate various languages from each other that highlight the complexity of generating native-level syntax as well as native-level pronunciation. Depending on your level of interest, feel free to read the portions of this section that interest you and then proceed to the next section of this chapter.

The discussion regarding regional accents and slang contain anecdotal observations based on the experiences of the author: there is no scientifically rigorous basis or any studies to support those observations, which means that they are not necessarily true in a general case. However, you might find some of the sections somewhat interesting (and in some cases, they might be similar to your own experiences).

Indeed, various subsections reflect the author's experiences in multilingual environments while living and working in various countries: in particular, this includes Italian, Spanish, French, and Japanese, as well as language dialects (the Venetian dialect and Venezuelan Criollo) and the challenges facing nonnative-English speakers.

The second part of the chapter introduces you to NLP and a brief history of the major stages of NLP. We include NLP applications, NLP use cases, NLU, and NLG. We also discuss word sense disambiguation. This section only provides a brief description of these topics, some of which can fill entire books and full-length courses.

The third part of this chapter discusses various NLP techniques and the major steps in an NLP-related process. You also learn about standard NLP-related tasks, such as text normalization, tokenization, stemming, lemmatization, and the removal of stop words. Some of these tasks (e.g., tokenization) involve implicit assumptions that are not true for all languages.

The final section introduces NER and topic modeling, which involves named entities and finding the main topic(s) in a text document.

THE ORIGIN OF LANGUAGES

Someone once remarked that "the origin of language is an enigma," which is viscerally appealing because it has at least a kernel of truth. Although there are multiple theories that attempt to explain how and why languages developed, none of them has attained universal consensus. Nevertheless, there is no doubt that humans have far surpassed all other species in terms of language development.

There is also the question of how the vocabulary of a language is formed, which can be the confluence of multiple factors, as well as meaning in a language. According to Ludwig Wittgenstein (1953), who was an influential philosopher in many other fields, language derives its meaning from use.

One theory about the evolution of language in humans asserts that the need for communication between humans makes language a necessity. Another explanation is that language is influenced by the task of creating complex tools, because the latter requires a precise sequence of steps, which ultimately spurred the development of languages.

Without delving into their details, the following list contains some theories that have been proposed regarding language development. Keep in mind that they vary in terms of their support in the academic community:

- Strong Minimalist Thesis
- The FlintKnapper Theory
- The Sapir-Whorf Hypothesis
- Universal Grammar (Noam Chomsky)

The Strong Minimalist Thesis (SRT) asserts that language is based on something called the hierarchical syntactic structure. The FlintKnapper Theory asserts that the ability to create complex tools involved an intricate sequence of steps, which in turn necessitated communication between people. In simplified terms, the Sapir–Whorf Hypothesis (also called the *linguistic relativity hypothesis*, which is a slightly weaker form) posits that the language we speak influences how we think. Consider how our physical environment can influence our spoken language: Eskimos have several words to describe snow, whereas people in some parts of the Middle East have never seen a snow storm.

Universal Grammar is a genetic-based theory by Noam Chomsky in which he asserts that all humans have an innate capacity to learn languages (provided that they are raised in a reasonably normal environment). This innate capacity is not bound to a grammar or vocabulary of any human language, and diverges from earlier "tabula rasa" (blank slate) theories regarding the human mind at birth. While Chomsky's theory has appealing aspects, there are critic of UG, which you can read if you are interested in the details:

https://en.wikipedia.org/wiki/Universal_grammar

Despite the grammatical diversity of human languages and the rich set of sounds that are possible in human languages, consider the following fact: a healthy newborn infant from one country can be placed in any other country and learn to speak the common language of that country, regardless of the genetic makeup of the infant.

Hence, humans have a universal capacity to learn languages, and seem to have an innate ability to learn multiple languages. This capacity to learn languages separates us from animals because the latter are unable to create a language that is close to the complexity of human languages. Noam Chomsky explains that this capability exists in humans because of a broader skill: humans have a recursive-like capacity to mentally combine objects to create new objects.

Language Fluency

As mentioned in the previous section, human infants are capable of producing the sounds of any language, given enough opportunity to imitate those

sounds. They tend to lose some of that capacity as they become older, which might explain why some adults speak another language with an accent (of course, there are plenty of exceptions).

Interestingly, babies respond favorably to the sound of vowel-rich "Parentese," and a study in 2018 suggested that babies prefer the sound of other babies instead of their mother:

https://eurekalert.org/pub_releases/2018-05/asoa-ftm042618.php
https://getpocket.com/explore/item/babies-prefer-the-sounds-of-other-babies-to-the-cooing-of-their-parents

There are two interesting cases in which people can acquire native-level speech capability. The first case is intuitive: people who have been raised in a bilingual (or multilingual) environment tend to have a greater capacity for learning how to speak other languages with native level (or near native level) speech. Second, people who speak phonetic languages have an advantage when they study another phonetic language, especially one that is in their language group, because they already know how to pronounce the majority of vowel sounds.

However, there are consonants that occur in a limited number of languages whose pronunciation can be a challenge for practically every non-native speaker. For example, letters that have a guttural sound (such as those in Dutch, German, and Arabic), the glottal stop (most noticeable in Arabic), and the letter "ain" in Arabic are generally more challenging to pronounce for native speakers of romance languages and some Asian languages.

To some extent, the non-phonetic nature of the English language might explain why some monolingual native-English speakers might struggle with learning to speak other languages with native-level speech. Perhaps the closest language to English (in terms of cadence) is Dutch, and people from Holland can often speak native-level English. This tends to be true of Swedes and Danes as well, whose languages are Germanic, but not necessarily true of Germans, who can speak perfect grammatical English but sometimes speak English with an accent.

Perhaps somewhat ironically, sometimes accents can impart a sort of cachet, such as speaking with a British or Australian accent in the United States. Indeed, a French accent can also add a certain *je-ne-sais-quoi* to a speaker in various parts of the United States.

Major Language Groups

There are more than 140 language families, and the six largest language families (based on language count) are listed here:

- Niger-Congo
- Austronesian
- Trans-New Guinea
- Sino-Tibetan
- Indo-European
- Afro-Asiatic

English belongs to the Indo-European group, Mandarin belongs to the Sino-Tibetan, and Arabic belongs to the Afro-Asiatic group. According to Wikipedia, Indo-European languages comprise almost 600 languages, including most of the languages in Europe, the northern Indian subcontinent, and the Iranian plateau. Almost half the world speaks an Indo-European language as a native language, which is greater than any of the language groups listed in the introduction of this section. Indo-European has several major language subgroups, which are Germanic, Slavic, and Romance languages. The preceding information is from the following Wikipedia link:

https://en.wikipedia.org/wiki/List_of_language_families

As of 2019, the top four languages that are spoken in the world, which counts the number of people who are native speakers or secondary speakers, are as follows:

- English: 1.268 billion
- Mandarin: 1.120 billion
- Hindi: 637.3 million
- Spanish: 537.9 million
- French: 276.6 million

The preceding information is from the following Wikipedia link:

https://en.wikipedia.org/wiki/List_of_languages_by_total_number_of_speakers

Many factors can influence the expansion of a given language into multiple countries, such as commerce, economic factors, technological influence, and warfare, thereby resulting in the absorption of new words by another language. Somewhat intuitively, countries with a common border influence each other's language, sometimes resulting in new hybrid languages. For example,

Catalan is a hybrid of Spanish and French and Provencal is a hybrid of French and Italian (both of which have delicious cuisine) that are spoken by people who live close to the border of the respective adjacent countries. Other examples include the influence of Farsi on Urdu (spoken in Pakistan) and the influence of French on Vietnamese and the presence of French words in some Arab countries.

Surprisingly, sometimes languages from geographically distant countries share linguistic features. For example, the Finno-Ugric (or Finno-Ugrian) language group comprises Hungarian, Finnish, and Estonian because they are related, despite their geographic distance from each other. Nevertheless, a plausible explanation may well exist; the other explanation for their commonality is due to random events (which seems unlikely).

Peak Usage of Some Languages

As you might have surmised, different languages have been in an influential position during the past 2,000 years. If you trace the popularity and influence of Indo-European languages, you will find periods of time with varying degrees of influence involving multiple languages, including Hebrew, Greek, Latin, Arabic, French, and English.

Latin is an Indo-European language (apparently derived from the Etruscan and Greek alphabets), and during the 1st century AD, Latin became a mainstream language. In addition, romance languages are derived from Latin. Today Latin is considered a dead language in the sense that it's not actively spoken on a daily basis by large numbers of people. The same is true of Sanskrit, which is a very old language from India.

During the Roman Empire, Latin and Greek were the official languages for administrative as well as military activities. In addition, Latin was an important language for diplomacy among countries for many centuries after the fall of the Roman Empire.

You might be surprised to know that Arabic was the *lingua franca* throughout the Mediterranean during the 10th and 11th centuries AD. As another example, French was spoken in many parts of Europe during the 18th century, including the Russian aristocracy.

Today English appears to be in its ascendancy in terms of the number of native English speakers as well as the number of people who speak English as a second (or third or fourth) language. Although Mandarin is a widely spoken Asian language, English is the *lingua franca* for commerce as well as technology: virtually every computer language is based on English.

Languages and Regional Accents

Accents, slang, and dialects have some common features, but there can be some significant differences. Accents involve modifying the standard pronunciation of words, which can vary significantly in different parts of the same country.

One interesting phenomenon pertains to the southern region of some countries (in the northern hemisphere), which tend to have a more "relaxed" pronunciation compared to the northern region of that country. For example, some people in the southeastern United States speak with a so-called "drawl," whereas newscasters will often speak with a midwestern pronunciation, which is considered a neutral pronunciation. The same is true of people in Tokyo, who often speak Japanese with a "flat" pronunciation (which is also true of Japanese newscasters on NHK), versus people from the Kansai region (Kyoto, Kobe, and Osaka) of Japan, who vary the tone and emphasis of Japanese words.

Regional accents can also involve modifying the meaning of words in ways that are specific to the region in question. For example, Texans will say "I'm fixing to graduate this year" whereas people from other parts of the United States would say "going" instead of "fixing." In France, Parisians are unlikely to say *Il faut fatiguer la salade* ("it's necessary to toss the salad"), whereas this sentence is much more commonplace in southern France. (The English word "fatigue" is derived from the French verb *fatiguer*)

Languages and Slang

The existence of slang words is interesting and perhaps inevitable, they seem to flourish in every human language. Sometimes slang words are used for obfuscation so that only members of an "in group" understand the modified meaning of those words. Slang words can also be a combination of existing words, new words (but not officially recognized), and short-hand expressions. Slang can also "invert" the meaning of words ("bad" instead of "good"), which can be specific to an age group, minority, or region. In addition, slang can also assign an entirely unrelated meaning to a standard word (e.g., the slang terms "that's dope," "that's sick," and "the bomb").

Slang words can also be specific to an age group to prevent communication with members of different age groups. For example, Japanese teens can communicate with each other by reversing the order of the syllables in a word, which renders those "words" incomprehensible to adults. The inversion of syllables is far more complex than "pig Latin," in which the first letter of a word

is shifted to the end of the word, followed by the syllable "ay." For example, "East Bay" (an actual location in the Bay Area in Silicon Valley) is humorously called "beast" in pig Latin.

Teenagers also use acronyms (perhaps as another form of slang) when sending text messages to each other. For example, the acronym "aos" means "adult over shoulder." The acronym "bos" has several different meanings, including "brother over shoulder" and "boyfriend over shoulder."

The slang terms that you use with your peers invariably simplifies communication with others in your in-group, sometimes accompanied by specialized interpretations to words (such as reversing their meaning). A simple example is the word *zanahoria*, which is the Spanish word for carrot. In colloquial speech in Venezuela, calling someone a *zanahoria* means that that person is very conservative and as "straight" as a carrot.

Slang enables people to be creative and also playfully break the rules of language. Both slang and colloquial speech simplify formal language and rarely (if ever) introduce greater complexity in alternate speech rules.

Perhaps that's the reason that slang and colloquial speech cannot be controlled or regulated by anyone (or by any language committee): like water, they are fluid and adapt to the preferences of their speakers.

One more observation: while slang can be viewed as a creative by-product of standard speech, there is a reverse effect that can occur in certain situations. For example, you have probably noticed how influential subgenres are eventually absorbed (perhaps only partially) into mainstream culture: witness how commercials eventually incorporated a "softened" form of rap music and its rhythm in commercials for personal products. There's a certain irony in hearing "Stairway to Heaven" as elevator music.

Another interesting concept is a "meme" (which includes Internet memes) in popular culture, which refers to something with humorous content. While slang words are often used to exclude people, a meme often attempts to communicate a particular sentiment. One such meme is "OK Boomer," which some people view as a derogatory remark that's sometimes expressed in a snarky manner, and much less often interpreted as a humorous term. Although language dialects can also involve regional accents and slang, they also have more distinct characteristics, as discussed in the next section.

Languages and Dialects

Dialects often replace standard words with substantively different words that have the same meaning as their standard counterpart. In fact, dialects often include words that do not even exist in the language that they are based on.

However, dialects also tend to have a consistent set of grammatical rules for conjugating verbs.

For example, despite having a population of under 60 million people, Italian has several dozen dialects, some of which are pair-wise incomprehensible to people from different regions of Italy. For instance, here are some words in the Venetian dialect (word spellings are approximate and some accent marks have been omitted), along with their counterpart in standard Italian, followed by their translation into English:

- *bragghe* means *pantaloni* (pants)
- *ciappa* means *ha preso* (he/she got)
- *coppa* means *ammazza* (kills)
- *ghe xe* means *c'e* (there is)
- *schei* means *soldi* (money)
- *toxhi* means *bambini* (children)

As you can see, the spelling of the preceding words in the Venetian dialect bear no resemblance to their counterparts in standard Italian.

Dialects can also have significant differences, even in cities that are relatively close to each other. For example, Milan and Vicenza are two cities in northern Italy, with Milan located to the west of Vicenza, and slightly more than 100 kilometers (60 miles) apart. Maniago is a town in northeastern Italy (and well known for its production of steel blades that are in knives), located roughly 50 kilometers (32 miles) northwest of Vicenza. However, Milanese and the Venetian dialect are much closer to each other than to Friuliano, which is the dialect spoken in Maniago. In fact, Maniago is close to the border of Yugoslavia, and the influence of the latter is visible in the names of towns and highway signs that are near Maniago.

Given the differences in the dialects of Italian, how can people communicate? In large cities such as Milan, which comprise people from many parts of Italy, the only way that people can communicate with each other is to speak standard Italian.

THE COMPLEXITY OF NATURAL LANGUAGES

This section contains many subsections, and to give you some context, consider the scenario in which two people are having a conversation in which they do not speak a common language. In addition to human translators, there are software applications to perform the translation task. However, in the latter

case, translating a sentence to a different language in such a way that it sounds like a native speaker involves many details. This section highlights some aspects of the translation process between different languages, and especially between languages that are in different language groups.

Natural languages involve a set of grammar rules of varying degrees of complexity, along with language specific features. For example, English, romance languages, and some Asian languages have a subject/verb/object pattern for many sentences.

By contrast, Japanese and Korean have a subject/object/verb pattern (German has a subject/verb/object/verb pattern for compound verbs), along with declension of adjectives and nouns (in German and Slavic languages) or postpositions (in Japanese) that serve as "markers" for the grammatical function of nouns in sentences. As a result, it's possible to change the order of the words in sentences in German, Japanese, and Slavic languages and still maintain exactly the same meaning of those sentences.

Another interesting fact: although most languages are written in a left-to-right manner, some are written in a right-to-left fashion (including Hebrew and Arabic) or a top-to-bottom fashion (Japanese does both). If that doesn't impress you, consider the fact that some languages (including Hebrew and Arabic) also treat vowels as optional: native speakers of these languages have the advantage of having learned their vocabulary since childhood, so they recognize the meaning of words without vowels.

Word Order in Sentences

As mentioned previously, German and Slavic languages allow for a rearrangement of the words in sentences because those languages support *declension*, which involves modifying the endings of articles and adjectives in accordance with the grammatical function of those words in a sentence (such as the subject, direct object, and indirect object). Those word endings are loosely comparable to prepositions in English, and sometimes they have the same spelling for different grammatical functions. For example, in German, the article *den* precedes a masculine noun that is a direct object and also a plural noun that is an indirect object: ambiguity can occur if the singular masculine noun has the same spelling in its plural form.

Alternatively, since English is word order dependent, ambiguity can still arise in sentences, which we have learned to parse correctly without any conscious effort.

Groucho Marx often incorporated ambiguous sentences in his dialogues, such as the following paraphrased examples:

"This morning I shot an elephant in my pajamas. How he got into my pajamas I have no idea."

"In America, a woman gives birth to a child every fifteen minutes. Somebody needs to find that woman and stop her."

Now consider the following pair of sentences involving a boy, a mountain, and a telescope:

I saw the boy on the mountain with the telescope.
I saw the boy with the telescope on the mountain.

Human speakers interpret both English sentences as having the same meaning; however, arriving at the same interpretation is less obvious from the standpoint of a purely NLP task. Why does this ambiguity in the preceding example not arise in Russian? The reason is simple: the preposition *with* is associated with the instrumental case in Russian, whereas *on* is not the instrumental case, and therefore the nouns have suffixes that indicate the distinction.

What about Verbs?

Verbs exist in every written language, and they undergo conjugation that reflects their tense and mood in a sentence. Such languages have an overlapping set of verb tenses, but there are differences. For instance, Portuguese has a future perfect subjunctive, as does Spanish (but it's almost never used in spoken form), whereas these verb forms do not exist in English. English verb tenses (in the indicative mood) can include:

- present
- present perfect
- present progressive
- present perfect progressive
- preterite (simple past)
- past perfect
- past progressive
- past perfect progressive
- future tense
- future perfect
- future progressive
- future perfect progressive (does not exist in Italian)

Here are some examples of English sentences that illustrate (most of) the preceding verb forms:

- I read a book.
- I have read a book.
- I am reading a book.
- I have been reading a book.
- I read a book.
- I have read a book.
- I had been reading a book.
- I will read a book.
- I will have read a book.
- I will be reading a book.
- At 6 p.m., I will have been reading a book for 3 hours.

Verb moods can be indicative (as shown in the preceding list), subjunctive (discussed soon), and conditional ("I would go but I have work to do"). In English, subjunctive verb forms can include the present subjunctive ("I insist that he do the task"), the past subjunctive ("If I were you"), and the pluperfect subjunctive ("Had I but known …"). Interestingly, Portuguese also provides a future perfect subjunctive verb form; Spanish also has this verb form but it's never used in conversation.

Interestingly (from a linguistic perspective, at least), there are modern languages, such as Mandarin, that have only one verb tense: they rely on other words in a sentence (such as time adverbs or aspect particles) to convey the time frame. Such languages would express the present, the past, and the future in a form that is comparable to the following:

- "I read a book now."
- "I read a book yesterday."
- "I read a book tomorrow."

Auxiliary Verbs

Languages such as Italian and French use the verbs "to be" and "to have" as auxiliary verbs. In particular, Italian uses the verb *essere* and French uses the verb *etre* (note that an accent mark is missing here) as an auxiliary verb with intransitive (no direct object) verbs of motion. By contrast, English always uses the verb "to have" in sentences that contain compound verb forms.

Here are some examples of sentences that contain auxiliary verbs in English, French, and Italian:

- I have gone to school.
- *Je suis allé à l'école.*
- *Sono andato a scuola*.

Compound verbs involving motion in French and Italian use *essere* and *essere*, respectively, whereas English always uses "have." Hence, the following sentences are *incorrect* because the French sentence has the verb *avoir* and the Italian sentence has the verb *avere* as the auxiliary verb:

- *J'ai allé à l'école.*
- *Ho andato a scuola*.

French and Italian tend to use the present perfect (which involves an auxiliary verb) in conversations, whereas English, Portuguese, and Spanish tend to use the simple past (also called the preterit, which does not involve an auxiliary verb):

Spanish:

- *Fui a la escuela.* (I went to school.)
- *He ido a la escuela*. (I have gone to school.)

Portuguese:

- *Eu fui a escola* ("I went to school")
- *Eu he ido para a escola* ("I have gone to school")

Spanish and Portuguese have an additional interesting feature: they have two verbs, *estar* and *ser*, that are related "to be" yet have different connotations. The verb *estar* refers to a temporary or transient state, such as *Estoy aqui* (I am here) or *Estoy cansado* (I am tired), whereas the verb *ser* refers to a (perceived) longer term state, such as *Soy rico* (I am rich) or *Soy viejo* (I am old).

In Portuguese, the corresponding sentences are *Estou aqui* (I am here), *Estou cansado* (I am tired), *Sou rico* (I am rich), and *Sou velho* (I am old). The word *rico* is singular masculine. *Rica* and *ricos* are for singular feminine and plural (male and female), respectively.

Verbs sometimes undergo changes in pronunciation and sentence undergo "contractions" in casual conversation. Here are examples of three sentences that mean "I do not know" in French, and only the first sentence is grammatically correct:

- *Je ne sais pas.*
- *Je sais pas.*
- *Sheh pas.* (This is similar to saying "I dunno.")

What are Case Endings?

A case ending is a suffix of a word that indicates the grammatical function of a word in a sentence. English has no case endings (except in rare cases) and is also word-order dependent, which means that the following pair of sentences have the opposite meaning:

- The man sees the dog.
- The dog sees the man.

The first sentence can be written in two ways in German (notice the definite articles *den* and *der*), both of which have the same meaning:

- *Der Mann sieht* ***den*** *Hund.*
- ***Den*** *Hund sieht der Mann.*

The German article *den* indicates a direct object, which means that the previous pair of sentences have the same meaning in German.

German has case endings for articles and adjectives. For example, *ein*, *diese*, and *gut* mean "a," "the," and "good," respectively, in German. When these words are used in the dative (indirect object) case, they become *einem*, *diesem*, and *gutem*, for a masculine noun. Here's a summary of case endings for German:

Table 2.1 Case Endings in German.

	Masc	Fem	Neut	Plural
Nom	der	die	das	die
Gen	des	der	des	der
Dat	dem	der	dem	den
Acc	den	die	das	die

Interestingly the indirect object pronoun *ihm* in German has the counterpart "him" in English, and the word *ihr* in German is the counterpart to the word "her" in English. Thus, the syntax and bounces in the following German sentence is similar to its English counterpart:

Ich gebe ihm das Buch. (I give him the book.)
Ich gebe ihr das Buch. (I give her the book.)

The preceding sentence has a subject/verb/object structure because the verb is a simple verb. If you use a compound verb involving an auxiliary verb,

then the structure of the sentence is subject/verb/object/verb. For example, the following German sentence means "I have given him a book:"

Ich ***habe*** *ihm das Buch* ***gegeben.*** (I **have given** *him* the book.)

Let's return to the topic of case endings. The following languages also have case endings (in increasing order with respect to the number of cases in each language):

- Arabic (3)
- German (4)
- Greek (5)
- Russian (6)
- Lithuanian (7)
- Latin (15)
- Finnish (21; but no gender)

By contrast, English, romance languages, and Asian languages (Cantonese, Mandarin, Japanese, and Korean) do not have case endings. The meaning of sentences in English and romance languages is typically word-order dependent. However, Korean and Japanese both have postpositions that indicate the grammatical function of the nouns in a sentence, so it's possible to reorder sentences in both of these languages and still retain the same meaning of the original sentence. Context is also very useful, especially in Japanese sentences that are ambiguous in terms of the number of people (or objects) that are referenced.

Languages and Gender

Romance languages have a masculine and feminine form for nouns and are preceded by definite articles that reflect the gender and number of nouns. Although Romanian is a romance language, it has a masculine, feminine, and neuter form for nouns. When you consider the fact that romance languages are derived from Latin, which has a masculine, feminine, and neuter form for nouns, perhaps Romanian is the only romance language that retained the neuter form for nouns.

Germanic languages and Slavic languages also have three genders, and the endings of adjectives and definite/indefinite articles that precede them are modified (see the section regarding case endings). By contrast, English, Finnish, Japanese, and Korean do not have gender forms for nouns.

Singular and Plural Forms of Nouns

All the languages in the preceding section that have two or more genders also have singular and plural forms for nouns. Here are examples of sentences in Italian about buying one or more books:

- *Ho comperato il libro*. [I bought the book.]
- *L'ho comperat**o***. [I bought it (ex: a book).]
- *L'ho comperat**a***. [I bought it (ex: a car).]
- *Ho comperato i libri*. [I bought the books.]
- *Gli ho comperati*. [I bought them (the books).]

Notice the use of *il* for the singular case and *i* for the plural case, as well as *L'ho* and *gli ho* for the direct object referring to one versus multiple books, respectively. In addition, the second sentence changes the verb from *comperato* to *comperata* to *comperati*, because its form must agree in gender and number when there is a preceding direct object. Hence, the English sentence "I bought them" when referring to a feminine plural noun is written as follows in Italian:

Le ho comperate.

Once again, Finnish, Japanese, and Korean do not have a plural form for nouns, which avoids having to learn rules for forming the plural of nouns. However, most of them involve other grammatical challenges for non-native speakers.

Changes in Spelling of Words

Another example involves different spellings for the same word, such as center/centre, favor/favour, and color/colour. These variations in spelling appear in different English-speaking countries (United States, Canada, UK, and Australia). Yet another example is a false cognate, in which a word in one language has an entirely different meaning in another language. A simple example is the German word *gift*, which translates as "poison" in English.

JAPANESE GRAMMAR

The Foreign Service Institute (FSI) ranks various languages from the perspective of an English speaker, and provides an estimate of the number of

hours that are required to achieve a general level of proficiency. The FSI does note that "some language speakers or experts may disagree with the ranking."

Level 5 is the most difficult, and consists of Arabic, Cantonese, Japanese, Korean, and Mandarin. Languages that are "usually more difficult" for native English speakers include Japanese and seven other languages, as shown here:

https://effectivelanguagelearning.com/language-guide/language-difficulty/

This section contains several subsections that describe grammatical features, some of which are unique to Japanese and also pose interesting challenges for NLP.

Japanese Postpositions (Particles)

Instead of prepositions, Japanese uses postpositions (which can occur multiple times in a sentence). Here are some common Japanese postpositions that are written in *Romanji*:

- Ka (a marker for a question)
- Wa (the topic of a sentence)
- Ga (the subject of a sentence)
- O (direct object)
- To (can mean "for" and "and")
- Ni (physical motion toward something)
- E (toward something)

The particle *ka* at the end of a sentence in Japanese indicates a question. A simple example of *ka* is the Romanji sentence *Nan desu ka*, which means "What is it?"

An example of *wa* is the following sentence: *Watashi **wa** Nihon jin desu*, which means "As for me, I'm Japanese." By contrast, the sentence *Watashi **ga** Nihon jin desu*, which means "It is I (not somebody else) who is Japanese."

As you can see, Japanese makes a distinction between the topic of a sentence (with *wa*) versus the subject of a sentence (with *ga*). A Japanese sentence can contain both particles *wa* and *ga*, with the following twist: if a negative fact is expressed about the noun that precedes *ga*, then *ga* is replaced with *wa* and the main verb is written in the negative form. For example, the *Romanji* sentence "I still have not studied *Kanji*" is translated into *Hiragana* as follows:

*Watashi **wa** kanji **wa** mada benkyou shite imasen.*
わたし わ かんじ わ まだ べんきょ して いません

However, Google Translate generates the following humorous translation for the preceding Romanji sentence:

I'm a toilet I haven't done it yet.

By contrast, if you enter the sentence "I have studied *Kanji,*" Google Translate generates the following:

漢字を勉強しました
Kanji o benkyō shimashita

As you can see, the preceding Romanji sentence omits *Watashi wa* and treats the noun *Kanji* as the direct object of the verb "studied."

Yet another use of *ga* is to express "but," as in the sentence "Today I will work, but tomorrow I will play tennis," where the Japanese sentence consists of *Kanji*, *Hiragana*, and *Katakana* (for the word "tennis") and does not contain spaces between words, whereas the *Romanji* translation includes spaces for your convenience:

今日は勉強しますが明日はテニスをします
Kyō wa benkyō shimasu ga ashita wa tenisu o shimasu

As you can see from the preceding examples, there are multiple rules regarding the various combinations of *wa* and *ga* in the same sentence.

An example of a direct object is illustrated in the sentence *Watashi wa terebi* ***o*** *miru*, which means "I watch television" because *o* follows *terebi* (and the latter is derived from "television").

The sentence *Tomodachi* ***to*** *isshyo ni ikimashita* means "I went with my friend" (and other translations are possible as well).

Japanese sentences can contain a combination of *Hiragana*, *Katakana* (just for foreign words), and *Kanji*. For example, the following sentence in *Romanji* means "He loves drinking beer:"

Kare wa biru o nomu no ga daisuki desu.

Although the Japanese verb *nomu* means "to drink," the preceding sentence contains *nomu no ga*, which is called nominalizing a verb.

The preceding sentence written using a combination of *Hiragana*, *Katakana*, and *Kanji* is here (and it's obviously much more complex than *Romanji*):

彼はビールを飲むのが大好きです

If you use Google Translate, the following sentence is generated, which is almost identical to the preceding sentence (the only difference is the direct object particle):

彼わビルお飲むのが大好きです

Consecutive postpositions in Japanese are also possible. For example, the sentence *Nihon e iku **toki ni wa**, sushi o tabemasu* means "when (whenever) [I] go to Japan, I eat sushi," and also contains three consecutive postpositions. The pronoun "I" is in square brackets because the speaker might be one or more different people.

However, multiple consecutive postpositions adhere to rules (i.e., not all combinations are possible), which creates more complexity for non-native Japanese speakers. If a Japanese sentence in *Hiragana* is written without spaces, ambiguity can arise regarding whether to interpret a syllable as a postposition or as part of a word. To illustrate this detail, consider the interesting Japanese sentence "The artist drew a picture," whose translation is clear when it's written as follows:

Gaka ga e o kaita

As you already know, the particles *ga* and *o* are postpositions; the word *e* is a postposition as well, but in this case it's a homonym for the word "picture." This is an example whereby two words with different *Kanji* have the same pronunciation (which can also happen in Mandarin and Cantonese).

Now consider what happens when the preceding *Romanji* sentence is written without any spaces:

Gakagaeokaita

The preceding sentence might appear to have six consecutive postpositions: *ga*, *ka*, *ga*, *e*, *o*, and *ka*. Knowledge of Japanese vocabulary is necessary to parse the preceding sentence correctly. Incidentally, the preceding *Romanji* sentence can also be written as shown here, with no change in meaning, because the postpositions are markers for the grammatical function of the nouns in the sentence:

E o gaka ga kaita

The following link contains an extensive list of Japanese sentences with postpositions:

https://en.wikipedia.org/wiki/Japanese_particles

Ambiguity in Japanese Sentences

Since Japanese does not pluralize nouns, the same word is used for singular as well as plural, which requires contextual information to determine the exact meaning of a Japanese sentence. As a simple illustration, which is discussed

in more detail later in this chapter under the topic of tokenization, here is a Japanese sentence written in *Romanji*, followed by *Hiragana* and *Kanji* (the second and third sentences are from Google Translate):

Watashi wa tomodachi ni hon o agemashita
わたし わ ともだち に ほん お あげました
友達に本をあげた

The preceding sentence can mean any of the following, and the correct interpretation depends on the context of a conversation:

- I gave a book to a friend.
- I gave a book to friends.
- I gave books to a friend.
- I gave books to friends.

Moreover, the context for the words "friend" and "friends" in the Japanese sentence is also ambiguous: they do not indicate whose friends (mine, yours, his, or hers). In fact, the following Japanese sentence is also grammatically correct and ambiguous:

Tomodachi ni hon o agemashita

The preceding sentence does not specify who gave a book (or books) to a friend (or friends), but its context will be clear during a conversation. Incidentally, Japanese people often omit the subject pronoun (unless the sentence becomes ambiguous), so it's more common to see the second sentence (i.e., without *Watashi wa*) instead of the first *Romanji* sentence.

Contrast the earlier Japanese sentence with its counterpart in the romance languages Italian, Spanish, French, Portuguese, and German (some accent marks are missing for some words):

- Italian: *Ho dato un libro a mio amico.*
- Spanish: *[Yo] Le di un libro a mi amigo.*
- Portuguese: *Eu dei um livro para meu amigo.*
- French: *J'ai donne un livre au mon ami.*
- German: *Ich habe ein Buch dem Freund gegeben.*

Notice that the Italian and French sentences use a compound verb whose two parts are consecutive (adjacent), whereas German uses a compound verb in which the second part (the past participle) is at the end of the sentence. However, the Spanish and Portuguese sentences use the simple past (the preterit) form of the verb "to give."

Japanese Nominalization

Nominalizers convert verbs (or even entire sentences) into a noun. Nominalizers resemble a "that" clause in English, and they are useful when speaking about an action as a noun. Japanese has two nominalizers: *no* and *koto ga*.

The nominalizer の (*no*) is required with verbs of perception, such as 見る (to see) and 聞く (to listen). For example, the following sentence mean "I love listening to music," written in *Romanji* in the first sentence, followed by a second sentence that contains a mixture of *Kanji* and *Hiragana*:

Watashi wa ongaku o kiku no ga daisuki desu
私わ音楽おきくのが大好きです

The next three sentences all mean "He loves reading a newspaper," written in *Romanji* and then *Hiragana* and *Kanji*:

Kare wa shimbun o yomu no ga daisuki desu
かれは新聞を読みのがだいすきです
彼わ　しmぶんお読むのが大好きです

The *koto ga* nominalizer, which is the other Japanese nominalizer, is used sentences of the form "have you ever . . ." For example, the following sentence means "Have you (ever) been in Japan?"

にほんにいたことがですか
日本にいたことがですか

Google Translate and Japanese

Google Translate provides a wonderful service, yet sometimes its translations from Japanese to English are incorrect. This point is not intended as a criticism of Google Translate; on the contrary, it's an indication of the complexity of the translation process, even for simple Japanese sentences.

For example, the following sentence means "I love reading a newspaper," but it is incorrectly translated in Google Translate as "I love reading."

彼わしmぶんお読むのが大好きです

Note that the letter "m" in the preceding sentence is due to a limitation of the keyboard in translating ASCII letters to Japanese. As another example, the following (almost identical) sentence is incorrectly translated in Google Translate as "I love to read him" because of the white space that precedes *yomu* (読む):

彼わしmぶんお 読むのが大好きです

If you enter the following sentence in *Hiragana* in Google Translate, the correct *Romanji* is generated:

ゆきがほんお読みました
Yuki ga hon o yomimashita

The preceding sentence means "Yuki read the book." However, Google Translate provides this incorrect English translation:

I just read Yuki.

As another example, the following sentence in *Hiragana* is translated correctly in Google Translate:

すしが　ゆきに　食べられた
Sushi ga yuki ni taberareta
Sushi was eaten by Yuki.

Notice how the particle (postposition) *ni* in the preceding sentence is translated as "by" when it's used in the context of the passive voice in English.

Japanese and Korean

As you learned earlier in this chapter, both Japanese and Korean have postpositions, some of which are similar. There appears to be some degree of comingling among Korean and Japanese, both of which have been influenced by Chinese. The following link contains some interesting details (i.e., a mixture of speculation, conjectures, and some facts) regarding the common aspects of Japanese and Korean:

https://linguistics.stackexchange.com/questions/41/are-the-japanese-and-korean-subject-particles-known-to-be-related-in-any-way-in

Vowel-Optional Languages and Word Direction

Some languages treat vowels as optional in written form, which includes the right-to-left languages Hebrew and Arabic. Native speakers of these languages know the correct vowels to insert in written text so that they can read newspapers and articles correctly. Interestingly, Arabic provides a letter called *sukun*, which looks like a small circle placed between two consecutive consonants to indicate that a vowel is *not* required between the consonants.

As mentioned earlier, most languages are written in a left-to-right fashion. By contrast, Japanese *Kanji* is written from top-to-bottom, and then in a right-to-left fashion. Arabic and Hebrew are written from right-to-left. Moreover,

letters in Arabic words can be vertically "stacked" as they are written in a right-to-left manner.

Arabic also has the concept of a cluster of three consonants that pertain to related concepts. For example, the sequence of three consonants *k-t-b* can be filled in with different vowels, and all of those combinations are related to the verb "read."

Mutating Consonant Spelling

Most languages with alphabets have a single form for each letter in their respective alphabet. However, a letter in Arabic can be written in two, three, or four different ways, depending on its location in a word or sentence. Specifically, the initial, medial, stand-alone, or terminal positions of a letter determine the manner in which the letter is written. For example, the stand-alone Arabic letters k, t, and b are shown here (from left to right):

ك ت ب

However, the Arabic word *kitab* (book) is written in Arabic as follows (notice the absence of any vowels):

كتاب

In Farsi, the word *kitab* is translated in Google Translate as follows (which includes the short vowel "i"):

کتاب

In Urdu the word *kitab* is translated in Google Translate as follows (which includes the long vowel "y" that is pronounced "ee" as in "meek"):

کیتاب

As another example, the following letter is the standalone Arabic letter "s:"

س

However, the Arabic letters that spell "seen" are as follows:

سین

As you can see, only the right-most portion of the letter "s" is displayed in the preceding text, followed by the long vowel "y," and then the letter "n."

Another example is the Arabic word for "lemon:"

لیمون

The preceding Arabic word consists of the letters (from right to left) "l," long "y," "m," "u," and "n" (in left-to-right order). However, the letter "m" in the stand-alone position looks like the following:

م

A complete list of the Arabic alphabet and many additional nuances and details is available online:

https://en.wikipedia.org/wiki/Arabic_alphabet

Expressing Negative Opinions

In addition to the plethora of grammar rules in Japanese, there is another aspect in Japanese culture: how to decide which sentence to use when there is more than one way to express a negative opinion.

For example, the following English sentences are essentially the same in meaning:

I do not think he will go to Tokyo.
I think he will not go to Tokyo.

However, Japanese people view "I do not think" in the first sentence as expressing a personal belief that is stronger than "I think" in the second sentence. In general, it's better to avoid personal opinions when it's possible to do so, and therefore Japanese people will favor the second sentence over the first sentence.

The preceding example illustrates a subtle cultural detail that is not encoded in grammar rules, which can pose a challenge for NLP to generate a translation that is 100% correct from the perspective of a native Japanese speaker.

Now that you have a high-level view of various languages, let's briefly look at the other end of the linguistic spectrum, which is the topic of the next section.

PHONETIC LANGUAGES

Many Indo-European languages and most Asian languages are phonetic, which facilitates the task of learning to pronounce words in an unrelated language. For example, Japanese and Italian are both phonetic, and a native Italian speaker can easily learn to pronounce Japanese words correctly (learning the vocabulary and grammar are far more complex).

By way of comparison, a native English speaker knows that every word in the following list has the same vowel sound, and can also pronounce these words with ease: {I, eye, sigh, why, guy, fly, buy, tie}. However, try explaining to a non-English speaker why all the words in the preceding set have the same vowel sound. By contrast, the following set of Italian words have the same pronunciation for the vowel "o" because no other vowel has the same sound: {*forno*, *bocca*, *giorno*, *dio*, *guasto*, *sporco*, *andiamo*, and *vediamo*}.

Vowels in phonetic languages have one pronunciation, regardless of their location in a word. Consonants *usually* have a single pronunciation, although there are situations in which an adjacent consonant can change the pronunciation of its preceding consonant. For example, in Italian the letter "h" can *modify* the pronunciation of the consonant that appears immediately prior to the letter "h," such as *c'e* and *che*, *ge*, and *ghe* in Italian words.

Thus, the letter "h" can modify the pronunciation of a preceding consonant, whereas in English the combination of "gh" can be silent (which never happens in Italian), such as the words "bough" or "bought," which have entirely different meanings even though they differ by a single consonant.

Many languages have words that contain double consonants, and in phonetic languages, double consonants maintain the same pronunciation while doubling the amount of time to pronounce the two consonants. For example, the double consonant in the Italian words *vacca* and *nanna* merely lengthen the pronunciation of the letter "c" or the letter "n," and sometimes change the meaning of the word (such as *nanna* versus *nana*). Note that Spanish has very few double consonants (the Spanish word for *vaca* is the same as the Italian word *vacca*).

By contrast, English words with double consonants sometimes change the pronunciation of the second consonant, such as the English words "accent," "accident," and "flaccid" (which has two different acceptable pronunciations). Yet, the consecutive occurrences of the letter "k" in "bookkeeper" do not change their pronunciation (and are twice as long as a single "k").

Although the Romance languages Italian, Spanish, and Portuguese (in Portugal) are phonetic, the pronunciation of French words is an exception, even though French is also a Romance language. Some French words have a distinctly nasal pronunciation, and other French words comprise multiple consecutive vowels. For example, the word *oiseau* (bird) is pronounced *wa-ZOE*, which is far different from the spelling of the word. French is also unusual because of the number of words that contain three consecutive vowels. Romance languages are derived from Latin, which includes French, despite the fact that the latter is a non-phonetic language. Interestingly, many

words in Brazilian Portuguese can be pronounced in multiple ways (and not always phonetically).

English is particularly interesting because it's assuredly not a phonetic language, even though it's ranked first in terms of the number of people who speak English. Although there are many non-native-English speakers who can speak English fluently (and grammatically correct), it's sometimes more challenging for non-native English speaking adults to speak English without an accent.

However, when Dutch speakers speak their native language, they have a similar cadence as English. This detail is plausible, because Dutch is a Germanic language, which in turn had a significant influence on old English, just as Latin and Greek have had a significant influence on English.

Although English pronunciation can be a challenge for some people, it pales in comparison to Gaelic, whose pronunciation rules are as complex as they are remarkable. The number of people who speak Gaelic appears to be slowly decreasing, even in Ireland, and perhaps the language is slowly disappearing (to my Irish friends, I simply say, *Éirinn go Brách*.)

Phonemes and Morphemes

In linguistics, a *phoneme* is the smallest unit of speech in a language. Although standard English contains 26 letters, there are 44 phonemes in English. For example, the word "bat" consists of the three phonemes "b," "a," and "t." Phonemes exclude diphthongs and triphthongs that consist of two phonemes and three phonemes, respectively.

Alternatively, a *morpheme* is the smallest meaningful unit of a language. A morpheme carries meaning, whereas a phoneme does not (the latter is a sound unit). Morphemes (and words) are combinations of phonemes, and they can be prefixes, syllables, or prefixes of words. For example, "disappeared" consists of the three morphemes "dis." "appear," and "ed."

English Words of Greek and Latin Origin

There are many interesting words in English whose roots are in Greek. Words with the suffix "ology," which means "study of," are of Greek origin. Examples include biology ("bios" = "life"), anthropology ("anthros" = "man"), anthropomorphism ("morphism" = "to change") to ascribe human characteristics to inanimate objects or nonhumans (such as pets).

If you love languages, then you are a linguaphile, and if you love words, then you are a logophile, which are derived from the Greek words *lingua*, *philia*, and *logos* that mean language, love, and words, respectively.

The English word "telephone" is derived from the Greek words *telos* (far away) and *phonos* (to speak), whose combination means "to speak far away" (which is a suitable combination for the word telephone).

MULTIPLE WAYS TO PRONOUNCE CONSONANTS

The pronunciation of letters such as "c" and "g" in Italian words depends on the vowel that follows these letters. In German, when the letters "b," "d," and "g" appear at the end of a word, they are pronounced "p," "t," and "k," respectively. The following subsections illustrate the changes in the pronunciation of the consonants in words in various languages.

The Letter "j" in Various Languages

The letter "j" in English words has two sounds: one sounds like the letter "j" in "John," and the other sounds like the letter "j" in *je suis*, such as the letter "g" in the English word "mirage" (which is a word borrowed from French). The latter pronunciation of the letter "j" also occurs in Portuguese, Russian, and in parts of Argentina that is close to the border with Brazil. However, the more common pronunciation of "j" in Spanish is similar to "ch" in the German word *achtung.* As an added twist, some Spanish speakers pronounce the word *yo* that sounds similar to "Joe."

In addition, the double "l" in *calle* is pronounced like "KA-yeh" in Spanish-speaking countries, with the exception of Argentina (perhaps because it's near Brazil), where the Spanish word *calle* sounds like "KA-jeh" (pronounced like the "j" in *je suis*).

However, the "j" sound (as in "mirage") does not exist in standard Spanish that is spoken in other regions, nor does this sound exist in Italian. In fact, the letter "j" is not part of the original Italian alphabet. In Italian, the correct pronunciation of the combination "ga," "go," and "gu" sounds like the letter "g" in the English word "gap." whereas "ge," "gi," and "gio" sound like the letter "j" in "John." Thus, the correct pronunciation of the Italian word *parmigiano* sounds like "par-mee-JA-noh," where "JA" sounds like "jacket."

"Hard" versus "Soft" Consonant Sounds

For all Italian words containing the letter "c" or "g" that are immediately followed by the letter "h," the pronunciation is modified when the vowel that immediately follows the letter "h" is either "e" or "i."

Specifically, the combinations *ci* and *c'e* are pronounced like "ch" in "cheek" and "check," respectively, whereas the combinations "chi" and "che" are pronounced like "k" in "keep" and "kettle," respectively. Hence, the word *cecci* (chickpeas) is pronounced with two consecutive "ch" sounds as in "check" or "CHE-chee." The same pattern applies to "gi" and "ge" versus "ghi" and "ghe."

The preceding paragraph brings us to the proper noun Giorgio (George), which is sometimes pronounced as four distinct syllables ("gee-OR-gee-OH") by non-native Italian speakers, in an earnest attempt at making the correct pronunciation. Despite all of the preceding rules, Italians pronounce Giorgio as two syllables "JYOR-jo" (which admittedly is not entirely phonetic).

In addition, some German consonants also undergo changes in pronunciation: words that end in the letters "b," "d," or "g" are pronounced as "p," "t," and "k," respectively. Hence, "ab" (a prefix), "todd" (death), and "berg" (mountain) are pronounced as "ap," "tot," and "berk." Thus, *Bergman* is translated as "mountain man."

Another interesting grammatical scenario involves pairs of voiceless and voiced consonants, which is discussed in the next section.

"Ess," "Zee," and "Sh" Sounds

Some languages have pronunciation rules involving consecutive consonants that do not exist in English. For example, the letter "s" can be voiceless (as in "seed") or a voiced consonant that sounds like "z" (as in "boys"). In Italian, the letter "s" can also be a voiceless or a voiced consonant. The consonant "n" in English and Italian is a voiced consonant.

In addition, Italian also has a grammatical rule that a voiceless consonant that is immediately followed by a voiced consonant must be "converted" to a voiced consonant as well. Consequently, some Italians pronounce the English word "snow" as "znow," and "slice" is pronounced as "zlice."

However, the letter "z" in the initial position of Italian words (such as "zio" or "zia") is pronounced like "ds" in the word "ads." Moreover, an "s" between two vowels is pronounced as a voiced "s," whereas a double "z" is pronounced as a voiceless "s," which means that "pizza" is *not* pronounced "PEET-suh." In Spanish, the letter "z" is always pronounced as a voiceless "s," and a voiced counterpart of the letter "z" does not exist in Spanish.

The pair of consonants "st" and "sp" are pronounced with a voiced "s," as in "street" or "spa." However, in Naples, the letter "s" in the pair "sp" sounds like "sh" as in "sheep." Hence, Neapolitans say "SHpaghetti" instead

of "spaghetti." Coincidentally, the pair of consonants "st" and "sp" in Arabic words rarely (never?) have an intermediate consonant, which is indicated by the letter *sukun*.

Spanish also undergoes some changes when the article *la* precedes a feminine noun with initial letter "a." For example, *agua* is feminine, but instead of *la agua*, the correct sequence in Spanish is *el agua*.

In Venezuela, many people speak Criollo, whose pronunciation differs from spoken Spanish in the other south American countries. One significant difference in Criollo involves the silent letter "s" at the end of a word. To be more precise, the letter "s" in the terminal position is pronounced like the letter "h" in the word "hot."

For example, the sentence *El eta alla en la equina con do uiqui* in Criollo is equivalent to the Spanish sentence *El esta alla en la esquina con dos uisquis* (the accent marks have been omitted). Keep in mind that Spanish refers to someone from Spain, whereas Castellano is the language that is spoken by people in Hispanic countries.

Three Consecutive Consonants

English has various word that contain three consecutive consonants, such as "street," "straight," and "spray." However, some languages avoid the occurrence of three consecutive consonants in a word, sometimes by changing the preceding article ("a" or "the") that precedes such a word.

For example, Italian uses *la* for feminine (*la casa*) and *il* for masculine (*il libro*), and *lo* for masculine words that start with two consecutive consonants. Thus, *lo sport* is used instead of *il sport*, and *lo stile* instead of *il stile*, thereby avoiding the three consecutive consonants "lsp" and "lst."

In addition, *lo zucchero* and *lo zio* are correct, even though the sequence *il zucccero* involves a two-consonant sequence "lz." However, the initial rule still stands, because an initial "z" in Italian words is pronounced like "ds" in the English word "ads," which in turn sounds like "dz" in English. Thus, *lo zucchero* is correct because it avoids the three consecutive consonants "ldz" resulting from *il zucchero*, and similarly for *lo zio* and *lo zaino*.

Arabic also avoids three consecutive consonants. However, since vowels are optional and frequently omitted in written Arabic, fluent readers can silently insert vowels between consecutive consonants. In some cases of consecutive consonants, there is no intermediate vowel, which readers will know due to the presence of the *sukun* symbol, which is a small open circle that appears between the pair of consecutive consonants.

Diphthongs and Triphthongs in English

Instead of "pure" vowels, English words typically contain diphthongs (and also some triphthongs) that are combinations of phonetic vowels. Here are some examples of English vowels and their phonetic counterparts:

- "a" is phonetic "e" + "i" (when it sounds like "pay")
- "i" is phonetic "a" + "i" (when it sounds like "eye")
- "o" is phonetic "o" + "u" (when it sounds like "foe")
- "u" is phonetic "i" + "u" (when it sounds like "you")
- "y" is phonetic "y" + "a" + "i" (when it sounds like "why")

Thus, the English vowels "a," "o," and "u" are diphthongs, and the English letter "y" is actually a triphthong. The lone exception is the letter "e," which is a phonetic "i" when it rhymes with "pea."

Semi-Vowels in English

English has consonants that can sometimes function as semi-vowels, such as the letter "m" in "prism," the letter "l" in "castle," and the letter "r" in "center." There are other languages with consonants that can function as semi-vowels. One humorous (and challenging) example is the following Czech sentence that contains the semi-vowel "r" and no vowels ("Stick finger through throat"):

Strč prst skrz krk

Challenging English Sounds

Some sounds in English are difficult for non-native English speakers, generally for people whose native language is phonetic. For example, the combination "or" and "er" is simple when they are pronounced like "or" in the word "for" and "er" in "feral." However, the "er" sound in the following words can be more challenging for non-native English speakers:

- world
- hurt
- her
- earnest
- girth

Notice that every vowel is followed by "r" in the preceding list of words, and yet all of words have the same "er" sound.

Other difficult combinations are "th" in words such as "three," which is approximated as "tree" with a "trilled r" sound, and "third" is pronounced as

"turd," and also with a "trilled r" sound. The sound of "th" in the word "then" can also be difficult, and it's sometimes approximated as "den."

As an aside, during a late evening talk show, Charlize Theron once explained that her last name is pronounced like "tron," with a strong trilled "r" sound, instead of "thur-ON," which makes perfect sense to people who speak phonetic languages.

Silent letters can be especially challenging for non-native English speakers. People who speak phonetic languages will sometimes say "PLUM-ber" instead of "plummer." Sometimes the reverse happens: they pronounce a letter that is actually silent, such as the letter "h:" they will say "HON-est" instead of "ON-est." Another fun set of English words and the variations in their pronunciation: "but" versus "put" versus "putrid" versus "purple;" "low" versus "plow," and "row" (a boat) versus "row" (an argument). Now try explaining the logic behind the pronunciation of the following words containing vowels that sound like "ow," "oh," "uh," "oo," "er," and "yoo" to a non-English speaker:

- plough
- bough
- rough
- through
- furlough
- fur
- eunuch

Another combination that is challenging (and sometimes for native-English speakers as well) is the combination of "th" and "z" sound in a word such as "youths." Sometimes you will hear "youths" pronounced as "yoots," depending on the speaker's location in the United States.

English in Canada, UK, Australia, and the United States

There are some relatively minor differences in the spelling of words in English spoken in the United States versus other English-speaking countries. One such difference is replacing "ou" with "o," as in "color," "favor," and "neighbor" instead of "colour," "favour," and "neighbor." Another difference involves replacing "ll" or "pp" with a single "l" or "p," such as "traveled" and "worshiped" instead of "travelled" and "worshipped."

Other simple changes are "tire" instead of "tyre" (UK), "trunk" instead of "boot" (UK), "aluminum" instead of "aluminium," and "eraser" instead of "rubber." Changes in pronunciation include "PRY-vacy" instead of "PRIV-acy" and "a-LOO-minum" instead of "a-loo-MIN-ium."

This concludes the high-level introduction to languages, and by now you are probably saturated with the bewildering variety of grammatical rules, word order, gender, plural forms, and pronunciation rules. There are thousands of human languages, most of which have not been discussed in this introduction.

The main purpose of speech is communication between people, regardless of the grammatical errors or accents of the speaker. Now that you have an understanding of the many nuances of human languages, you now have a greater understanding of the various challenges and nuances facing NLP.

ENGLISH PRONOUNS AND PREPOSITIONS

If you have struggled with the correct combination of pronouns and prepositions in English, there is a simple rule to remember: a subject pronoun in English (I, he, she, and so forth) can *never* follow a preposition (such as to, for, between, and so forth).

The following table displays subject pronouns, direct object pronouns, and indirect object pronouns in English:

Table 2.2 Subject, Direct Object, and Indirect Object Pronouns in English.

Subject	Direct	Indirect
I	me	me
you	you	you
he/she/it	him/her/it	him/her/it
we	us	us
you	you	you
they	them	them

Based on the contents of Table 2.2, which of the following fragments is correct?

1. "you and I disagree"
2. "between you and I"
3. "between you and me"
4. "between you and him"
5. "between we and him"

6. "between us and him"
7. "between we and they"
8. "him and I went to the store"

[Correct answers: 1, 3, 4, and 6]

If you replace the word "between" with the word "for" or "to" (or any other English preposition) in the preceding list, *the list of correct answers is the same*. A list of English prepositions is available online:

https://www.englishclub.com/grammar/prepositions-list.htm

This concludes our discussion of languages.

WHAT IS NLP?

NLP is an important branch of AI that pertains to processing human languages with machines. In fact, 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 multiple 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.

With regard to language translation, Facebook has created an impressive model called the M2M model, which was trained on more than 2,000 languages and provides a 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 results.

In particular, the transformer architecture (which relies on the concept of attention) has eclipsed earlier neural network architectures. The transformer architecture is the basis for BERT, which is a pretrained NLP model with 1.5 billion parameters, along with numerous other pretrained models that are based (directly or indirectly) on BERT. Chapter 11 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 11), 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 the 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 available online:

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. NLP can help you analyze documents that contain structured and unstructured data (or a combination of both types of data).

The Evolution of NLP

NLP has undergone many changes since the mid-20th 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, 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) spans several decades and primarily focused on rule-based systems, which means that they used largely 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 (but not with modern NLP methods). This era of NLP also performed various 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 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 classifying images. Researchers then learned how to use CNNs 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 (LSTMs), 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 here:

- 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 see numerous books about this topic, it's a significant subset of NLP. In high-level terms, NLU attempts to understand human language in determining the context of a text string or document. NLU addresses various NLP tasks, such as sentiment analysis and topic classification. Another 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?

As you undoubtedly know, 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.

Another 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 11) 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, or the contents of a Webpage. 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.

As you can probably surmise, 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) algorithm that determines the word frequency in documents. The BoW algorithm is explained in Chapter 5, along with code samples for the BoW algorithm in Chapter 6.

Text summarization is related to text classification, and it's described in Chapter 9 in the section that discusses the recommender system.

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 a larger company by an even larger company. Information extraction involves 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) that is discussed toward the end of this chapter.

Information extraction 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 lowercase, and then splitting sentences into tokens (words). Fortunately, the Beautiful-Soup Python library can easily perform each of the preceding tasks.

Another area of great interest in NLP is the proliferation of chatbots, which interact with users to provide information (such as directions or hours of operation) or perform specific tasks (make reservations, book hotels, or rent cars).

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.*
- The NYT article is available online:

https://www.nytimes.com/1983/04/28/business/technology-the-computer-as-translator.html

As another 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. The dismal 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 whose accuracy 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 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 here:

- 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

Most of the items in the preceding list are discussed in Chapter 4; in some cases, there are associated Python code samples in Chapter 5 and 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) stop word removal
- (3) stemming
- (3) lemmatization

The number in parentheses 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. 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 (or the task of converting categorical (character) data into a numeric counterpart).

Although *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 accent 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, although 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 Named Entity Recognition (NER), 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. Another complicating factor is the existence of multiple alphabets in Japanese, 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 sentence, which means "I gave a book to my friend" in English:

watashiwatomodachinihonoagemashita

The translation (which is almost correct) 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 sentence, but 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

The preceding sentence starts with the Kanji character 私 that is the correct translation for *watashi*.

Mandarin and Cantonese are two more languages that involve complicated tokenization. Both of these languages are tonal, and they use pictographs

instead of an alphabets. Mandarin can be written in Pinyin, which is the romanization of the sounds in Mandarin, along with 4 digits to indicate the specific tone for each syllable (the neutral sound does not have a tone). Mandarin has 6 tones, of which 4 are commonly used, whereas Cantonese has 9 tones (but does not have a 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 in the preceding group 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 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 to see the following translation:

how many kids do you have

The preceding translation is quite impressive, when you consider that the tones were omitted, which 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 German. In Chapter 4, there are 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: mea-
sure 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:

```
1 a
2 based
1 between
1 changes
1 determining
2 distance
1 edit
1 endings
1 for
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, but they can adversely models that use word context to detect semantic meaning. 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," "her," and so forth). However, a list of stop words for the text in a marketing-related Website is probably different from such a list for a technical Website. Fortunately, Sklearn enables you to specify your own list of stop words via the hyperparameter `stop_words`.

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. A stemmer operates on individual words without any context for those words. Stemming truncates the ends of words, which means that "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 versus 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), and a plural form with the letter "i" (such as cactus/cacti and appendix/appendices).

However, 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
- SnowballStemmer (more than 10 languages)
- ISRIStemmer (Arabic)
- RSLPS Stemmer (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 the results can be odd). The other three stemmers 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 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. A list of prefixes in the English language is available online:

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 also online:

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, and universe.

The stem for the four preceding words is *universe*, 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

Both techniques are designed for "recall," whereas precision tends to suffer. 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, the stem might not be an actual word, whereas the lemma *is* an actual language word. In general, 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 online:

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 following subsections provide more details regarding POS, some POS techniques, and also NER (Named Entity Recognition).

POS Tagging

POS refers to 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.

When the meaning of a word is *overloaded*, its function 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 a document).

POS Tagging Techniques

The major POS tagging techniques (followed by brief descriptions) are as follows:

- Lexical-Based Methods
- Rule-Based Methods
- Probabilistic Methods
- Deep Learning Methods

Lexical-Based Methods assign POS tags based on the most frequently occurring 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. Alternatively, 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.

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 into 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. Moreover, there are various predefined named entity types, such as `PERSON` (people, including fictional), `ORG` (companies, agencies, institutions), and `GPE` (countries, cities, states). A complete list of named entity types is here:

https://spacy.io/api/annotation

Although NER is very useful, there are situations in which NER can produce incorrect results, such as

- an 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. Chapter 4 contains an example of performing a chunking operation on some text.

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 hidden Markov models (HMM), decision trees, maximum entropy models, support vector machines (SVM), and conditional random fields (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 refers to 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 reemerge as topics.

There are several algorithms available for topic modeling, some of which are in the following list:

- LDA (Latent Dirichlet Allocation)
- LSA (Latent Semantic Analysis)
- Correlated Topic Modeling

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 nondeterministic algorithm that produces different topics each time the algorithm is invoked.

By way of analogy, LDA resembles the kMeans algorithm (discussed later in this book). LDA requires that you specify a value for the number of topics, just as kMeans requires a value for the number of clusters. 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 reassigned 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 sense. The longer the document, the greater the possibility that the 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 of 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` (term frequency-inverse document frequency) values of words in a corpus (discussed in Chapter 4) and BERT models (discussed in Chapter 11). 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, which are 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 word 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. Chapter 9 contains additional details regarding text summarization and sentiment analysis.

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 obtained a brief introduction to NLP applications, NLP use cases, NLU, and NLG. Then you learned about concepts such as word sense disambiguation, text normalization, tokenization, stemming, lemmatization, and the removal of stop words. Finally, you learned about POS and NER and topic modeling in NLP.

CHAPTER 3

NLP CONCEPTS (II)

This chapter discusses NLP concepts, such as word relevance, vectorization, basic NLP algorithms, language models, and word embeddings. Please keep in mind that this chapter focuses on NLP concepts, and Chapters 4 and 5 contain Python-based code samples that illustrate many of the concepts that are discussed in this chapter as well as 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. If you are not interested in the more theoretical underpinnings of machine learning algorithms, you can skim through this section of the chapter and perhaps 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 a 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 would suggest that the word could have higher relevance. Contrastingly, 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 (although 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 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 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. We need to consider 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. As you learned in the previous chapter, German supports the declension of articles and adjectives (discussed in Chapter 3). In the following example of two identical German sentences, notice that the word order is reversed in the second sentence (see Chapter 3 for an explanation):

*Der Mann sieht **den** Hund.*
***Den** Hund sieht der Mann.*

One approach to managing 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 (discussed in Appendix A), word2vec with the cosine similarity (the latter is discussed in this chapter), and the Latent Dirichlet Analysis (LDA, which is discussed later in this chapter) 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 the Transformer, LSTMs (Long Short Term Memory), and VAEs (Variational Auto Encoders), but the latter two are beyond the scope of this book. You can even use the kMeans clustering algorithm in machine learning to perform sentence similarity analysis. Yet another technique is the universal sentence encoder, as discussed in the next section.

Sentence Encoders

Pretrained sentence encoders for sentences are the counterparts of word2vec and GloVe 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.

Google created the Universal Sentence Encoder that encodes text into high dimensional vectors that can be used for various natural language tasks, and the pretrained 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 a deep

averaging network (DAN), which has lower accuracy. There are 11 models available that have been trained to perform different tasks.

WORKING WITH DOCUMENTS

Two tasks pertaining to documents involve document classification (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 subsentence 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 support vector machines (SVMs) and Naive Bayes.

Document Similarity (doc2vec)

There are several algorithms for determining document similarity, including Jaccard (see Appendix A), doc2vec (discussed in this section), and BERT (discussed in Chapter 11).

The doc2vec algorithm is an unsupervised algorithm that converts documents into corresponding vectors 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 available online:

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.

However, the use of term frequency or tf-idf to determine semantically related documents does not work for the two documents that contain the words tigers and lions. In this case, we need to use word2vec or doc2vec (a technique that involves word2vec).

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 11) 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.

In Chapter 5, we give an example of performing document similarity using the gensim Python library.

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. The cosine similarity of two vectors is the cosine of the angle between the two vectors; when this 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.

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).

Please keep in mind that the purpose of this distinction is to 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 approach, but they do not capture the context of words, nor do they track the grammatical aspects (such as subject, verb, or object) 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 (all of which involve integer-valued vector), as listed here:

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 Chapter 5, we give 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. (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 = the presence and 0 = the absence of a word. This is called a *one-hot document vectorization*. Although this does not preserve word order in the input text, it's easy to interpret and easy to generate.

As an illustration, the following technique performs document vectorization by performing the following steps:

```
• Determine the 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 three 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 1x11 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)

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 10x10 identity matrix is an OHE 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 5 contains a code sample that illustrates the OHE of a vocabulary.

An OHE relies on a BoW representation of the words in a vocabulary. An 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 has 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," "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 called *out of vocabulary* (OOV) 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, it also 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 do not discuss these word encoders, but information regarding the text encoders (along 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 (Bag of Words) algorithm generates an array with the number of occurrences

in the document of each dictionary word. The advantages of the BoW algorithm include simplicity and an easy way to see the frequency of each word in a document. The BoW algorithm is essentially an n-gram model with n=1 (n-grams are discussed later in this chapter).

However, the BoW algorithm does not maintain any word order and no form of context, and in the case of multiple documents, the BoW algorithm 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 1x5 vector for the dictionary is (this, is, a, short, sentence). Hence, the phrase "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 1xN vector of zeroes and positive integers that indicate the number of times that words appear in a sentence.

The Sklearn library (not discussed in this chapter) provides a `CountVectorizer` class that implements the BoW algorithm. The `CountVectorizer` class that 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 the 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, with the corresponding code in a later chapter, 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]]
```

BoW models also 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 "bigram" 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 phonemes, syllables, letters, or 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 (bigrams): (this, is), (is, a), (a, sentence)

Example #2: "This is a sentence" has the following 3-grams (trigrams): (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 here:

```
{'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" = ¼
```

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 the BoW algorithm 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, the tf-idf algorithm 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 on 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 (albeit contrived) example shows you how to use term frequency to calculate numeric vectors associated with three documents in order to determine which pair of documents are more closely related.

Let's suppose that `doc1`, `doc2`, and `doc3` contain the words "beer," "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
-------------------------
```

Now 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) is a proportion of 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 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 sort of 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, the BoW and tf-idf algorithms differ from word embeddings (discussed later in this chapter) in two important ways:

1. The BoW and tf-idf algorithms calculate one number per word, whereas word embeddings create one vector per word.

2. The BoW and tf-idf algorithms 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.

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 here:

```
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 calculating the word relevance of individual words, 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 beautiful racing horse." Would you accept the phrase "strong beautiful 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). The Transformer-based architecture is discussed in the final chapter of this book.

BoW models also 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.

Pointwise Mutual Information (PMI)

PMI is an alternative to the tf-idf algorithm, 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. Here, we discuss the distributional hypothesis regarding the context of words. An important is that 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, we discuss 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 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")

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. 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 vary (as you might expect). One of the techniques for training the BERT model is called NSP, which is an initialism for Next Sentence Prediction. More details regarding NSP are in Chapter 11.

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. 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 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." The attention mechanism is used in transformers, both of which are discussed in Chapter 11.

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 `[v1,v2]`. Then the difference between these two vectors is `[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 `u` and `10*v`, which we cannot accomplish if we use the Euclidean metric.

The solution is to calculate 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), 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 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 (antonyms?).

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 Appendix A.

TEXT VECTORIZATION (A.K.A. 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. There are various publicly available word embeddings available, so you don't need to be concerned about generating those vectors (unless you have a custom dictionary).

Depending on your task, you might be able to work with small vectors, 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 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. 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 various other tasks that are discussed in this chapter, such as normalization, stop word removal, and lemmatization.

As you will see later in this chapter, word2vec (developed in 2013) 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

The 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. However, 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 word2vec.

Fortunately, word embeddings are useful for analyzing text data in many languages (i.e., not just English text). Moreover, there are pre-trained word embeddings available, and it's worthwhile performing an analysis of 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
- other lesser-known algorithms

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-gram) 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 the 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:

- learn the distributed representations for words
- focus on the meaning of words
- attempt to understand meaning and semantic relationships among words
- handle unlabeled data
- works similarly to deep learning 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 as follows:

("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 detect 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.

Later in this chapter you will see the neural networks for CBoW and skip-grams.

The Principle Behind 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 (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," "appalled," and "astonished" are usually used in a similar context. As you saw earlier in this chapter, this means that "The meaning of a word can be inferred by the company it keeps."

word2vec is well-suited for sentiment analysis based on a corpus of user-based reviews (such as movies or 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 the following:

- genes, code, likes, playlists, social media graphs
- other verbal or symbolic series in which patterns may be discerned

Word2vec can also be used for labeled data as well as unlabeled data. 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 the skip-gram (default) or Continuous Bag of Words. The training algorithm is hierarchical softmax (default) or negative sampling.

The *minimum word count* helps limit the size of the vocabulary to meaningful words. Any word that does not occur at least this many times across all documents is ignored.

Reasonable values could be between 10 and 100. In this case, since each movie occurs 30 times, we set the minimum word count to 40, to avoid attaching too much importance to individual movie titles. This resulted in an overall vocabulary size of around 15,000 words. Higher values also help limit run time.

There is more information about backward error propagation in word2vec, with details for CBoW and skip-grams, available online:

http://www.claudiobellei.com/2018/01/06/backprop-word2vec/

Limitations of word2vec

Word2vec provides only one word embedding per word. Moreover, 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 this limitation 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 (which is one type of word2vec model) involves a feed forward neural network that determines word embeddings. The neural network consists of the following:

- an input layer
- a hidden layer (no activation function)
- 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 3.1 shows the CBoW architecture and Figure 3.2 in the next section displays the skip-grams architecture, both of which are shallow neural networks.

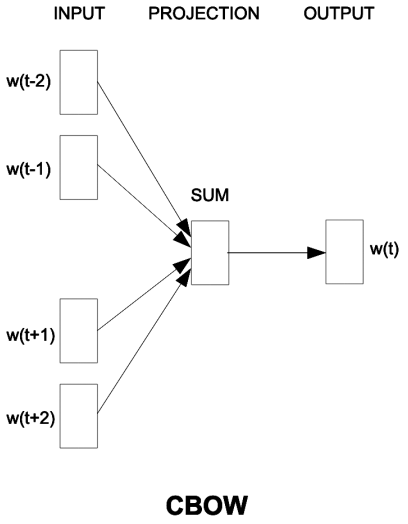

FIGURE 3.1 The CBoW architecture.

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?

As you learned in the previous section, 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 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 presented in the next section. 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.

Skip-gram Example

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 unigrams, bigrams, and trigrams, respectively.

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 here:

```
[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

```
[(ate,the,big), (ate,big,mouse), (ate,the,cheese)]
```

The Skip-gram Architecture

Figure 3.2 shows the skip-gram architecture that is based on a shallow neural network.

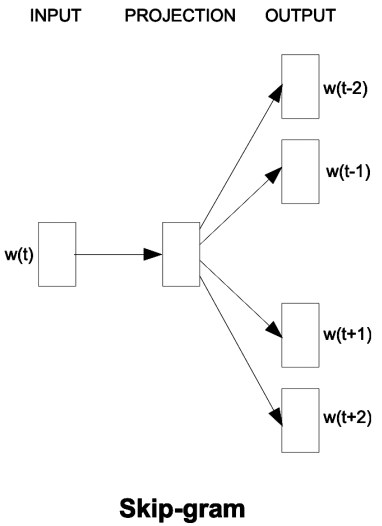

FIGURE 3.2 The skip-gram architecture.

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)]

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, perform an online search for articles that explain neural networks.

Figure 3.2 shows the skip-gram architecture consists of the following components:

- the input layer is a single word
- a hidden layer
- 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:

- There is no bias term.
- There is no activation function between the input layer and the hidden layer.
- There is a softmax activation function from the hidden layer to the output layer.
- The input layer and the output layer have the same size.

If you are familiar with convolutional neural networks (CNNs), then you already know that the softmax activation function is applied between the right-most 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 1x300 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:

- subsample frequent words (which decreases the number of training examples)
- modify the optimization objective via "negative sampling"

These two techniques reduce the computational complexity and improve the quality of the results.

The intuition 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 how to find the bigrams of a simple sentence, and this information is reproduced here:

```
[(ate,the), (ate,big), (ate,mouse), (ate, the),
(ate,cheese)]
```

The previous set of bigrams includes stop words that 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 × context.
2. Factor the matrix into a matrix of dimensionality word × 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, which 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).

The 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 (developed at Stanford University) 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)

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/

The *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 fastText 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 `train_unsupervised()` method in fastText uses the skip-gram model in order 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 indicate that the pair of words are more similar in meaning.

The fastText library 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. The fastText vectors are more accurate than word2vec vectors based on various criteria. Moreover, fastText can handle OOV words and the sub-word n-grams corresponding to "intuition" (shown for multiple languages).

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 embeddings 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 (bigrams) for word2vec
  - Key point: only one embedding for each word (regardless of its context)
- Group 3—Contextual word representation (such as BERT)
  - transformer architecture (no CNNs/RNNs/LSTMs)
  - Pays "attention" to ALL the 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. 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:

- Each document consists of a mixture of topics.
- 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 as follows:

- Latent Dirichlet Analysis (LDA)
- Latent Semantic Indexing (LSI)
- Latent Semantic Analysis (LSA)

Details regarding LDA are in the next section, and you can perform an Internet search for details regarding the LSI and LSA.

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 that are discussed in Appendix B.

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 topic

LDA uses the JS (Jenson-Shannon) metric, which is based on the JS divergence, and the latter is based on the KL divergence (more information about these topics in an appendix). Since the 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. As you might have already surmised, 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 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 versus Topic Modeling

Text classification involves supervised learning on documents or articles with a known set of labels and classifies the 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 discussed in Appendix B) 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 (described elsewhere in this chapter).

According to some NLP experts, 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:

- Symbol 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, some NLP experts believe 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 pretrained 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 motivating principle behind 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 it 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 latent semantic analysis (LSA) 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 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.

The usefulness of a VSM model is due to its basis in 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 document terms. Unfortunately, documents with 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 terms are statistically independent. Even so, some of the disadvantages can be ameliorated by using techniques such as Singular Value Decomposition (SVD).

NLP AND TEXT MINING

In high-level terms, text mining performs an analysis of large amounts of unstructured data in order to find patterns in that data. Text mining tasks involve finding keywords, topics, and patterns. The general sequence of steps (tasks) is shown here:

- preprocessing
- text transformation
- attribute selection
- visualization
- evaluation

Text mining also 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 and product issues. Applications of text mining include spam detection, sentiment analysis, e-commerce, and customer segmentation. The NLTK (Natural Language Tool Kit) library is well-suited for text mining tasks, and you will see code samples in Chapter 4.

Text Extraction Preprocessing and N-Grams

As you learned earlier in this chapter, 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 (Natural Language Understanding), 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, 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

For 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.

However, *relation classification* is the task of identifying the semantic relation holding between two nominal entities in text. As you might have surmised, 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 BLUE score is typically published alongside NLP models, its inclusion has become standard practice.

However, BLEU was created 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) and determines the number of n-grams of the reference translation that 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 other alternatives to BLEU.

SUMMARY

This chapter started with a quick 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 obtained an introduction to distance metrics, such as the cosine similarity (for measuring the distance between two vectors) and document similarity. In addition, you learned about the concepts of vector space models (VSMs) and topic modeling.

CHAPTER 4

ALGORITHMS AND TOOLKITS (I)

This chapter contains Python code samples that illustrate various NLP concepts in the previous two chapters. Since those chapters contain sufficient NLP-related theory, this chapter focuses almost exclusively on code samples. The majority of the code samples in this chapter involve NLTK (Natural Language Toolkit), along with several code samples that are based on Gensim.

The first section (approximately two-thirds of this chapter) introduces NLTK and code samples that use NLTK with BoW, stemmers, and lemmatization. You will also see some examples of NLTK with Wordnet, lxml, XPath (not discussed in this book), n-grams, and skip-grams.

The second section introduces GloVe and Gensim, which are very useful NLP Python libraries, along with some code samples.

WHAT IS NLTK?

NLTK is an open source Python library specifically for NLP-related tasks. Although NLTK does not provide state-of-the-art performance, it does provide a variety of solutions to many NLP tasks. This library was developed in 2002, and its home page is available online:

www.nltk.org

NLTK provides support for many NLP-related tasks, such as stemmers, tokenization (words, sentences, and documents), lemmatization, chunking, and grammars. In particular, NLTK supports the SnowballStemmers that creates non-English stemmers for more than 10 languages: Danish, Dutch, English, French, German, Hungarian, Italian, Norwegian, Portuguese, Romanian, Russian, Spanish, and Swedish.

NLTK also supports n-grams, skip-grams, BoW, word2vec, Parts Of Speech (POS), and Named Entity Recognition (NER). In fact, NLTK enables you to define your own custom grammars and then parse sentences to determine if their structure conforms to a custom grammar.

NLTK is well-suited for various NLP tasks, such as recommendation systems and sentiment analysis, both of which are discussed in more detail in Chapter 6. NLTK also supports Wordnet, which enables you to find words (via Sysnet) and their homonyms and synonyms.

NLTK AND BoW

Listing 4.1 shows the contents of `ntk_bow.py` that illustrate how to implement BoW in NLTK. This code sample involves regular expressions, also known as RegExs. If you are unfamiliar with RegExs, you can either "comment out" the two code snippets that involve regular expressions, or you can read the appendix that discusses this topic.

LISTING 4.1: nltk_bow.py

```
import nltk
import numpy as np
import re

text = 'the SF weather is hot and the LA weather is hotter'
ds = nltk.sent_tokenize(text)

# clean the words in the dataset:
for i in range(len(ds)):
  ds[i] = ds[i].lower()
  ds[i] = re.sub(r'\W', ' ', ds[i])
  ds[i] = re.sub(r'\s+', ' ', ds[i])

print("cleaned dataset:")
print(ds)
print()

# construct BoW model:
word2count = {}
```

```
for data in ds:
  words = nltk.word_tokenize(data)
  for word in words:
    if word not in word2count.keys():
      word2count[word] = 1
    else:
      word2count[word] += 1

# display word/frequency counts:
for word, freq in word2count.items():
  print(f'word: {word:8} frequency: {freq:3d}')
```

Listing 4.1 initializes the variable text with a string and then initializes the variable `ds` with the tokens of text. Next, a `for` loop iterates through the tokens and converts them to lowercase, removes all nonalphabetic characters, and replaces multiple whitespaces with a single whitespace.

Next, the cleaned data is displayed, followed by a loop that constructs a BoW model based on the tokens in the variable `ds`. Note that multiple occurrences of a token are taken into account when populating the dictionary word2count. The final portion of Listing 4.1 displays each word and its frequency. Launch the code in Listing 4.1 and you will see the following output:

```
cleaned dataset:
['the sf weather is hot and the la weather is hotter']

word: the      frequency:  2
word: sf       frequency:  1
word: weather  frequency:  2
word: is       frequency:  2
word: hot      frequency:  1
word: and      frequency:  1
word: la       frequency:  1
word: hotter   frequency:  1
```

NLTK AND STEMMERS

Listing 4.2 shows the contents of `stem_documents1.py` that illustrate how to perform stemming on a sentence with a `PorterStemmer`.

LISTING 4.2: stem_documents1.py

```
import nltk
from nltk.stem import PorterStemmer
from nltk.tokenize import sent_tokenize, word_tokenize

# read file contents:
file = open("data-science-wiki.txt")
my_lines_list = file.readlines()
#print("my_lines_list:")
#print(my_lines_list)

porter = PorterStemmer()

def stemSentence(sentence):
  token_words = word_tokenize(sentence)
  token_words
  stem_sentence = []

  for word in token_words:
    stem_sentence.append(porter.stem(word))
    stem_sentence.append(" ")

  return "".join(stem_sentence)

def saveStemmedLines():
  stem_file = open("stem-data-science-wiki.txt",mode =
                                    "a+",encoding = "utf-8")
  for line in my_lines_list:
    stem_sentence = stemSentence(line)
    stem_file.write(stem_sentence)
  stem_file.close()

print(my_lines_list[0])
print("Stemmed sentence:")
x = stemSentence(my_lines_list[0])
print(x)
```

Listing 4.2 starts by reading the contents of a text file into the variable `my_lines_list`. Next, the variable `porter` is initialized as an instance of the `PorterStemmer` class that is available in NLTK. The next portion of Listing 4.2 is the definition of the Python function `stemSentence()`, which determines the stem of each word in the sentence that is passed into the function.

The next portion of code defines the Python function `saveStemmedLines()`, which contains a `for` loop that iterates through the sentences that have been processed by the function `stemSentence()`. The last portion of Listing 4.2 invokes the Python function `stemSentence()` with the first sentence in `my_lines_list`, displays the output, and then saves the stemmed text. Launch the code in Listing 4.2 to see the following output:

```
'data scienc is an interdisciplinari field that use
scientif method, process, algorithm and system to extract
knowledg and insight from data in variou form, both
structur and unstructur, [1] [2] similar to data mine.
\n', 'data scienc is a '' concept to unifi statist, data
analysi, machin learn and their relat method '' in order
to '' understand and analyz actual phenomena '' with
data. [3] It employ techniqu and theori drawn from mani
field within the context of mathemat, statist, inform
scienc, and comput scienc. \n',
```

Listing 4.3 shows the contents of `porter_lancaster1.py` that illustrate how to invoke a `Porter` stemmer and a Lancaster stemmer to perform stemming on a set of words.

LISTING 4.3: porter_lancaster1.py

```
from nltk.stem import PorterStemmer
from nltk.stem import LancasterStemmer

# create two stemmers:
porter = PorterStemmer()
lancaster = LancasterStemmer()

#provide a word to be stemmed
print("Porter Stemmer:")
print(porter.stem("cats"))
print(porter.stem("trouble"))
print(porter.stem("troubling"))
print(porter.stem("troubled"))

print("Lancaster Stemmer:")
print(lancaster.stem("cats"))
print(lancaster.stem("trouble"))
print(lancaster.stem("troubling"))
print(lancaster.stem("troubled"))
```

In Listing 4.3, the first portion displays the stemmed values that are determined by an instance of the `PorterStemmer` class, followed by another code block that performs the same calculations based on a `LancasterStemmer`. Launch the code in Listing 4.3 to see the following output:

```
Porter Stemmer:
cat
troubl
troubl
troubl
Lancaster Stemmer:
cat
troubl
troubl
troubl
```

Listing 4.4 shows the contents of `porter_lancaster2.py` that illustrate how to invoke a `PorterStemmer` and a `LancasterStemmer` to perform stemming on a set of words.

LISTING 4.4: porter_lancaster2.py

```
from nltk.stem import PorterStemmer
from nltk.stem import LancasterStemmer

# create two stemmers:
porter = PorterStemmer()
lancaster = LancasterStemmer()

word_list = ["friend", "friendship", "friends", "friendships",
"stabil","destabilize","misunderstanding","railroad",
                                    "moonlight","football"]

print("{0:20}{1:20}{2:20}".format("Word","Porter
                              Stemmer","lancaster Stemmer"))

for word in word_list: print ("{0:20}{1:20}{2:20}".
format(word,porter.stem(word),lancaster.stem(word)))
```

Listing 4.4 initializes the variables `porter` and `lancaster` as instances of the classes `PorterStemmer` and `LancasterStemmer`, respectively, and then initializes the array `word_list` with a set of strings. The next portion of Listing 4.4 iterates through the words in the variable `word_list` and displays the stemmed values produced via the `porter` stemmer and the `lancaster` stemmer. Launch the code in Listing 4.4 to see the following output:

```
Word              Porter Stemmer     lancaster Stemmer
friend            friend             friend
friendship        friendship         friend
friends           friend             friend
friendships       friendship         friend
stabil            stabil             stabl
destabilize       destabil           dest
misunderstanding  misunderstand      misunderstand
railroad          railroad           railroad
moonlight         moonlight          moonlight
football          footbal            footbal
```

NLTK AND LEMMATIZATION

Listing 4.5 shows the contents of `lemmatizer1.py` that illustrate how to perform lemmatization on a sentence.

LISTING 4.5: lemmatizer1.py

```
import nltk
from nltk.stem import WordNetLemmatizer

wordnet_lemmatizer = WordNetLemmatizer()

sentence = "He eats Chicago deep dish pizzas, and lots of
                                  pizzas from Pizzeria Uno!"
punctuations = "?:!.,;"
```

```
sentence_words = nltk.word_tokenize(sentence)

for word in sentence_words:
  if word in punctuations:
    sentence_words.remove(word)

print("{0:20}{1:20}".format("Word","Lemma"))
for word in sentence_words:
  print
("{0:20}{1:20}".format(word,wordnet_lemmatizer.
                                          lemmatize(word)))
```

Listing 4.5 initializes the variable `wordnet_lemmatizer` as an instance of the `WordNetLemmatizer` class that is available in NLTK. The variables `sentence` and `punctuations` are initialized, as well as the variable `sentence_words` that consists of the tokens in the variable `sentence`. Next is a loop that removes any punctuation symbols from the variable `sentence_words`. The final code block contains a loop that displays each word in `sentence_words`, along with its lemmatized value. Launch the code in Listing 4.5 to see the following output:

```
Word                Lemma
He                  He
eats                eats
Chicago             Chicago
deep                deep
dish                dish
pizzas              pizza
and                 and
lots                lot
of                  of
pizzas              pizza
from                from
Pizzeria            Pizzeria
Uno                 Uno
```

Listing 4.6 shows the contents of `lemmatizer2.py` that illustrate how to perform lemmatization on a sentence.

LISTING 4.6: lemmatizer2.py

```
import nltk
from nltk.stem import WordNetLemmatizer

wordnet_lemmatizer = WordNetLemmatizer()

sentence = "He eats Chicago deep dish pizzas, and lots of
                                  pizzas from Pizzeria Uno!"
punctuations = "?:!.,;"

sentence_words = nltk.word_tokenize(sentence)

for word in sentence_words:
  if word in punctuations:
   sentence_words.remove(word)

# display "part-of-speech" via pos = "v"
print("{0:20}{1:20}".format("Word","Lemma"))
for word in sentence_words:
  print ("{0:20}{1:20}".format(word,wordnet_lemmatizer.
lemmatize(word, pos = "v")))
```

Listing 4.6 is similar to Listing 4.5, with the addition of a `for` loop (shown in bold) that displays the words in `sentence_words`, along with their lemmatized values. Launch the code in Listing 4.6 to see the following output:

```
Word                Lemma
He                  He
eats                eat
Chicago             Chicago
deep                deep
dish                dish
pizzas              pizzas
and                 and
lots                lot
of                  of
pizzas              pizzas
from                from
Pizzeria            Pizzeria
Uno                 Uno
```

NLTK AND STOP WORDS

Listing 4.7 shows the contents of `nltk_newstop.py` that illustrate how to add new stop words to the default set of stop words.

LISTING 4.7: nltk_newstop.py

```
from sklearn.feature_extraction.text import
                                                CountVectorizer
from nltk.corpus import stopwords

newstop = set(stopwords.words('english')+['pasta','bliffet'])

print("new stopwords:")
print(newstop)
```

Listing 4.7 initializes the variable `newstop` with a pair of strings, after which the variable `cv` is initialized as an instance of the `CountVectorizer` class from Sklearn. Note that the latter initialization specifies the variable `newstop` as the set of stop words. Launch the code in Listing 4.7 to see the following output:

```
new stopwords:
{'couldn', 'hadn', 'theirs', 'shan', 'weren',
'having', 'y', 'between', 'before', 'can', 'other',
'all', 'wouldn', 'once', 'of', 'me', 'but', 'doing',
'because', 'own', 'mightn', 'hers', 'after', 'out',
'd', 'where', 'the', 'her', 'nor', 'him', 'below',
'both', 'do', "you'll", 'won', 'ourselves', "needn't",
'ma', 'now', 'from', 'she', 'down', "you'd", 'an',
"hadn't", 'should', 'were', 'you', 'it', 'such',
'their', 'bliffet', 'isn', 'who', 'had', "didn't",
'mustn', 'our', "you've", 'or', 'again', "aren't",
"won't", 'itself', 'any', 'those', "don't", "isn't",
'am', "she's", 'how', 'than', 'be', 'as', 'has',
'being', 'each', 'doesn', 'wasn', 'ours', 'while',
"hasn't", 'about', 'herself', 'with', 'on', 'them',
'shouldn', 'a', 'if', 'off', 'will', "wouldn't",
'over', 'some', 'these', 'there', 'why', 'yourself',
'too', "doesn't", 'just', 'didn', "shouldn't", 'don',
'which', 'few', "mightn't", "you're", "haven't",
'my', 'i', 'and', 'then', 'only', 'by', 'your',
```

```
'what', 'when', 'up', 'here', 'o', 't', 'during',
'are', "couldn't", 'through', 'themselves', 'himself',
'until', 'did', 'against', 's', 'was', 'so', 'this',
'have', 'to', 'll', 'm', 'myself', 'at', 've', 'for',
'we', 'does', 'its', 're', 'needn', "mustn't", 'more',
'pasta', 'he', 'no', "shan't", 'his', 'above', 'that',
'under', "weren't", 'whom', 'further', 'ain', 'is',
'into', "should've", 'been', 'haven', 'yourselves',
'very', "wasn't", 'hasn', 'same', 'most', 'not',
'they', "that'll", "it's", 'yours', 'in', 'aren'}
```

WHAT IS WORDNET?

Wordnet is a corpus reader that is provided by NLTK, and you can use either of the following snippets to import Wordnet:

```
from nltk.corpus import wordnet
from nltk.corpus import wordnet as wn
```

Wordnet provides the `synsets()` function that enables you to search for words and display their POS, as well as synonyms and antonyms of a given word.

Wordnet (via NLTK) provides the following similarity scorers (authors are in parentheses):

- jcn_similarity
- lch_similarity (Leacock-Chodorow)
- lin_similarity
- path_similarity
- res_similarity
- wup_similarity (Wu-Palmer)
- PPMI

Listing 4.8 shows the contents of `similarity1.py` that illustrate how to use `wup_similarity()` to find the similarity between a pair of words.

LISTING 4.8: similarity1.py

```
from nltk.corpus import wordnet as wn

# Wu and Palmer method to compare word similarity
w1 = wn.synset('ship.n.01')
```

```
w2 = wn.synset('boat.n.01')
print("=> similarity of ship and boat:")
print(w1.wup_similarity(w2))
print()

w1 = wn.synset('ship.n.01')
w2 = wn.synset('car.n.01')
print("=> similarity of ship and car:")
print(w1.wup_similarity(w2))
print()

w1 = wn.synset('ship.n.01')
w2 = wn.synset('dog.n.01')
print("=> similarity of ship and dog:")
print(w1.wup_similarity(w2))
print()
```

Listing 4.8 contains three blocks of code that compare the word "ship" with "boat," "car," and "dog," using wordnet. Launch the code in Listing 4.8 to see the following output:

```
=> similarity of ship and boat:
0.9090909090909091

=> similarity of ship and car:
0.6956521739130435

=> similarity of ship and dog:
0.4
```

Listing 4.9 shows the contents of `wordnet1.py` that illustrate how to use `path_similarity()` to find the similarity between a pair of words.

LISTING 4.9: wordnet1.py

```
from nltk.corpus import wordnet as wn

# Multilingual WordNet (ISO-639 language codes):
print("=> sorted(wn.langs()):")
print(sorted(wn.langs()))
print()
```

```
drink = wn.lemma('drink.v.03.drink')
print("=> drink:",drink)
print("=> count:",drink.count())
print()

horse = wn.synset('horse.n.01')
giraffe = wn.synset('giraffe.n.01')
zebra = wn.synset('zebra.n.01')

print("=> horse.path_similarity(giraffe):")
print(horse.path_similarity(giraffe))
print()

print("=> horse.path_similarity(zebra):")
print(horse.path_similarity(zebra))
```

Listing 4.9 starts by displaying a sorted list of language code for supported languages, followed by the lemmatized value and frequency of the word "drink." The next code snippet compares the word "horse" with the word "giraffe" and then with the word "zebra," similar to the code in Listing 4.8. Launch the code in Listing 4.9 to see the following output:

```
=> sorted(wn.langs()):
['als', 'arb', 'bul', 'cat', 'cmn', 'dan', 'ell',
'eng', 'eus', 'fas', 'fin', 'fra', 'glg', 'heb', 'hrv',
'ind', 'ita', 'jpn', 'nld', 'nno', 'nob', 'pol', 'por',
'qcn', 'slv', 'spa', 'swe', 'tha', 'zsm']

=> drink: Lemma('toast.v.02.drink')
=> count: 1

=> horse.path_similarity(giraffe):
0.14285714285714285

=> horse.path_similarity(zebra):
0.3333333333333333
```

Synonyms and Antonyms

Listing 4.10 shows the contents of `nltk_syn_ant.py` that illustrate how to find synonyms and antonyms of a word using NLTK.

LISTING 4.10: nltk_syn_ant.py

```
from nltk.corpus import wordnet

word = "work"

synonyms = []
for syn in wordnet.synsets(word):
  for lemma in syn.lemmas():
    synonyms.append(lemma.name())

print("=> synonyms for",word,":")
print(synonyms)
print("-------------------------")

antonyms = []
for syn in wordnet.synsets(word):
  for lemma in syn.lemmas():
    if lemma.antonyms():
      antonyms.append(lemma.antonyms()[0].name())

print("=> antonyms for",word,":")
print(antonyms)
```

Listing 4.10 starts with a loop that finds and displays the words that are synonyms for the word "work." The next block of code in Listing 4.10 contains a loop that finds and displays the antonyms for the word "work." Launch the code in Listing 4.10 to see the following output:

```
=> synonyms for work :
['work', 'work', 'piece_of_work', 'employment', 'work',
'study', 'work', 'work', 'workplace', 'work', 'oeuvre',
'work', 'body_of_work', 'work', 'work', 'do_work',
'work', 'act', 'function', 'work', 'operate', 'go',
'run', 'work', 'work_on', 'process', 'exercise',
'work', 'work_out', 'make', 'work', 'work', 'work',
'work', 'bring', 'work', 'play', 'wreak', 'make_for',
'work', 'put_to_work', 'cultivate', 'crop', 'work',
'work', 'influence', 'act_upon', 'work', 'work',
```

```
'work', 'work', 'work', 'shape', 'form', 'work',
'mold', 'mould', 'forge', 'work', 'knead', 'work',
'exploit', 'work', 'solve', 'work_out', 'figure_out',
'puzzle_out', 'lick', 'work', 'ferment', 'work',
'sour', 'turn', 'ferment', 'work', 'work']
-------------------------
=> antonyms for work :
['idle', 'malfunction']
```

NLTK, lxml, AND XPath

This section contains a code sample that combines NLTK, lxml, and XPath expressions. As a reminder, you can find online tutorials that discuss basic concepts of XPath if you are unfamiliar with XPath expressions.

NOTE *Make sure you invoke* `pip3 install lxml`.

Listing 4.11 shows the contents of `nltk_xpath.py` that illustrate how to retrieve the contents of the HTML Webpage *https://www.github.com*.

LISTING 4.11: nltk_xpath.py

```
import nltk
import lxml
from lxml import html
import requests

page = requests.get('https://www.ibm.com/events/think')
root = lxml.html.fromstring(page.content)
tree = html.fromstring(page.content)
data = tree.xpath('//*[@id = "conference-overview"]/div/
                                                div[2]/div/p')

print("root:")
print(root)
print("--------------------------\n")

print("data:")
print(data)
print("--------------------------\n")
```

```
items = []
for item in data:
  content = item.text_content()
  items.append(content)
  print("=> content:",item.text_content())
print("---------------------------\n")

print("items1:")
print(items)
print("---------------------------\n")
```

Listing 4.11 contains several `import` statements, followed by the initialization of several variables. The `page` variable is initialized with the contents of the URL in the `requests.get()` method. The variable `root` is assigned the top-level node, and the `tree` variable is assigned the contents of the HTML Webpage.

The fourth variable, `data`, is initialized with the result of an `XPath` expression, as shown here:

```
data = tree.xpath('//*[@id="conference-overview"]/div/
                                                div[2]/div/p')
```

The left-most portion of the variable `data` consists of a tree of elements (starting from the `root` element) whose `id` value equals the string `conference-overview`. This tree is further pruned by selecting the leaf nodes of the preceding tree that have a descendant that matches the partial path `div/div[2]/div`. The preceding partial path involves (a) selecting elements that have a `<div>` element, then (b) navigating to the *second* `<div>` child element, and (c) further navigating to the `<div>` child elements of the elements in (b). The final step involves selecting the `<p>` elements that are child elements of the previous step.

The next section in Listing 4.11 displays the `root` element and the `data` element, followed by a `for` loop that iterates through the data subtree to append the text content of each element in the `data` variable. Launch the code in Listing 4.11 to see 102 lines of output. Only the first portion of the output is shown.

```
root:
<Element html at 0x1160ee778>
---------------------------
```

```
data:
[<Element p at 0x11613b138>, <Element p at 0x11613b048>,
<Element p at 0x11613b188>, <Element p at 0x11613b1d8>,
<Element p at 0x11613b228>]
---------------------------

=> content: As the evolving impacts of COVID-19 ripple
through our communities, we are all facing unforeseen
challenges.
=> content: Gain new skills needed to adapt and evolve.
Explore new ways of working and learn how to stabilize
and protect your organization. Enhance IT resiliency,
ensure business continuity and most importantly, stay
connected.
=> content: Sessions are on demand. Self-paced labs are
available all day through Sunday, May 10.
=> content: Let's get thinking.
=> content: Watch now View self-paced labs
---------------------------

items1:
['As the evolving impacts of COVID-19 ripple through
our communities, we are all facing unforeseen
challenges.', 'Gain new skills needed to adapt and
evolve. Explore new ways of working and learn how
to stabilize and protect your organization. Enhance
IT resiliency, ensure business continuity and most
importantly, stay connected.', 'Sessions are on demand.
Self-paced labs are available all day through Sunday,
May 10.', "Let's get thinking.", 'Watch now View
self-paced labs']
-------------------------
```

NLTK AND N-GRAMS

This section contains a code sample that uses NLTK to generate n-grams from a document. Listing 4.12 shows the contents of `nltk_ngrams.py` that illustrate how to retrieve the contents of the HTML webpage *https://www.github.com*.

LISTING 4.12: nltk_ngrams.py

```
import re
from nltk.util import ngrams

str = "Natural-language processing (NLP) is an area of
computer science and artificial intelligence concerned with
the interactions between computers and human (natural)
languages."

str = str.lower()
str = re.sub(r'[^a-zA-Z0-9\s]', ' ', str)
tokens = [token for token in str.split(" ") if token != ""]
grams5 = list(ngrams(tokens, 5))

print("Generated 5-grams:")
print(grams5)
```

Listing 4.12 initializes the variable `str` with a text string, converts the text to lowercase, and then replaces every nonalphanumeric character with a single white space via a regular expression.

Next the `tokens` variable is initialized with the nonempty tokens in the `str` variable, followed by the variable `grams5` that is a list of 5 grams that are constructed from the `tokens` variable. Launch the code in Listing 4.12 to see 102 lines of output. Only the first portion of the output is given here.

```
Generated 5-grams:
[('natural', 'language', 'processing', 'nlp',
'is'), ('language', 'processing', 'nlp', 'is',
'an'), ('processing', 'nlp', 'is', 'an', 'area'),
('nlp', 'is', 'an', 'area', 'of'), ('is', 'an',
'area', 'of', 'computer'), ('an', 'area', 'of',
'computer', 'science'), ('area', 'of', 'computer',
'science', 'and'), ('of', 'computer', 'science',
'and', 'artificial'), ('computer', 'science', 'and',
'artificial', 'intelligence'), ('science', 'and',
'artificial', 'intelligence', 'concerned'), ('and',
'artificial', 'intelligence', 'concerned', 'with'),
('artificial', 'intelligence', 'concerned', 'with',
'the'), ('intelligence', 'concerned', 'with',
'the', 'interactions'), ('concerned', 'with',
'the', 'interactions', 'between'), ('with', 'the',
'interactions', 'between', 'computers'), ('the',
```

```
'interactions', 'between', 'computers', 'and'),
('interactions', 'between', 'computers', 'and',
'human'), ('between', 'computers', 'and', 'human',
'natural'), ('computers', 'and', 'human', 'natural',
'languages')]
```

NLTK AND POS (I)

This section contains a code sample that uses `NLTK` to display the parts of speech for the words in a sentence.

Listing 4.13 shows the contents of `nltk_pos.py` that illustrate how to tokenize a sentence and then determine the parts of speech for each word in that sentence.

LISTING 4.13: nltk_pos.py

```
import nltk
from nltk.tokenize import word_tokenize
from nltk.corpus import wordnet

sentence = "I love pizza, and also pasta, what about you?"
split_words = sentence.split(" ")
print("sentence:")
print(sentence)
print()
print("split_words:")
print(split_words)
print()

word_tokenize(sentence)

w = word_tokenize(sentence)
pos = nltk.pos_tag(w)

print("tokenized:")
print(w)
print()

print("parts of speech:")
print(pos)
print()
```

```
syn = wordnet.synsets("spaceship")
print(syn)
print(syn[0].name())
print(syn[0].definition())

syn = wordnet.synsets("sleep")
print("examples of sleep:")
print(syn[0].examples())
print()
```

Listing 4.13 initializes the variable `sentence` to a text string, and initializes the variable `split_words` with the tokens in the variable `sentence`. The next block of `print()` statements displays the contents of `sentence` and `split_words`.

The next code snippet initializes the variable `w` with the result of passing `sentence` to the `word_tokenize` method from NLTK. The variable `pos` is then initialized with the parts of speech of the elements in `w`, followed by a block of `print()` statements that display their values as well as their parts of speech.

The final portion of Listing 4.13 invokes the `wordnet.synsets()` method to find the definitions of the word `spaceship` and the word `sleep`. In both cases, the definitions are displayed. Launch the code in Listing 4.14 to see the following output:

```
sentence:
I love pizza, and also pasta, what about you?


split_words:
['I', 'love', 'pizza,', 'and', 'also', 'pasta,',
'what', 'about', 'you?']


tokenized:
['I', 'love', 'pizza', ',', 'and', 'also', 'pasta',
',', 'what', 'about', 'you', '?']


parts of speech:
[('I', 'PRP'), ('love', 'VBP'), ('pizza', 'NN'), (',',
','), ('and', 'CC'), ('also', 'RB'), ('pasta', 'NN'),
(',', ','), ('what', 'WP'), ('about', 'IN'), ('you',
'PRP'), ('?', '.')]
```

```
[Synset('starship.n.01')]
starship.n.01
a spacecraft designed to carry a crew into interstellar
space (especially in science fiction)

examples of sleep:
["he didn't get enough sleep last night", 'calm as a
child in dreamless slumber']
```

Listing 4.14 shows the contents of `nltk_movie_reviews.py` that illustrate how to tokenize a sentence and display relevant movie-related words.

LISTING 4.14: nltk_movie_reviews.py

```
import nltk
from nltk.corpus import movie_reviews
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize

print("first 16 English stop words:")
print(stopwords.words('english')[:16])
print()

para = "I started with Deep Learning, then proceeded
to Machine Learning, then NLP, and finally reached
Deep Reinforcement Learning. However, despite the
preparatory classes, the challenge of DRL was very
steep."

words = word_tokenize(para)
print("tokenized words:")
print(words)
print()

useful_words = [word for word in words if word not in
                              stopwords.words('english')]
print("useful words:")
print(useful_words)
print()
```

```
print("movie reviews words:")
print(movie_reviews.words())
print()

print("movie reviews categories:")
print(movie_reviews.categories())
print()

print("movie reviews fileids:")
print(movie_reviews.fileids()[:4])
print()

all_words = movie_reviews.words()
freq_dist = nltk.FreqDist(all_words)
print("frequency distribution for 20 most common words:")
print(freq_dist.most_common(20))
print()
```

Listing 4.14 starts with several `import` statements, prints some stop words, and then initializes the variable `para` as a text string, which is tokenized and its tokens are then displayed. Next, the variable `useful_words` is initialized with the result of removing the stop words from the variable `words`.

The next code block in Listing 4.14 prints the nonstop words, followed by the movie-related words (from the `movie_reviews` class). The last portion of Listing 4.14 displays several file IDs and then the distribution for the 20 most common words. Launch the code in Listing 4.14 to see the following output:

```
first 16 English stop words:
['i', 'me', 'my', 'myself', 'we', 'our', 'ours',
'ourselves', 'you', "you're", "you've", "you'll",
"you'd", 'your', 'yours', 'yourself']


tokenized words:
['My', 'goal', 'was', 'simple', ':', 'learn',
'as', 'much', 'as', 'possible', 'about', 'Deep',
'Reinforcement', 'Learning', '.', 'Even', 'after',
```

```
'studying', 'machine', 'learning', ',', 'deep',
'learning', ',', 'and', 'natural', 'language',
'processing', ',', 'the', 'challenge', 'of', 'DRL',
'was', 'very', 'steep', '.']

useful words:
['My', 'goal', 'simple', ':', 'learn', 'much',
'possible', 'Deep', 'Reinforcement', 'Learning', '.',
'Even', 'studying', 'machine', 'learning', ',', 'deep',
'learning', ',', 'natural', 'language', 'processing',
',', 'challenge', 'DRL', 'steep', '.']

movie reviews words:
['plot', ':', 'two', 'teen', 'couples', 'go', 'to', ...]

movie reviews categories:
['neg', 'pos']

movie reviews fileids:
['neg/cv000_29416.txt', 'neg/cv001_19502.txt', 'neg/
cv002_17424.txt', 'neg/cv003_12683.txt']

frequency distribution for 20 most common words:
[(',', 77717), ('the', 76529), ('.', 65876), ('a',
38106), ('and', 35576), ('of', 34123), ('to', 31937),
("'", 30585), ('is', 25195), ('in', 21822), ('s',
18513), ('"', 17612), ('it', 16107), ('that', 15924),
('-', 15595), (')', 11781), ('(', 11664), ('as',
11378), ('with', 10792), ('for', 9961)]
```

NLTK AND POS (2)

This section contains a code sample that uses NLTK to display the parts of speech for the words in a sentence. Listing 4.15 shows the contents of `nltk_entities.py` that illustrate how to tokenize a sentence and then determine the parts of speech for each word in that sentence.

LISTING 4.15: nltk_entities.py

```
import nltk
from nltk.tokenize import sent_tokenize, word_tokenize
from nltk import ne_chunk, pos_tag

text = 'the SF weather is hot and the LA weather is hotter'
words = word_tokenize(text)
print("words:")
print(words)
print()

print("nltk.pos_tag(words):")
print(nltk.pos_tag(words))

def entities(text):
  return ne_chunk(pos_tag(word_tokenize(text)))

tree = entities(text)
print("tokenized:")
print(tree)
print("----------------")
tree.draw()
```

Listing 4.15 initializes the variable `text` with a text string, followed by the variable `words` that consists of the tokens of the variable `text`. Next, several `print()` statements display the contents of the variable `words` as well as the parts of speech of the tokens in the variable `words`.

The next portion of Listing 4.15 is a Python function `entities()` that returns the parts of speech of the tokens in the text string that is passed in to the `entities()` function. The final portion of Listing 4.15 invokes the `entities()` function, assigns the result to the variable `tree`, and then displays the contents of `tree`. Launch the code in Listing 4.15 to see the following output:

```
words:
['the', 'SF', 'weather', 'is', 'hot', 'and', 'the',
'LA', 'weather', 'is', 'hotter']


nltk.pos_tag(words):
[('the', 'DT'), ('SF', 'NNP'), ('weather', 'NN'),
('is', 'VBZ'), ('hot', 'JJ'), ('and', 'CC'), ('the',
```

```
'DT'), ('LA', 'NNP'), ('weather', 'NN'), ('is', 'VBZ'),
('hotter', 'RBR')]
tokenized:
(S
  the/DT
  (ORGANIZATION SF/NNP)
  weather/NN
  is/VBZ
  hot/JJ
  and/CC
  the/DT
  (ORGANIZATION LA/NNP)
  weather/NN
  is/VBZ
  hotter/RBR)
----------------
```

FIGURE 4.1. The tree structure with named entities that is generated when the code in Listing 4.16 is launched.

NLTK AND TOKENIZERS

This section contains a code sample that contains various tokenizers that are defined in NLTK. Listing 4.16 shows the contents of `nltk_tokenizers.py` that illustrate various ways of tokenizing the words in a sentence.

LISTING 4.16: nltk_tokenizers.py

```
import nltk

text = "I love deep dish pizza. Mainly from Chicago. Also
                                                  with beer."
```

```
sents = nltk.sent_tokenize(text)
words = nltk.word_tokenize(text)
tokens = nltk.wordpunct_tokenize(text)
tags = nltk.pos_tag(words)

print("text: ",text)
print("sents: ",sents)
print("words: ",words)
print("tokens:",tokens)
print("tags: ",tags)
```

Listing 4.16 initializes the variable `text` with a text string, followed by variables that are instances of various tokenizers that are available in NLTK. The block of `print()` statements displays the tokens that are produced by each of the tokenizers. Launch the code in Listing 4.16 to see the following output:

```
text: I love deep dish pizza. Mainly from Chicago. Also
with beer.
sents: ['I love deep dish pizza.', 'Mainly from
Chicago.', 'Also with beer.']
words: ['I', 'love', 'deep', 'dish', 'pizza', '.',
'Mainly', 'from', 'Chicago', '.', 'Also', 'with',
'beer', '.']
tokens: ['I', 'love', 'deep', 'dish', 'pizza', '.',
'Mainly', 'from', 'Chicago', '.', 'Also', 'with',
'beer', '.']
tags: [('I', 'PRP'), ('love', 'VBP'), ('deep', 'JJ'),
('dish', 'JJ'), ('pizza', 'NN'), ('.', '.'), ('Mainly',
'RB'), ('from', 'IN'), ('Chicago', 'NNP'), ('.', '.'),
('Also', 'RB'), ('with', 'IN'), ('beer', 'NN'), ('.', '.')]
```

The following URL contains a list of the terms in the preceding output, along with their corresponding parts of speech:

https://cs.nyu.edu/grishman/jet/guide/PennPOS.html

The next section provides another example of finding named entities using the NLTK library.

NLTK AND CONTEXT-FREE GRAMMARS (OPTIONAL)

In simplified terms, a context-free grammar (CFG) is a set of rules or "productions" that are used to generate patterns of strings. Such rules are typically recursive, and they involve a set of terminal symbols, a set of nonterminal symbols, and an end symbol. There is also a start symbol that is a nonterminal symbol and appears in the initial string that is generated by a CFG. Depending on the complexity of the CFG, you can also see regular expressions in the productions.

Many popular programming languages, such as C, C++, and Java (but not Fortran), are context-free grammars. For example, the complete set of production rules for the C programming language is about five pages long.

As a simple example, here is a grammar that represents arithmetic expressions that contain any combination of the arithmetic operators *, /, +, and – (and numeric values as operands):

```
<expression> --> number
<expression> --> ( <expression> )
<expression> --> <expression> + <expression>
<expression> --> <expression> - <expression>
<expression> --> <expression> * <expression>
<expression> --> <expression> / <expression>
```

You might be surprised to discover that the NLTK library provides support for CFGs. Listing 4.17 shows the contents of `nltk_grammar.py` that illustrate how to specify a set of production rules for a simple context-free grammar. Note that it's more common to see the single letters N, V, and P used instead of noun, verb, and preposition in CFGs. The latter are just to show you that you can be more expressive in the productions of a CFG.

LISTING 4.17: nltk_grammar.py

```
import re
import nltk
from nltk.parse import RecursiveDescentParser

# Define a CFG (Context Free Grammar):
mygrammar = nltk.CFG.fromstring("""
S -> NP VP
NP -> Art Noun | Art Noun PP
VP -> Verb | Verb NP | Verb NP PP
PP -> Prep NP
```

```
Art -> 'a' | 'an' | 'the'
Noun -> 'boy' | 'ball' | 'apple' | 'hot' | 'dog'
Verb -> 'ate' | 'ran' | 'studied'
Prep -> 'in' | 'with'
""")

# a top-down parser:
rdstr = RecursiveDescentParser(mygrammar)

str1 = "the boy with the ball ate a hot dog"
str2 = "the boy ate an apple with the ball"

#display the grammar trees:
print("grammar trees for str1:")
for tree in rdstr.parse(str1.split()):
 print("tree:",tree)
print("------------------------")

print("grammar trees for str2:")
for tree in rdstr.parse(str2.split()):
 print("tree:",tree)
```

Listing 4.17 initializes the variable `mygrammar` with 8 production rules that are used to determine whether a text string will parse correctly according to the given grammar. The next code snippet initializes the variable `rdstr` as an instance of the class `RecursiveDescentParser` (from NLTK) with the argument `mygrammar`.

The two variables `str1` and `str2` are initialized with text strings. One of them parses according to the grammar and one does not parse correctly. The two strings are very similar, and it's challenging to parse them visually against the grammar. However, the code in Listing 4.17 also contains two loops for displaying the parse tree for `str1` and `str2`. Notice that there is only a parse tree for `str2` because `str1` does not belong to the defined grammar. Launch the code in Listing 4.17 to see trees for the second sentence `str2` but not for `str1` (can you see why?).

```
grammar trees for str1:
------------------------
grammar trees for str2:
tree: (S
  (NP (Art the) (Noun boy))
  (VP
```

```
      (Verb ate)
      (NP
        (Art an)
        (Noun apple)
        (PP (Prep with) (NP (Art the) (Noun ball))))))
tree: (S
  (NP (Art the) (Noun boy))
  (VP
    (Verb ate)
    (NP (Art an) (Noun apple))
    (PP (Prep with) (NP (Art the) (Noun ball)))))
```

WHAT IS GENSIM?

Gensim is an open source Python-based NLP library for performing NLP-related tasks, such as text processing, word embeddings, and topic modeling. Its homepage is at *https://gensim.readthedocs.io/en/latest/index.html.*

Gensim supports unsupervised topic modeling and uses modern statistical machine learning. Gensim is also implemented in Cython (in addition to Python). Gensim works with word vector models, such as Word2Vec and FastText, and also supports LDA and LSI for topic modeling. Moreover, Gensim is compatible with scipy and NumPy, and provides functions to convert from/to NumPy arrays.

Gensim works with a single text-based document or a corpus (collection) of text-based documents. Gensim uses vectors to represent the words in a document, and its model is an algorithm for transforming vectors from one representation to another.

In addition, Gensim can create a BoW corpus, a tf-idf model, a word2vec model, and n-grams. Moreover, Gensim computes similarity metrics. Now let's look at an example of gensim with tf-idf that is discussed in the next section.

Gensim and tf-idf Example

Gensim works with a single text-based document or a corpus (collection) of text-based documents. Listing 4.18 shows the contents of `gensim_tfidf.py` that illustrate how to combine `gensim` with `tf-idf`.

LISTING 4.18: gensim_tfidf.py

```
from gensim import models
from gensim import corpora
from gensim.utils import simple_preprocess
import numpy as np

documents = ["This is the first line",
             "This is the second sentence",
             "This third document"]

# Create the Dictionary and Corpus
mydict = corpora.Dictionary([simple_preprocess(line) for
                                          line in documents])
corpus = [mydict.doc2bow(simple_preprocess(line)) for line
                                                in documents]

# Show the Word Weights in Corpus
for doc in corpus:
    print([[mydict[id], freq] for id, freq in doc])

# Create the TF-IDF model
tfidf = models.TfidfModel(corpus, smartirs = 'ntc')

# Show the TF-IDF weights
for doc in tfidf[corpus]:
  print([[mydict[id], np.around(freq, decimals = 2)] for id,
                                                freq in doc])
```

Listing 4.18 starts with several `import` statements and then initializes the variable `documents` as an array of sentences. Next, the variable `mydict` is initialized as a dictionary that is based on the sentences in the `documents` variable. Notice that these sentences are first processed via the `simple_preprocess` class that tokenizes each sentence and converts the tokens to lowercase (and tokens are UTF-8 format).

The next code snippet instantiates the variable `tfidf` as an instance of the `TfidfModel` class. The final portion of Listing 4.18 displays the contents of `tfidf`.

Launch the code in Listing 4.19 to see the following output:

```
# [['first', 1], ['is', 1], ['line', 1], ['the', 1],
['this', 1]]
```

```
# [['is', 1], ['the', 1], ['this', 1], ['second', 1],
['sentence', 1]]
# [['this', 1], ['document', 1], ['third', 1]]


# [['first', 0.66], ['is', 0.24], ['line', 0.66],
['the', 0.24]]
# [['is', 0.24], ['the', 0.24], ['second', 0.66],
['sentence', 0.66]]
# [['document', 0.71], ['third', 0.71]]
```

Saving a Word2vec Model in Genism

Listing 4.19 shows the contents of `gensim_word2vec.py` that illustrate how to save a word2vec model in Gensim.

LISTING 4.19: gensim_word2vec.py

```
from gensim.models import word2vec

corpus = [
          'Text of the first document.',
          'Text of the second document made longer.',
          'Number three.',
          'This is number four.',
]

# split sentences:
tokenized_sentences = [sentence.split() for sentence in
                                                    corpus]

model = word2vec.Word2Vec(tokenized_sentences, min_count = 1)
                                        print("model:",model)

model.save("word2vec.model")
```

Listing 4.19 starts with an `import` statement and then initializes the variable `corpus` as an array of sentences. Next, the variable `tokenized_sentences` is initialized with the result of splitting `corpus` into sentences. The next code snippet instantiates the variable `model` as an instance of the `Word2Vec` class, along with the contents of the variable `tokenized_sentences`.

The final portion of Listing 4.20 displays the contents of `model` and then saves the model in the file `word2vec.model`. Launch the code in Listing 4.19 to see the following output:

```
model: Word2Vec(vocab=15, size=100, alpha=0.025)
```

AN EXAMPLE OF TOPIC MODELING

Chapter 3 contains a high-level description of LDA (Latent Dirichlet Allocation), and this section contains a code sample for LDA. Listing 4.20 shows the contents of `lda_topic_modeling.py` that illustrate how to use LDA to perform topic modeling. The "documents" in this code sample are very short (and admittedly contrived), but you can replace them with your own set of documents and launch the code.

LISTING 4.20: lda_topic_modeling.py

```
# define a set of short documents:
doc1 = "Our plan was not without merit is similar in
meaning to Our plan has merit."
doc2 = "I like the pizza toppings but I do not like the
crust."
doc3 = "The only thing worse than being talked about, is
not being talked about according to Oscar Wilde"
doc4 = "Everything is funny, as long as it's happening to
somebody else according to Will Rogers"
doc5 = "When ignorance is bliss, 'tis folly to be wise by
William Shakespeare and my favorite misquoted quote"
doc6 = "Good judgement is the result of experience and
experience the result of bad judgement as Mark Twain
astutely observed."

all_docs = [doc1, doc2, doc3, doc4, doc5, doc6]

from nltk.corpus import stopwords
from nltk.stem.wordnet import WordNetLemmatizer
import string

stop = set(stopwords.words('english'))
exclude = set(string.punctuation)
lemma = WordNetLemmatizer()
```

```
def clean_documents(doc):
  no_stop = ' '.join([i for i in doc.lower().split() if i
                                                  not in stop])
  no_punct = ' '.join([ch for ch in no_stop if ch not in
                                                      exclude])
  normalized = ' '.join(lemma.lemmatize(word) for word in
                                              no_punct.split())
  return normalized

cleaned_docs = [clean_documents(doc).split() for doc in
                                                      all_docs]

import gensim
from gensim import corpora
dictionary = corpora.Dictionary(cleaned_docs)
doc_term_matrix = [dictionary.doc2bow(doc) for doc in
                                                  cleaned_docs]

# import and get an instance of the LdaModel:
Lda = gensim.models.ldamodel.LdaModel
ldamodel = Lda(doc_term_matrix, num_topics = 3, id2word =
                                      dictionary, passes=50)

print(ldamodel.print_topics(num_topics=3, num_words=3))
```

Listing 4.20 starts by initializing six variables `doc1` through `doc6` with a sentence, followed by the variable `all_docs`, which is an array consisting of those six variables.

The next portion of 6.20 contains several `import` statements, after which the variables `stop`, `exclude`, and `lemma` are initialized appropriately. Then the Python function `clean_documents()` is defined, which removes stop words and punctuation, and returns the sentence `normalized` that contains the cleaned words.

Next, the `gensim` library is imported and the variables `dictionary` and `doc_term_matrix` are initialized as a dictionary from `cleaned_docs` and a document/term matrix from `cleaned_docs`, respectively.

The final code block in Listing 4.20 initializes the variable `Lda` as an instance of the class `LdaModel`. The final code snippet initializes the variable `ldamodel` as an instance of `Lda` with four parameters, as shown here:

```
ldamodel = Lda(doc_term_matrix, num_topics = 3, id2word =
                                      dictionary, passes=50)
```

Launch the code in Listing 4.21 to see the following output:

```
model:
[
(0,'0.075*"experience" + 0.075*"result" +
                                    0.075*"judgement"'),
(1,'0.060*"plan" + 0.060*"merit" + 0.034*"wise"'),
(2,'0.106*"talked" + 0.061*"according" +
                                     0.061*"oscar"')
]
```

Notice that the final code snippet in Listing 4.20 specifies the value 3 for `num_topics` (the number of topics) and also for `num_words`. Hence, there are three elements in the preceding output: The first digit (shown in bold) of each element is the topic number. Each highlighted digit is followed by an element that is a linear combination of three strings. This is because the value of `num_words` is 3. The words in each linear combination are the most meaningful "topic" words, and the numeric coefficients indicate the relative weight (i.e., importance) of the associated word in the document.

Change the number 3 to a 4 in the final line of code in Listing 4.20 as follows:

```
print(ldamodel.print_topics(num_topics = 3, num_words = 4))
```

Launch the code in Listing 4.21 to see the following output:

```
model:
[
(0,'0.067*"talked" + 0.067*"according" + 0.067*"like" +
                                     0.038*"about"'),
(1,'0.065*"experience" + 0.065*"result" +
                     0.065*"judgement" + 0.037*"good"'),
(2,'0.111*"merit" + 0.111*"plan" + 0.063*"similar" +
                                    0.063*"meaning"')
]
```

Compare the preceding output block with the first output block and observe the differences.

A BRIEF COMPARISON OF POPULAR PYTHON-BASED NLP LIBRARIES

This section contains information that you have seen in earlier sections and chapters, which is provided in a consolidated manner for your convenience.

The natural language toolkit NLTK is used for such tasks as tokenization, lemmatization, stemming, parsing, and POS tagging. This library has tools for almost all NLP tasks.

SpaCy is the main competitor of the NLTK. These two libraries can be used for the same tasks.

Gensim is the package for topic and vector space modeling and document similarity analysis.

Sklearn provides a large library for machine learning. The tools for text preprocessing are also presented here.

The general mission of the Pattern library is to serve as the web mining module. So, it supports NLP only as a side task.

Polyglot is the yet another Python package for NLP. It is not very popular but can be used for a wide range of the NLP tasks.

MISCELLANEOUS LIBRARIES

This section contains task-specific libraries that can be used in NLP as well as non-NLP projects.

https://www.kdnuggets.com/2020/04/five-cool-python-libraries-data-science.html

1. Numerizer

   https://github.com/jaidevd/numerizer

   This library converts text numerics into `int` and `float`, and is installed via

```
pip3 install numerizer
```

A simple example is shown here:

```
from numerizer import numerize
print(numerize('Eight fifty million'))
print(numerize('one two three'))
```

```
print(numerize('Fifteen hundred'))
print(numerize('Three hundred and Forty five'))
print(numerize('Six and one quarter'))
print(numerize('Jack is having fifty million'))
print(numerize('Three hundred billion'))
```

2. Missingo

This library provides a way to visualize missing values from an Excel spreadsheet, and is installed via

```
pip3 install missingno
```

A simple example is shown here:

```
import pandas as pd
import missingno as mi

# reading the dummy dataset
data = pd.read_excel("dummy.xlsx")

# checking missing values
data.isnull().sum()

#Visualizing using missingno
print("Visualizing missing value using bar graph")
mi.bar(data, figsize = (10,5))
print("Visualizing missing value using matrix")
mi.matrix(data, figsize = (10,5))
```

3. Faker

This library generates various types of test data, and is installed via

```
pip3 install faker
```

A simple example is shown here:

```
# Generating fake email
print (fake.email())
# Generating fake country name
print(fake.country())
```

```
# Generating fake name
print(fake.name())
# Generating fake text
print(fake.text())
# Generating fake lat and lon
print(fake.latitude(), fake.longitude())
# Generating fake url
print(fake.url())
# Generating fake profile
print(fake.profile())
# Generating random number
print(fake.random_number())
```

4. EMOT

This library can collect emojis (small images) and emoticons (keyboard-based characters), and then perform a sentiment-like analysis. Install this library via this command:

```
pip3 install emot
```

A simple example is shown here:

```
import re
# Function for converting emojis into word
from emot.emo_unicode import UNICODE_EMO, EMOTICONS


def convert_emojis(text):
  for emot in UNICODE_EMO:
    text = text.replace(emot, "_".join(UNICODE_
    EMO[emot].replace(",","").replace(":","").split()))
 return text# Example


text1 = "Awesome üòÇ. The exhilaration from
successfully completing my project üòé, The euphoria
                                        is wonderfuls üòí"
convert_emojis(text1)
```

5. Chartify

   Chartify is a user-friendly visualization library for charts, and is installed via

```
pip3 install chartify
```

   A simple example is shown here:

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

#loading example dataset from chartify
data = chartify.examples.example_data()
data.head()

# Calculating total quantity for each fruits
quantity_by_fruit = (data.groupby('fruit')['quantity'].
                                    sum().reset_index())
ch = chartify.Chart(blank_labels = True, x_axis_type =
                                          'categorical')
ch.set_title("Vertical bar plot.")
ch.set_subtitle("Automatically sorts by value counts.")

ch.plot.bar(
  data_frame = quantity_by_fruit,
  categorical_columns = 'fruit',
  numeric_column = 'quantity')
ch.show()
```

SUMMARY

This chapter introduced you to NLTK, along with code samples of using NLTK with lxml, XPath, stemmers, lemmatization, and stop words. Then you learned about some of the features of Wordnet, such as finding synonyms and antonyms of words. You also learned about NLTK with POS, along with various tokenizers.

Next, you learned how define a grammar in NLTK and determine whether a given sentence can be parsed with that grammar. Finally, you learned about Gensim and its core concepts, with code samples that illustrate how to calculate tf-idf values in Gensim, and how to save a word2vec model in Gensim.

CHAPTER 5

ALGORITHMS AND TOOLKITS (II)

This chapter contains Python code samples for some of the NLP-related concepts that were introduced in the previous chapter. We have included an assortment of code samples involving regular expressions, the Python library BeautifulSoup, Scrapy, and various NLP-related Python code samples that use the spaCy library.

In addition, appendix A is entirely devoted to regular expressions, and you can find online introductory articles regarding the Sklearn Python library, both of which are relevant for some of the code samples in this chapter.

The first part of this chapter contains some examples of data cleaning that involve regular expressions. The second section contains some basic examples involving BoW, one-hot encoding, and word embeddings in Sklearn.

The third section discusses BeautifulSoup, which is a Python module for scraping HTML Web pages. This section contains some Python code samples that retrieve and then manipulate the contents of an HTML Web page from the GitHub code repository. This section also contains a brief introduction to Scrapy, which is a Python-based library that provides Web scraping functionality and various other APIs.

The fourth section introduces spaCy, which is a Python-based library for NLP, along with Python code samples that show various features of spaCy.

CLEANING DATA WITH REGULAR EXPRESSIONS

This section contains a simple preview of what you can accomplish with regular expressions when you need to clean your data. If you are new to regular

expressions, you can read Appendix A. The main concepts to understand for the code samples in this section are listed here:

- the range `[A-Z]` matches any uppercase letter
- the range `[a-z]` matches any lowercase letter
- the range `[a-zA-Z]` matches any lowercase or uppercase letter
- the range `[^a-zA-Z]` matches anything *except* lowercase or uppercase letters

Listing 5.1 shows the contents of `text_clean_regex.py` that illustrate how to remove any symbols that are not characters from a text string.

LISTING 5.1: text_clean_regex.py

```
import re # this is for regular expressions

text = "I have 123 apples for sale. Call me at 650-555-
                1212 or send me email at apples@acme.com."

print("text:")
print(text)
print()

# replace the '@' symbol with the string ' at ':
cleaned1 = re.sub('@', ' at ',text)
print("cleaned1:")
print(cleaned1)

# replace non-letters with ' ':
cleaned2 = re.sub('[^a-zA-Z]', ' ',cleaned1)
print("cleaned2:")
print(cleaned2)

# replace multiple adjacent spaces with a single ' ':
cleaned3 = re.sub('[ ]+', ' ',cleaned2)
print("cleaned3:")
print(cleaned3)
```

Listing 5.1 contains an `import` statement so that we can use regular expressions in the subsequent code. After initializing the variable `text` with a sentence and displaying its contents, the variable `cleaned1` is defined, which involves replacing the "@" symbol with the text string "at." (Notice the white space before and after the text "at"). Next, the variable `cleaned2`

is defined, which involves replacing anything that is not a letter with a white space. Finally, the variable `cleaned3` is defined, which involves "squeezing" multiple white spaces into a single white space. Launch the code in Listing 5.1 to see the following output:

```
text:
I have 123 apples for sale. Call me at 650-555-1212 or
send me email at apples@acme.com.

cleaned1:
I have 123 apples for sale. Call me at 650-555-1212 or
send me email at apples at acme.com.
cleaned2:
I have     apples for sale  Call me at              or
send me email at apples at acme com
cleaned3:
I have apples for sale Call me at or send me email at
apples at acme com
```

Listing 5.2 shows the contents of `text_clean_regex2.py` that illustrate how to remove HTML tags from a text string.

LISTING 5.2: text_clean_regex2.py

```
import re # this is for regular expressions

# [^>]:  matches anything except a '>'
# <[^>]: matches anything after '>' except a '>'
tagregex = re.compile(r'<[^>]+>')

def remove_html_tags(doc):
  return tagregex.sub(':', doc)

doc1 = "<html><head></head></body><p>paragraph1</p>
                              <div>div element</div></html>"

print("doc1:")
print(doc1)
print()
```

```
doc2 = remove_html_tags(doc1)
print("doc2:")
print(doc2)
```

Listing 5.2 contains an `import` statement, followed by the variable `tagregex`, which is a regular expression that matches a left angle bracket, <, followed by any character except for a right angle bracket, >. Next, the Python function `remove_html_tags()` removes all the HTML tags in a text string. The next portion of Listing 5.2 initializes the variable `doc1` as an HTML string and displays its contents. The final code block invokes the Python function `remove_html_tags()` to remove the HTML tags from `doc1`, and then prints the results. Launch the code in Listing 5.2 to see the following output:

```
doc1:
<html><head></head></body><p>paragraph1</p><div>div
                                        element</div></html>


doc2:
:::::paragraph1::div element::
```

The third (and final) code sample for this section also involves regular expressions, and it's useful when you need to remove contractions (e.g., replacing "that's" with "that is").

Listing 5.3 shows the contents of `text_clean_regex3.py` that illustrate how to replace contractions with the original words.

LISTING 5.3: text_clean_regex3.py

```
import re # this is for regular expressions

def clean_text(text):
  text = text.lower()

text = re.sub(r"I'm",     "I am", text)
text = re.sub(r"he's",    "he is", text)
text = re.sub(r"she's",   "she is", text)
text = re.sub(r"that's",  "that is", text)
text = re.sub(r"what's",  "what is", text)
text = re.sub(r"where's", "where is", text)
text = re.sub(r"how's",   "how is", text)
text = re.sub(r"it's",    "it is", text)
text = re.sub(r"\'ll",    " will", text)
```

```
text = re.sub(r"\'ve",     " have", text)
text = re.sub(r"\'re",     " are", text)
text = re.sub(r"\'d",      " would", text)
text = re.sub(r"n't",      "not", text)
text = re.sub(r"won't",    "will not", text)
text = re.sub(r"can't",    "can not", text)

text = re.sub(r"[-()\"#/@;:<>{}`+=~|.!?,]", "", text)
return text

sentences = ["It's a hot day and I'm sweating",
             "How's the zoom class going?",
             "She's smarter than me - that's a fact"]

for sent in sentences:
print("Sentence:",sent)
print("Cleaned: ",clean_text(sent))
print("----------------")
```

Listing 5.3 contains an `import` statement, followed by the Python function `clean_text()` that performs a brute-force replacement of hyphenated strings with their unhyphenated counterparts. The final code snippet in this function also removes any special characters in a text string.

The next portion of Listing 5.3 initializes the variable `sentences`, which contains multiple sentences, followed by a `for` loop that passes individual sentences to the Python `clean_text()` function. Launch the code in Listing 5.3 to see the following output:

```
Sentence: It's a hot day and I'm sweating
Cleaned:  it is a hot day and I am sweating
----------------
Sentence: How's the zoom class going?
Cleaned:  how is the zoom class going
----------------
Sentence: She's smarter than me - that's a fact
Cleaned:  she is smarter than me  that is a fact
----------------
```

The Python package `contractions` provides an alternative to regular expressions for expanding contractions, an example of which is shown later in this chapter.

Listing 5.4 shows the contents of `remove_urls.py` that illustrate how to remove URLs from an array of strings.

LISTING 5.4: remove_urls.py

```
import re
import pandas as pd

arr1 = ["https://www.yahoo.com",
        "http://www.acme.com"]

data = pd.DataFrame(data = arr1)

print("before:")
print(data)
print()

no_urls = []
for url in arr1:
  clean = re.sub(r"http\S+", "", url)
  no_urls.append(clean)

data["cleaned"] = no_urls

print("after:")
print(data)
```

Listing 5.4 starts with two `import` statements, followed by the initialization of the variable `arr1` with two URLs. Next, the variable `data` is a Pandas data frame whose contents are based on the contents of `arr1`. The contents of `data` are displayed, followed by a `for` loop that removes the `http` prefix from each element of `arr1`. The last code section appends a new column called `cleaned` to the `data` data frame and then displays the contents of that data frame. Launch the code in Listing 5.4 to see the following output:

```
before:
                                0
0  https://www.yahoo.com web page
1    http://www.acme.com web page
```

```
after:
                                    0    cleaned
0  https://www.yahoo.com web page   web page
1    http://www.acme.com web page   web page
```

This concludes the brief introduction to cleaning data with regular expressions, which are discussed in more detail (along with code samples) in one of the appendices.

HANDLING CONTRACTED WORDS

The previous section showed you how to expand English contractions via regular expressions. This section contains an example of expanding contractions via the Python package `contractions` that is available online:

https://github.com/kootenpv/contractions

Install the contractions package with this command:

```
pip3 install contractions
```

Listing 5.5 shows the contents of `contract.py` that illustrate how to expand English contractions and how to add custom expansion rules.

LISTING 5.5: contract.py

```
import contractions

sentences = ["what's new?",
             "how's the weather",
             "it's humid today",
             "we've been there before",
             "you should've been there!",
             "the sky's the limit"]

for sent in sentences:
  result = contractions.fix(sent)
  print("sentence:",sent)
  print("expanded:",result)
  print()

print("=> updating contraction rules...")
contractions.add("sky's", "sky is")
```

```
sent = "the sky's the limit"
result = contractions.fix(sent)
print("sentence:",sent)
print("expanded:",result)
print()
```

Listing 5.5 contains an `import` statement and then initializes the variable `sentences` to an array of text strings. The next portion of Listing 5.5 contains a loop that iterates through the array of strings in `sentences`, and then prints each sentence as well as the expanded version of the sentence.

In the output, the contraction `sky's` in the last sentence in the array `sentences` is not expanded, so let's add a new expansion rule, as shown here:

```
contractions.add("sky's", "sky is")
```

Now when we process this string again, the contraction `sky's` is expanded correctly. Launch the code in Listing 5.5 to see the following output:

```
sentence: what's new?
expanded: what is new?

sentence: how's the weather
expanded: how is the weather

sentence: it's humid today
expanded: it is humid today

sentence: we've been there before
expanded: we have been there before

sentence: you should've been there!
expanded: you should have been there!

sentence: the sky's the limit
expanded: the sky's the limit

=> updating contraction rules...
sentence: the sky's the limit
expanded: the sky is the limit
```

As an alternative, you can also write Python code to expand contractions, as shown in the following code block:

```
CONTRACT = {"how's":"how is", "what's":"what is",
"it's":"it is", "we've":"we have", "should've" :"should
                                have", "sky's": "sky is"}

sentences = ["what's new?",
             "how's the weather",
             "it's humid today",
             "we've been there before",
             "you should've been there!",
             "the sky's the limit"]

for sent in sentences:
  words = sent.split()
  expanded = [CONTRACT[w] if w in CONTRACT else w for w
                                                  in words]
  new_sent = " ".join(expanded)
  print("sentence:",sent)
  print("expanded:",new_sent)
  print()
```

The next section contains some Python-based code samples that involve the BoW (Bag of Words).

PYTHON CODE SAMPLES OF BoW

BoW is a technique for creating a numeric vector encoding of words. This section contains some simple Python-based code samples that illustrate how to perform this technique on a set of words.

The BoW algorithm counts how many times a word appears in a document, which generalizes a one-hot encoding of a set of words. Those word counts allow us to compare documents and gauge their similarities. The BoW algorithm can be used in applications like search, document classification, and topic modeling. In addition, BoW can be used to prepare text for input in a deep learning neural network.

Listing 5.6 shows the contents of `bow_to_vector1.py` that illustrate how to create a one-dimensional numeric vector based on a given vocabulary.

LISTING 5.6: bow_to_vector1.py

```
VOCAB = ['dog', 'cheese', 'cat', 'mouse']
TEXT1 = 'the mouse ate the cheese'
TEXT2 = 'the horse ate the hay'

def to_bow(text):
  words = text.split(" ")
  return [1 if w in words else 0 for w in VOCAB]
  print("VOCAB: ",VOCAB)
  print("TEXT1:",TEXT1)
  print("BOW1: ",to_bow(TEXT1)) # [0, 1, 0, 1]
  print("")

  print("TEXT2:",TEXT2)
  print("BOW2: ",to_bow(TEXT2)) # [0, 0, 0, 0]
```

Listing 5.6 initializes the variables `VOCAB`, `TEXT1`, and `TEXT2`, followed by the Python function `to_bow()` that constructs a BoW representation for text strings. Next, this Python function is invoked twice, first with `TEXT1` and then with `TEXT2`. Launch the code in Listing 5.6 to see the following output:

```
VOCAB:  ['dog', 'cheese', 'cat', 'mouse']
TEXT1: the mouse ate the cheese
BOW1:  [0, 1, 0, 1]

TEXT2: the horse ate the hay
BOW2:  [0, 0, 0, 0]
```

Listing 5.7 shows the contents of `bow_to_vector2.py` that illustrate how to create a one-dimensional numeric vector based on a given vocabulary.

LISTING 5.7: bow_to_vector2.py

```
VOCAB  = ['dog', 'cheese', 'cat', 'mouse']
MYTEXT = 'the mouse ate the cheese'
```

```
def to_bow(text):
  words = text.split(" ")
  found = [w if w in words else "MISS" for w in VOCAB]
  missing = [w if w not in words else "FOUND" for w in
                                                    VOCAB]
  print("Found:",found)
  print("Missing:",missing)

  return [1 if w in words else 0 for w in VOCAB]

print("MYTEXT:",MYTEXT)
print("VOCAB: ",VOCAB)
print("MYTEXT:",to_bow(MYTEXT))  # [0, 1, 0, 1]
```

Listing 5.7 extends the code in Listing 5.6 by keeping track of the words in the vocabulary that are not included in a given sentence. Launch the code in Listing 5.7 to see the following output:

```
MYTEXT: the mouse ate the cheese
VOCAB:  ['dog', 'cheese', 'cat', 'mouse']
Found: ['MISS', 'cheese', 'MISS', 'mouse']
Missing: ['dog', 'FOUND', 'cat', 'FOUND']
MYTEXT: [0, 1, 0, 1]
```

Listing 5.8 shows the contents of `count_vectorize.py` that illustrate how to create a one-dimensional numeric vector based on a given vocabulary.

LISTING 5.8: count_vectorize.py

```
VOCAB = ['dog', 'cheese', 'cat', 'mouse']
TEXT1 = 'the mouse ate the cheese'
TEXT2 = 'the horse ate the hay'

from sklearn.feature_extraction.text import
                                          CountVectorizer
vectorizer = CountVectorizer(vocabulary = VOCAB)

result1 = vectorizer.transform([TEXT1]).todense() #
                                      matrix([[0,1,0,1]])
result2 = vectorizer.transform([TEXT2]).todense() #
                                      matrix([[0,1,0,1]])
```

```
print("VOCAB: ",VOCAB)
print("TEXT1:",TEXT1)
print("BOW1: ",result1)  # [0, 1, 0, 1]
print("")

print("TEXT2:",TEXT2)
print("BOW2: ",result2)  # [0, 1, 0, 1]
print()

print("=> contents of countvectorizer:")
print(vectorizer)
```

Listing 5.8 performs the same functionality as Listing 5.6 by means of the `CountVectorizer` class in Sklearn instead of the custom code in Listing 5.6. Launch the code in Listing 5.8 to see the following output, which is the same output from `bow_to_vector1.py`:

```
VOCAB:  ['dog', 'cheese', 'cat', 'mouse']
TEXT1: the mouse ate the cheese
BOW1:  [[0 1 0 1]]

TEXT2: the horse ate the hay
BOW2:  [[0 0 0 0]]

=> contents of countvectorizer:
CountVectorizer(analyzer = 'word', binary = False,
                                decode_error = 'strict',
                dtype = <class 'numpy.int64'>, encoding =
                             'utf-8', input = 'content',
                 lowercase = True, max_df = 1.0, max_
                            features = None, min_df = 1,
                 ngram_range = (1, 1), preprocessor =
                                None, stop_words = None,
                 strip_accents = None, token_pattern =
                                    '(?u)\\b\\w\\w+\\b',
                 tokenizer = None, vocabulary = ['dog',
                              'cheese', 'cat', 'mouse'])
```

As you can see from the preceding output, the vocabulary of the variable `vectorizer` matches the contents of the variable `VOCAB`.

Listing 5.9 contains a BoW example that illustrates how to use BoW with a Pandas data frame for an array of sentences.

LISTING 5.9: bow_pandas.py

```
import pandas as pd
from sklearn.feature_extraction.text import
                                              CountVectorizer

sent = ["this is a sentence with text and very simple",
        "a second sentence with text and listed second",
        "a third sentence with text and listed third",
        "a final sentence with text"]

print("=> list of sentences:")
for s in sent:
  print(s)
print()

bow = CountVectorizer()
bow_fit = bow.fit_transform(sent)
bag_words = pd.DataFrame(bow_fit.toarray())
bag_words.columns = bow.get_feature_names()
print("=> bag_words for the sentences:")
print(bag_words)
```

Listing 5.9 initializes the variable `sent` as an array of sentences that are subsequently displayed via a `for` loop. Next, the variable `bow` is initialized as an instance of the `CountVectorizer` class that is provided by the Sklearn library. The `bow_fit` variable is assigned the result of transforming and fitting the data in the `sent` variable. Next, the `bag_words` variable is initialized as a data frame that contains the data in `bow_fit`.

After specifying the column names for the `bag_words` data frame, the contents of `bag_words` are displayed via a `print()` statement. Launch the code in Listing 5.9 to see the following output (unfortunately, the output is wrapped because it's too wide for the page):

```
=> list of sentences:
this is a sentence with text and very simple
a second sentence with text and listed second
a third sentence with text and listed third
a final sentence with text
```

```
=> bag_words for the sentences:
   and final is listed second sentence simple text third this
very with
0    1    0    1    0    0    1    1    1    0    1    1    1
1    1    0    0    1    2    1    0    1    0    0    0    1
2    1    0    0    1    0    1    0    1    2    0    0    1
3    0    1    0    0    0    1    0    1    0    0    0    1
```

In Chapter 4, we show how to create a BoW model using the NLTK library, and in Chapter 6 we give an example of BoW using TF2/Keras.

ONE-HOT ENCODING EXAMPLES

Recall from Chapter 4 that a one-hot encoding for a set of words involves one numeric vector for each word, and each vector contains a single value of 1 and the other values are all 0. As a simple example, the sentence "I love thick pizza" can be tokenized as `["i", "love", "thick", "pizza"]`, and one-hot encoded as follows:

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

Based on the preceding one-hot encoding, the sentence "We also love thick pizza" can be encoded as

```
[0,1,1,1] = [0,1,0,0] + [0,0,1,0] + [0,0,0,1]
```

The left-side vector `[0,1,1,1]` is the sum of the three vectors that represent the one-hot encoding of the words "love," "thick," and "pizza," respectively. The first index of this vector is 0 because this sentence contains "we" instead of "i."

Listing 5.10 shows the contents of `onehot_encode.py` that illustrate how to perform a one-hot encoding on a set of words.

LISTING 5.10: onehot_encode.py

```
import numpy as np

CLASSES = list(np.array([3, 1, 2]))
```

```
# The dataset labels
LABELS = np.array([1, 2, 3, 1, 2, 1, 1, 2, 3])
VALUES = [3, 3, 1, 1, 2, 2]
ONEHOT = np.zeros((len(LABELS), len(CLASSES)))

for idx, value in enumerate(LABELS):
  print("idx:",idx,"value:",value)
  ONEHOT[idx, CLASSES.index(value)] = 1

print("One-hot Encoding:")
print(ONEHOT)
```

Listing 5.10 initializes the variable `CLASSES` as a list created from a NumPy array that contains the numbers 3, 1, and 2. The next code block initializes the variables `LABELS`, `VALUES`, and `ONEHOT`, followed by a `for` loop that initializes each row of `ONEHOT` as a one-hot encoded vector. Launch the code in Listing 5.10 to see the following output:

```
idx: 0 value: 1
idx: 1 value: 2
idx: 2 value: 3
idx: 3 value: 1
idx: 4 value: 2
idx: 5 value: 1
idx: 6 value: 1
idx: 7 value: 2
idx: 8 value: 3
One-hot Encoding:
[[0. 1. 0.]
 [0. 0. 1.]
 [1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]
 [0. 1. 0.]
 [0. 1. 0.]
 [0. 0. 1.]
 [1. 0. 0.]]
```

SKLEARN AND WORD EMBEDDING EXAMPLES

Word embeddings pertain to algorithms that represent words (as well as documents) using a dense vector representation, which is a distributed representation. By contrast, vector representations refer to discrete representations, such as the use of the tf-idf values of words to generate vector representations of words.

Earlier in this chapter, you saw an example of word vectorization, and this section contains a consolidated list of word embedding techniques, along with code samples that illustrate how to use these techniques:

- Count Vectorizer
- TF-IDF Vectorizer
- Hashing Vectorizer
- Word2Vec (Gensim)

Listing 5.11 shows the contents of `count_vectorize2.py` that illustrate how to vectorize an array of sentences using the `CountVectorizer` class in Sklearn.

LISTING 5.11: count_vectorize2.py

```
from sklearn.feature_extraction.text import
                                        CountVectorizer
import matplotlib.pyplot as plt
import seaborn as sns

sentences = ['the mouse ate the cheese',
             'the horse ate the hay',
             'the mouse saw the horse',
             'the mouse scared the horse']

vectorizer = CountVectorizer()
sentence_vectors = vectorizer.fit_transform(sentences)

print("vectorized sentences:")
print(sentence_vectors.toarray())
print()
```

```
# learn vocabulary and store CountVectorizer
# sparse matrix in term_frequencies
term_frequencies = vectorizer.fit_transform(sentences)
vocab = vectorizer.get_feature_names()

# convert sparse matrix to numpy array
term_frequencies = term_frequencies.toarray()

# plot #1: visualize term frequencies
sns.heatmap(term_frequencies,annot = True,cbar = False,
                                     xticklabels = vocab)
plt.show()

# plot #2: visualize "one hot" term frequencies
# one_hot_vectorizer = CountVectorizer(binary = True)
# one_hot = one_hot_vectorizer.fit_transform
                                    (sentences).toarray()
# vocab = one_hot_vectorizer.get_feature_names()
# sns.heatmap(one_hot,annot = True,cbar = False,
                                     xticklabels = vocab)
# plt.show()
```

Listing 5.11 contains several `import` statements, followed by the variable `sentences` that is initialized as an array of sentences. Next, the variable `vectorizer` is an instance of the `CountVectorizer` class in Sklearn, followed by assigning the result of fitting and transforming its contents to the variable `sentence_vectors`.

The next code snippet initializes the variable `term_frequencies` with the term frequencies of the contents of `sentences`. In addition, the variable `vocab` is assigned the array of unique words that appear in `sentences`. Launch the code in Listing 5.11 to see the following output:

```
vectorized sentences:
[[1 1 0 0 1 0 0 2]
 [1 0 1 1 0 0 0 2]
 [0 0 0 1 1 1 0 2]
 [0 0 0 1 1 0 1 2]]
```

```
vocabulary:
['ate', 'cheese', 'hay', 'horse', 'mouse', 'saw',
'scared', 'the']
```

In the preceding output, the word "the" appears twice in every row because this word occurs in every row of the variable sentences.

Figure 5.1 shows the term frequencies, where the rows are the "documents" in the variable `sentences`, and the columns are the distinct words.

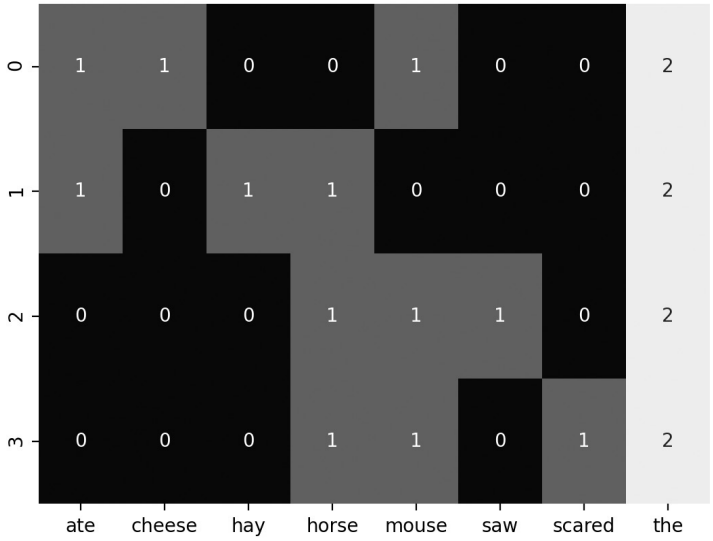

FIGURE 5.1 A matrix with term frequencies.

Uncomment the final block of code in Listing 5.11 to see the one hot encoding of the word frequencies.

Another way to vectorize an array of sentences involves the `CountVectorizer` class. Listing 5.12 shows the contents of `hashing_vectorize.py` that illustrate how to vectorize an array of sentences using the `CountVectorizer` class in Sklearn.

LISTING 5.12: hashing_vectorize.py

```
from sklearn.feature_extraction.text import
                                        HashingVectorizer

sentences = ['the mouse ate the cheese',
             'the horse ate the hay',
```

```
              'the mouse saw the horse',
              'the mouse scared the horse']

vectorizer = HashingVectorizer(norm = None, n_features = 8)

sentence_vectors = vectorizer.fit_transform(sentences)

print("vectorized sentences:")
print(sentence_vectors.toarray())
```

Listing 5.12 initializes the variable `vectorizer` as an instance of the `HashingVector` class in Sklearn, after which the variable `sentence_vectors` is assigned the result of transforming and fitting the contents of the `sentences` variable. Launch the code in Listing 5.12 to see the following output:

```
vectorized sentences:
[[ 0. -1.  0.  1.  1.  0. -2.  0.]
 [-2.  0.  0.  1.  0.  0. -2.  0.]
 [-1.  0.  0. -1.  1.  0. -2.  0.]
 [-1.  0. -1.  0.  1.  0. -2.  0.]]
```

A third way to vectorize an array of sentences involves the `TfidfVectorizer` class. Listing 5.13 shows the contents of `tfidf_vectorize.py` that illustrate how to vectorize an array of sentences using the `TfidfVectorizer` class in Sklearn.

LISTING 5.13: tfidf_vectorize.py

```
from sklearn.feature_extraction.text import TfidfVectorizer
import matplotlib.pyplot as plt
import seaborn as sns

sentences = ['the mouse ate the cheese',
             'the horse ate the hay',
             'the mouse saw the horse',
             'the mouse scared the horse']

vectorizer = TfidfVectorizer(norm = False, smooth_idf = False)
sentence_vectors = vectorizer.fit_transform(sentences)

print("vectorized sentences:")
print(sentence_vectors.toarray())

tfidf = vectorizer.fit_transform(sentences).toarray()
```

```
# display the tfidf frequencies:
sns.heatmap(tfidf,annot = True,cbar = False)
plt.show()
```

Listing 5.13 is similar to Listing 5.12, with the `TfidfVector` class in place of the `HashingVector` class. The other difference is the Seaborn Python library for visualization, which illustrates how to create a heat map in Seaborn. Note that Appendix F contains more information regarding Seaborn.

Launch the code in Listing 5.13 to see the following output (numbers have been truncated from 8 decimal places to 3 decimal places to avoid "wrapping" the vectors on multiple lines):

```
vectorized sentences:
[[1.693 2.386 0.    0.    1.287 0.    0.    2. ]
 [1.693 0.    2.386 1.287 0.    0.    0.    2. ]
 [0.    0.    0.    1.287 1.287 2.386 0.    2. ]
 [0.    0.    0.    1.287 1.287 0.    2.386 2. ]]
```

Figure 5.2 shows the `tf-idf` frequencies, where the rows are the "documents" in the variable `sentences`, and the columns are the distinct words.

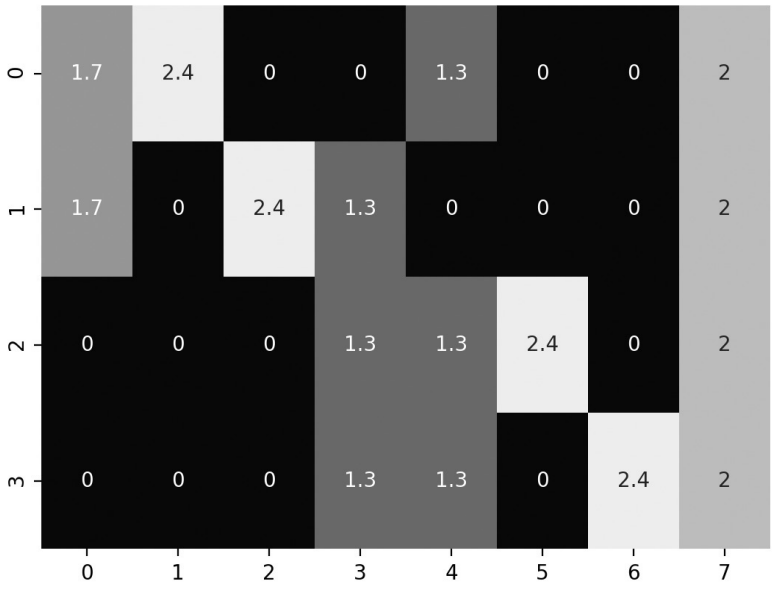

FIGURE 5.2 A heatmap with the tf-idf frequencies.

Listing 5.14 shows the contents of `gensim_vectorize.py` that illustrate how to vectorize an array of sentences using the word2vec class in Gensim.

LISTING 5.14: gensim_vectorize.py

```
from gensim.models import word2vec

sentences = ['the mouse ate the cheese',
             'the horse ate the hay',
             'the mouse saw the horse',
             'the mouse scared the horse']

for i, sentence in enumerate(sentences):
  tokenized = []
  for word in sentence.split(' '):
    tokenized.append(word)
  sentences[i] = tokenized

model = word2vec.Word2Vec(sentences,workers = 1,size = 2,
                          min_count = 1, window = 3, sg = 0)

word_list = model.wv.most_similar('mouse')
print("list of words:")
for word in word_list:
  print(word)
print()

similar_word = model.wv.most_similar('mouse')[0]
print("Most common word to mouse is: {}".format(similar_
                                                word[0]))
```

Listing 5.14 initializes the variable `sentences` as an array of sentences, followed by a `for` loop that tokenizes each sentence in the variable `sentences`. Next, the variable `model` is initialized as an instance of the `word2Vec` class that is available in the word2vec module in the Gensim library.

The next code snippet initializes the variable `word_list` with the words that are most similar to the word `mouse` (which is a word that appears in `sentences`), followed by a loop that displays those words. Launch the code in Listing 5.14 to see the following output:

```
list of words:
('hay', 0.1007559671998024)
('the', -0.37532204389572144)
```

```
('saw', -0.528710126876831)
('horse', -0.7526397109031677)
('scared', -0.7596074342727661)
('ate', -0.9832854270935059)
('cheese', -0.9918226003646851)

Most common word to mouse is: hay
```

Listing 5.15 shows the contents of `tfidf_l1_l2.py` that illustrate how to specify L1 or L2 normalized frequencies when generating word embeddings of words in a corpus. If you are unfamiliar with L1 or L2 normalization, you can treat this code sample as optional with no loss of continuity.

LISTING 5.15: tfidf_l1_l2.py

```
from sklearn.feature_extraction.text import TfidfVectorizer
import pandas as pd

text = ['the mouse ate the cheese',
        'the horse ate the hay',
        'the mouse saw the horse',
        'the mouse scared the horse']

# use L1 penalty:
tfidf1 = TfidfVectorizer(binary = False,norm = 'l1',
use_idf = False,smooth_idf = False,lowercase = True,
stop_words = 'english',min_df = 1,max_df = 1.0,max_
                      features = None,ngram_range = (1,1))

df = pd.DataFrame(tfidf1.fit_transform(text).toarray(),
                      columns = tfidf1.get_feature_names())

print("dataframe #1:")
print(df)
print()

# use L2 penalty:
tfidf2 = TfidfVectorizer(binary = False,norm = 'l2',use_idf =
False,smooth_idf = False,lowercase = True,stop_words =
'english',min_df = 1,max_df = 1.0,max_features =
                                  None,ngram_range = (1,1))

df = pd.DataFrame(tfidf2.fit_transform(text).toarray(),
                      columns = tfidf2.get_feature_names())
```

```
print("dataframe #2:")
print(df)
print()
```

Listing 5.15 contains a variation of the `TfidfVectorizer` class (displayed in Listing 5.13) that includes the parameter `norm` whose values can be either `l1` or `l2` that correspond to L1 or L2 normalization, respectively. The variable `tfidf1` is initialized as an instance of `TfidfVectorizer` with `norm` set equal to l1, and then the data frame `df` is initialized with the result of transforming and fitting the contents of the array text (which is an array of sentences). The variable `tfidf2` is similar to `tfidf1`, except that `norm` is set to the value `l2` instead of `l1`. Launch the code in Listing 5.15 to see the following output:

```
dataframe #1:
        ate    cheese       hay     horse     mouse       saw
scared
0  0.333333  0.333333  0.000000  0.000000  0.333333  0.000000
0.000000
1  0.333333  0.000000  0.333333  0.333333  0.000000  0.000000
0.000000
2  0.000000  0.000000  0.000000  0.333333  0.333333  0.333333
0.000000
3  0.000000  0.000000  0.000000  0.333333  0.333333  0.000000
0.333333


dataframe #2:
        ate    cheese       hay     horse     mouse      saw
scared
0   0.57735   0.57735   0.00000   0.00000   0.57735   0.00000
0.00000
1   0.57735   0.00000   0.57735   0.57735   0.00000   0.00000
0.00000
2   0.00000   0.00000   0.00000   0.57735   0.57735   0.57735
0.00000
3   0.00000   0.00000   0.00000   0.57735   0.57735   0.00000
0.57735
```

WHAT IS BEAUTIFULSOUP?

BeautifulSoup is a useful Python module that provides a plethora of APIs for retrieving the contents of HTML Web pages and extracting subsets of their content using XPath-based expressions. You need some familiarity with

regular expressions, which are discussed in Appendix A. If need be, you can find online tutorials that discuss basic concepts of XPath.

This section contains three code samples: how to retrieve the contents of an HTML Webpage, how to display the contents of HTML anchor ("a") tags, and how to remove non-alphanumeric characters from HTML anchor ("a") tags.

Listing 5.16 shows the contents of `scrape_text1.py` that illustrate how to retrieve the contents of *https://www.github.com*.

LISTING 5.16: scrape_text1.py

```
import requests
import re
from bs4 import BeautifulSoup

src = "https://www.github.com"

# retrieve html web page as text
text = requests.get(src).text
#print("text:",text)

# parse into BeautifulSoup object
soup = BeautifulSoup(text, "html.parser")
print("soup:",soup)
```

Listing 5.16 contains `import` statements, followed by initializing the variable `src` as the URL for Github. Next, the variable `text` is initialized with the contents of the Github page, and then the variable `soup` is initialized as an instance of the `BeautifulSoup` class. Notice that `html.parser` is specified, which is why the HTML tags are removed. Launch the code in Listing 5.16 to see 1,036 lines of output. Only the first portion of the output is shown.

```
soup:
<!DOCTYPE html>


<html lang = "en">
<head>
<meta charset = "utf-8"/>
<link href = "https://github.githubassets.com" rel =
                                          "dns-prefetch"/>
<link href = "https://avatars0.githubusercontent.com"
                                     rel = "dns-prefetch"/>
```

```
   <link href = "https://avatars1.githubusercontent.com"
                                        rel = "dns-prefetch"/>
   <link href = "https://avatars2.githubusercontent.com"
                                        rel = "dns-prefetch"/>
   <link href = "https://avatars3.githubusercontent.com"
                                        rel = "dns-prefetch"/>
   <link href = "https://github-cloud.s3.amazonaws.com"
                                        rel = "dns-prefetch"/>
   <link href = "https://user-images.githubusercontent.com/"
                                        rel = "dns-prefetch"/>
   <link crossorigin = "anonymous" href = "https://github.
   githubassets.com/assets/frameworks-146fab5ea30e8afac08d
   d11013bb4ee0.css" integrity = "sha512-FG+rXqMOivrAjdEQE
   7tO4BwM1poGmg70hJFTlNSxjX87grtrZ6UnPR8NkzwUHlQEGviu9XuR
   YeO8zH9YwvZhdg==" media = "all" rel = "stylesheet">
```

Listing 5.17 shows the contents of `scrape_text2.py` that illustrate how to retrieve the contents of *https://www.github.com* and display the contents of the HTML anchor ("a") tags. The new code is shown in bold.

LISTING 5.17: scrape_text2.py

```
import requests
import re
from bs4 import BeautifulSoup

src = "https://www.github.com"

# retrieve html web page as text
text = requests.get(src).text
#print("text:",text)

# parse into BeautifulSoup object
soup = BeautifulSoup(text, "html.parser")
print("soup:",soup)

# display contents of anchors ("a"):
for item in soup.find_all("a"):
  if len(item.contents) > 0:
    print("anchor:",item.get('href'))
```

Listing 5.17 contains three `import` statements and then initializes the variable src with the URL for Github. Next, the variable `text` is initialized with the contents of the Github page. The next snippet initializes the variable `soup` as an instance of the `BeautifulSoup` class. The final block of code is a `for` loop (shown in bold) that iterates through all the `<a>` elements in the variable `text`, and displays the `href` value embedded in each `<a>` element (after determining that its contents are non-empty).

Launch the code in Listing 5.17 to see 102 lines of output. Only the first portion of the output is shown.

```
anchor: #start-of-content
anchor: https://help.github.com/articles/supported-
browsers
anchor: https://github.com/
anchor: /join?ref_cta = Sign+up&ref_loc =
header+logged+out&ref_page = %2F&source = header-home
anchor: /features
anchor: /features/code-review/
anchor: /features/project-management/
anchor: /features/integrations
anchor: /features/actions
anchor: /features/packages
anchor: /features/security
anchor: /features#team-management
anchor: /features#hosting
anchor: /customer-stories
anchor: /security
anchor: /team
```

Listing 5.18 shows the contents of `scrape_text3.py` that illustrate how to remove the nonalphanumeric characters from the HTML anchor ("a") tags in the Webpage *https://www.github.com*. The new code is shown in bold.

LISTING 5.18: scrape_text3.py

```
import requests
import re
from bs4 import BeautifulSoup
```

```
# removes non-alphanumeric characters
def remove_non_alpha_chars(text):
  # define the pattern to keep
  regex = r'[^a-zA-Z0-9]'
  return re.sub(regex, '', text)

src = "https://www.github.com"

# retrieve html web page as text
text = requests.get(src).text
#print("text:",text)

# parse into BeautifulSoup object
soup = BeautifulSoup(text, "html.parser")
print("soup:",soup)
# display contents of anchors ("a"):
for item in soup.find_all("a"):
  if len(item.contents) > 0:
   #print("anchor:",item.get('href'))
    cleaned = remove_non_alpha_chars(item.get('href'))
    print("cleaned:",cleaned)
```

Listing 5.18 is similar to Listing 5.16, but it differs in the `for` loop (shown in bold) and it displays the `href` values after removing non-alphabetic characters and all the whitespaces. Launch the code in Listing 5.18 to see 102 lines of output. Only the first portion of the output is shown here.

```
cleaned: startofcontent
cleaned: httpshelpgithubcomarticlessupportedbrowsers
cleaned: httpsgithubcom
cleaned: joinrefctaSignupreflocheaderloggedout
refpage2Fsourceheaderhome
cleaned: features
cleaned: featurescodereview
cleaned: featuresprojectmanagement
cleaned: featuresintegrations
cleaned: featuresactions
cleaned: featurespackages
cleaned: featuressecurity
cleaned: featuresteammanagement
cleaned: featureshosting
```

```
cleaned: customerstories
cleaned: security
cleaned: team
```

WEB SCRAPING WITH PURE REGULAR EXPRESSIONS

The previous section contains several examples of using BeautifulSoup to scrape Webpages, and this section contains an example that involves only regular expressions. You need some familiarity with regular expressions, which are discussed in one of the appendices.

Listing 5.19 shows the contents of `scrape_pure_regex.py` that illustrate how to retrieve the contents of *https://www.github.com* and remove the HTML tags with a single regular expression.

LISTING 5.19: scrape_pure_regex.py

```
import requests
import requests
import re
import os

src = "https://www.github.com"

# retrieve the web page contents:
r = requests.get(src)
print(r.text)

# remove HTML tags (notice the "?"):
pattern = re.compile(r'<.*?>')

cleaned = pattern.sub('', r.text)

#remove leading whitespaces:
cleaned = os.linesep.join([s.lstrip() for s in cleaned.
                                        splitlines() if s])

#remove embedded blank lines:
cleaned = os.linesep.join([s for s in cleaned.splitlines()
                                                    if s])
print("cleaned text:")
print(cleaned)
```

```
#remove ALL whitespaces:
#cleaned = cleaned.replace(" ", "")

#this does not work:
#cleaned = cleaned.trim(" ", "")
```

Listing 5.19 also removes the HTML elements from the variable `text` that contains the contents of the Github Webpage. However, there is a subtle yet important detail regarding the regular expression `<.*>` versus the regular expression `<.*?>`.

The regular expression `<.*>` performs a *greedy* match, which means that it will continue matching characters until the right-most ">" is encountered. However, we want the greedy match to *stop* after finding the first ">" character, which matches the previous "<" character. The solution is simple. Specify the regular expression `<.*?>`, which contains a question mark ("?") whose purpose is to disable the greedy matching nature of the metacharacter "*."

Launch the code in Listing 5.19 to see the following output (some output omitted for brevity):

```
<!DOCTYPE html>
<html lang = "en">
  <head>
    <meta charset = "utf-8">
  <link rel = "dns-prefetch" href = "https://github.
                                      githubassets.com">
  <link rel = "dns-prefetch" href = "https://avatars0.
                                    githubusercontent.com">
  <link rel = "dns-prefetch" href = "https://avatars1.
                                    githubusercontent.com">
  <link rel = "dns-prefetch" href = "https://avatars2.
                                    githubusercontent.com">
  <link rel = "dns-prefetch" href = "https://avatars3.
                                    githubusercontent.com">
  <link rel = "dns-prefetch" href = "https://github-
                                  cloud.s3.amazonaws.com">
  <link rel = "dns-prefetch" href = "https://user-
                          images.githubusercontent.com/">
```

```
  <link crossorigin = "anonymous" media = "all" integrity =
"sha512-/uy49LxdzjR0L36uT6CnmV1omP/8ZHxvOg4zq/dczzABHq9atn
tjJDmo5B7sV0J+AwVmv0fR0ZyW3EQawzdLFA==" rel = "stylesheet"
href = "https://github.githubassets.com/assets/frameworks-
                    feecb8f4bc5dce34742f7eae4fa0a799.css"/>
  <link crossorigin = "anonymous" media = "all"
integrity = "sha512-37pLQI8klDWPjWVVWFB9ITJLwVTTkp3Rt4b
Vf+yixrViURK9OoGHEJDbTLxBv/rTJhsLm8pb00H2H5AG3hUJfg=="
rel = "stylesheet" href = "https://github.githubassets.
com/assets/site-dfba4b408f2494358f8d655558507d21.css"/>
  <meta name = "viewport" content = "width = device-width">

  <title>The world's leading software development
platform · GitHub</title>

[details omitted for brevity]
    <div class = "position-relative js-header-wrapper">
      <a href = "#start-of-content" class = "px-2 py-4
bg-blue text-white show-on-focus js-skip-to-content">Skip
                                          to content</a>
      <span class = "Progress progress-pjax-loader
         position-fixed width-full js-pjax-loader-bar">
        <span class = "progress-pjax-loader-bar top-0
                    left-0" style = "width: 0%;"></span>
      </span>
[details omitted for brevity]
cleaned text:
The world's leading software development platform ·
GitHub
Skip to content
GitHub no longer supports this web browser.
Learn more about the browsers we support.
<a
href = "/join?ref_cta=Sign+up&amp;ref_loc = header+
logged+out&amp;ref_page = %2F&amp;source = header-home"
[details omitted for brevity]
```

WHAT IS SCRAPY?

In the previous section, you learned that BeautifulSoup is a Python-based library for scraping Web pages. BeautifulSoup also supports XPath (which is an integral component of XSLT), whose APIs enable you to parse the scraped data and to extract portions of that data.

Scrapy is a Python-based library that provides data extraction and an assortment of additional APIs for a wide range of operations, including redirections, HTTP caching, filtering duplicated requests, preserving sessions/cookies across multiple requests, and various other features. Scrapy supports both CSS selectors and XPath expressions for data extraction. Moreover, you can also use BeautifulSoup or PyQuery as a data extraction mechanism.

While Scrapy and BeautifulSoup can do some of the same things (i.e., Web scraping), they have fundamentally different purposes. As a rule of thumb, use BeautifulSoup if you need a "one-off" Webpage scraper. However, if you need to perform Web scraping and perform additional operations for one or more webpages, then Scrapy is probably a better choice.

Scrapy is more difficult to master than BeautifulSoup, so decide whether the extra features of Scrapy are necessary for your requirements before you invest your time learning it. The Scrapy documentation page is available online:

https://doc.scrapy.org/en/latest/intro/tutorial.html

WHAT IS SpaCy?

The spaCy NLP library is an efficient Python-based library that provides many useful features, and its homepage is *spacy.io*.

The spaCy library provides support for many NLP-related tasks, some of which are listed here:

- NER (Named Entity Recognition)
- POS tagging
- support for more than 60 languages
- more than 40 statistical models
- pretrained word vectors

You might encounter the following error message if you do not have the right components installed:

```
OSError: [E050] Can't find model 'en'. It doesn't seem
to be a shortcut link, a Python package or a valid path
to a data directory.
```

The solution for the preceding error is to open a command shell and launch the following command:

```
python3 -m spacy download en
```

After the preceding command has been successfully completed, you will see this message:

```
You can now load the model via spacy.load('en')
```

SpaCy version 3.0 was released as this book went to print. Information is available online at

https://explosion.ai/blog/spacy-v3

The latest version contains significant new features, including transformer-based pipelines. The preceding link contains additional links that contain more information about the latest release.

SpaCy AND STOP WORDS

Listing 5.20 shows the contents of `spacy_stopwords.py` that illustrate how to find the stop words in a sentence.

LISTING 5.20: spacy_stopwords.py

```
import spacy

import spacy
from spacy import displacy
from spacy.lang.en.stop_words import STOP_WORDS

sentence = "This simple sentence contains a stop word or
                                                two or more"

nlp = spacy.load('en')
doc = nlp(sentence)

print("sentence: ",sentence)
non_stop = []
```

```
for word in doc:
  if word.is_stop == True:
    print("stop word:",word)
  else:
    non_stop.append(word)
print("Non-stop: ",non_stop)
```

Listing 5.20 contains `import` statements for `spaCy`, `displaCy` (the latter is for visualization), and for English stop words. The next code snippet initializes three variables. First, the variable `sentence` is initialized as a text string. Second, `nlp` is initialized as an instance of the English language model. Next, the variable `doc` is initialized with the result of processing the text in the variable `sentence`.

The last portion of Listing 5.20 contains a loop that displays the stop words in the variable `sentence`. Launch the code in Listing 5.20 to see the following output:

```
sentence:  This simple sentence contains a stop word or
two or more
stop word: This
stop word: a
stop word: or
stop word: two
stop word: or
stop word: more
Non-stop:  [simple, sentence, contains, stop, word]
```

SpaCy AND TOKENIZATION

Listing 5.21 shows the contents of `spacy_tokenize.py` that illustrate how to tokenize a sentence.

LISTING 5.21: spacy_tokenize.py

```
import spacy
nlp = spacy.load('en')

doc = nlp("I love Chicago deep dish pizza and Uno's as well.")
```

```
for token in doc:
  print(token)

print(f"Token \t\tLemma \t\tStopword".format('Token','Lemma',
'Stopword'))
print("-"*40)

for token in doc:
  print(f"{str(token)}\t\t{token.lemma_}\t\t{token.is_stop}")
```

Listing 5.21 contains an `import` statement followed by the initialization of the variable `nlp` as an instance of the English model in spaCy. Note that this code snippet will download the data model if it is not already present on your machine. The next portion of Listing 5.21 displays the tokens (words) in the initial text string, followed by a tabular display in which each row displays a token, its lemmatization, and whether it's a stop word. Launch the code in Listing 5.21 to see the following output:

```
I
love
Chicago
deep
dish
pizza
and
Uno
's
as
well
.
Token           Lemma           Stopword
----------------------------------------
I               -PRON-          True
love            love            False
Chicago         Chicago         False
deep            deep            False
dish            dish            False
```

```
pizza           pizza           False
and             and             True
Uno             Uno             False
's              's              True
as              as              True
well            well            True
.               .               False
```

SpaCy AND LEMMATIZATION

The code sample in this section shows how to perform lemmatization in spaCy. Listing 5.22 shows the contents of `spacy_lemma.py` that illustrate how to perform lemmatization in spaCy.

LISTING 5.22: spacy_lemma.py

```
import spacy

nlp = spacy.load("en_core_web_sm")

text = "I love Chicago deep dish pizza and Uno's as well."
 doc = nlp(text)

print("=> text, lemma_, pos_, and is_stop:")
for token in doc:
  print(token.text, token.lemma_, token.pos_, token.is_stop)
print()

import pandas as pd

cols = ("text", "lemma", "POS", "explain", "stopword")
rows = []

for t in doc:
  row = [t.text, t.lemma_, t.pos_, spacy.explain(t.pos_),
                                                  t.is_stop]
  rows.append(row)
```

```
df = pd.DataFrame(rows, columns = cols)
print("=> dataframe:")
print(df)
```

Listing 5.22 initializes the variable `nlp` with the spaCy model `en_core_web_sm` and the variable `text` with the same sentence as previous code samples. Next, the variable `doc` is instantiated by specifying the variable `text`, followed by a loop that displays the part of speech of each token (i.e., word) in the variable `text`.

The next portion of Listing 5.22 initializes the variable `cols` as the headings for the columns of a data frame, and the variable `rows` as an empty array. The following loop appends a row of attributes for each token in the variable `text`. Finally, the last code block initializes the data frame `df` with the contents of the variable `rows` and then displays its contents. Launch the code in Listing 5.22 to see the following output:

```
=> text, lemma_, pos_, and is_stop:
I -PRON- PRON True
love love VERB False
Chicago Chicago PROPN False
deep deep ADJ False
dish dish NOUN False
pizza pizza NOUN False
and and CCONJ True
Uno Uno PROPN False
's 's PART True
as as ADV True
well well ADV True
. . PUNCT False

=> dataframe:
      text    lemma    POS            explain  stopword
0        I   -PRON-   PRON            pronoun      True
1     love     love   VERB               verb     False
2  Chicago  Chicago  PROPN        proper noun     False
3     deep     deep    ADJ          adjective     False
4     dish     dish   NOUN               noun     False
```

```
5    pizza    pizza   NOUN              noun      False
6      and      and  CCONJ      coordinating       True
                                     conjunction
7      Uno      Uno  PROPN      proper noun      False
8       's       's   PART         particle       True
9       as       as    ADV           adverb       True
10    well     well    ADV           adverb       True
11       .        .  PUNCT      punctuation      False
```

SpaCy AND NER

The code sample in this section shows you how to perform NER (Named Entity Recognition) in spaCy. Listing 5.23 shows the contents of `spacy_ner.py` that illustrate how to perform NER in spaCy.

LISTING 5.23: spacy_ner.py

```
import spacy

nlp = spacy.load('en_core_web_sm')

text = "Chicago style deep dish pizza, Uno's, and Nancy's
                                                    pizza."
print("text:")
print(text)
print()

doc = nlp(text)

# Iterate over the entity text and label
for ent in doc.ents:
  print("text/label_:",ent.text, ent.label_)
```

Listing 5.23 is similar to Listing 5.21, except that this code only displays the `text` and `label_` values for each token in the string `text`. Launch the code in Listing 5.21 to see the following output:

```
text:
Chicago style deep dish pizza, Uno's, and Nancy's pizza.
```

```
text/label_: Chicago GPE
text/label_: Uno's ORG
text/label_: Nancy PERSON
```

Notice that `Chicago` is correctly identified as a location (`GPE`) and `Uno's` is correctly identified as an organization. However, `Nancy's` is identified as the person `Nancy` (which is actually correct), even though in this case the string `Nancy's` is the name of a pizza parlor in Chicago.

SpaCy PIPELINES

SpaCy provides a pipeline class that enables you to specify a sequence of tasks to perform. The default pipeline consists of a tagger, a parser and an entity recognizer. The output of each task is a document, which becomes the input for the next task in the pipeline.

Listing 5.24 shows the contents of `spacy_pipeline.py` that illustrate how to use a spaCy pipeline.

LISTING 5.24: spacy_pipeline.py

```
import spacy

sentences = [
  "We got a huge deep dish pizza from Pizzeria Uno.",
  "Supplemented with a couple of pitchers of beer.",
  "Then we went to the top of the Hancock building."
]

nlp = spacy.load("en_core_web_sm")

for doc in nlp.pipe(sentences, disable = ["tagger",
                                          "parser"]):
  print("=> document:",doc)
  for ent in doc.ents:
    print("=> text:",ent.text,"=> label:",ent.label_)
  print()
```

Listing 5.24 initializes the variable `sentences` as an array of sentences, and then initializes the variable `nlp` as an instance of a spaCy model. The next

portion of Listing 5.23 is a `for` loop that is based on `nlp.pipe()`, which in this example only processes the entity recognizer (the tagger and parser recognizers have been excluded).

Launch the code in Listing 5.24 to see the following output:

```
=> document: We got a huge deep dish pizza from
Pizzeria Uno.
=> text: Pizzeria Uno => label: PERSON


=> document: Supplemented with a couple of pitchers of
beer.


=> document: Then we went to the top of the Hancock
building.
=> text: Hancock => label: GPE
```

Pipelines are flexible and convenient for processing text-based documents. More code samples and detailed information regarding spaCy pipelines are available online:

https://spacy.io/usage/processing-pipelines

SpaCy AND WORD VECTORS

Listing 5.25 shows the contents of `spacy_word_vectors.py` that illustrate word vectors for words.

LISTING 5.25: spacy_word_vectors.py

```
import spacy

nlp = spacy.load('en_core_web_lg')

nlp(u'pizza').vector

doc = nlp(u'We ate deep dish pizza with wine and beer.')
print("=> doc.vector:")
print(doc.vector)
print()
```

```
# find similar pairs of tokens:
tokens = nlp(u'wine beer soda')
for token1 in tokens:
  for token2 in tokens:
    print("=> token1.text, token2.text, token1.
                                    similarity(token2):")
    print(token1.text, token2.text, token1.
                                      similarity(token2))
print()

# normalize tokens:
tokens = nlp(u'apple banana orange')
for token in tokens:
  print("=> token.text, token.has_vector, token.vector_
                                    norm, token.is_oov:")
  print(token.text, token.has_vector, token.vector_norm,
                                           token.is_oov)
print()

from scipy import spatial
cosine_similarity = lambda x, y: 1 - spatial.distance.
                                            cosine(x, y)

queen = nlp.vocab['queen'].vector
woman = nlp.vocab['woman'].vector
man = nlp.vocab['man'].vector

# find the closest vector to "man" - "woman" + "queen"
new_vector = queen - woman + man
calc_similarities = []

for word in nlp.vocab:
  # Ignore words without vectors and mixed-case words:
  if word.has_vector:
    if word.is_lower:
      if word.is_alpha:
        similarity = cosine_similarity(new_vector, word.
                                                  vector)
        calc_similarities.append((word, similarity))

calc_similarities = sorted(calc_similarities, key=lambda
                                          item: -item[1])
```

```
print("=> [w[0].text for w in calc_similarities[:10]]:")
print([w[0].text for w in calc_similarities[:10]])
print()

doc = nlp(u'The quick brown fox jumped over the lazy
                                                    dogs.')
print("doc:",doc)
print("doc.vector:",doc.vector)
```

Listing 5.25 initializes the variable `doc` as an instance of the `nlp` class, as well as the given text string, and then displays the contents of the associated word vector. Next, the variable `tokens` is initialized with three words, followed by a nested loop that displays the pair-wise similarity of the words in the `tokens` variable. The next portion of Listing 5.25 initializes the variable tokens with three different words, followed by a loop that displays various attributes these words.

The next portion of Listing 5.25 initializes the variable `cosine_similarity` as a lambda expression that calculates the cosine similarity between a pair of words. Then the variables `queen`, `woman`, and `man` are initialized, followed by the variable `new_vector` that is initialized as follows:

```
new_vector = queen - woman + man
```

Then another loop iterates through the words `nlp.vocab` and for each lowercase alphabetic word in `nlp.vocab`, computes the cosine similarity of that word with the variable `new_vector`. The resultant list of cosine similarities is sorted and then displayed. Launch the code in Listing 5.25 to see the following output:

```
=> doc.vector:
[-1.39887929e-01 -1.95095055e-02  8.34191069e-02 -2.64628291e-01
  2.40830615e-01  2.75315434e-01  2.51161046e-02  7.67326951e-02
  8.21892098e-02  1.90768397e+00 -2.59143114e-01  1.49312809e-01
 -6.83634281e-02 -1.11276604e-01 -1.17691800e-01  -2.02080399e-01
 -7.09474012e-02  1.21521211e+00 -1.57256007e-01 -2.19138991e-02
 // lines omitted for brevity
  3.04495007e-01 -1.26671968e-02  6.86664060e-02 -1.62135810e-01
 -6.57109767e-02  1.62497148e-01  4.26867157e-01  1.21716186e-01
  8.21024999e-02  2.98182011e-01  5.26543036e-02  2.68678162e-02
 -2.37585977e-01  1.15498960e-01 -1.25021964e-01 -1.34173691e-01
  9.52331945e-02 -1.57274202e-01  9.77694020e-02  1.11879949e-02]
```

```
tokens: wine beer soda

=> token1.text, token2.text, token1.similarity(token2):
wine wine 1.0
=> token1.text, token2.text, token1.similarity(token2):
wine beer 0.6600621
=> token1.text, token2.text, token1.similarity(token2):
wine soda 0.4345365
=> token1.text, token2.text, token1.similarity(token2):
beer wine 0.6600621
=> token1.text, token2.text, token1.similarity(token2):
beer beer 1.0
=> token1.text, token2.text, token1.similarity(token2):
beer soda 0.5788868
=> token1.text, token2.text, token1.similarity(token2):
soda wine 0.4345365
=> token1.text, token2.text, token1.similarity(token2):
soda beer 0.5788868
=> token1.text, token2.text, token1.similarity(token2):
soda soda 1.0

=> token.text, token.has_vector, token.vector_norm, token.is_oov:
apple True 7.1346846 False
=> token.text, token.has_vector, token.vector_norm, token.is_oov:
banana True 6.700014 False
=> token.text, token.has_vector, token.vector_norm, token.is_oov:
orange True 6.5420218 False

=> [w[0].text for w in calc_similarities[:10]]:
['queen', 'man', 'he', 'let', 'nothin', 'lovin', 'nuff',
'dare', 'doin', 'all']
```

THE ScispaCy LIBRARY (OPTIONAL)

The scispaCy Python library for NLP is based on spaCy, and it's used for processing biomedical text. This library performs common NLP tasks, such as NER, POS tagging, dependency parsing, and sentence segmentation.

The scispaCy library contains two primary packages that contain models: `en_core_sci_sm` (a smaller vocabulary without word vectors) and `en_core_sci_md` (a larger vocabulary with word vectors).

In particular, NER tasks play a key role in various types of biomedical data in datasets of differing sizes and number of entities (and their types) for different domains, such as cancer genetics, disease-drug interactions, pathway analysis, and trial population extraction. The accuracy of scispaCy NER models is comparable to some other existing models. If this Python library interests you, more information regarding scispaCy is available online:

https://arxiv.org/pdf/1902.07669.pdf

SUMMARY

This chapter showed how to perform data cleaning tasks that involve regular expressions. Then you learned about BoW, along with an explanation of word embeddings. In addition, you saw how to use BeautifulSoup, which is a Python module for scraping HTML Web pages.

Then you learned about Scrapy, which is a Python-based library that provides Web-scraping functionality and other APIs. In addition, you were introduced to spaCy, which is a Python-based library for NLP, along with Python code samples that show you various features of spaCy.

You also learned about word embeddings and the fact that they provide context for words, which can yield more powerful models. Then you saw a Python code sample that involved word2vec, which is a Python-based library (discussed in Chapter 4) for NLP-related tasks that involve unstructured data as well as labeled data.

CHAPTER 6

NLP APPLICATIONS

Chapters 4 and 5 provided a fast-paced introduction to several NLP-related Python libraries and related code samples. By contrast, this chapter is primarily about text classification, recommendation systems, and sentiment analysis.

The first section describes two main types of text summarization (extractive and abstractive), as well as text recommendation. This section also contains Python code samples that illustrate how to use Gensim and spaCy to perform text classification.

The second section contains a brief overview of the recommender systems used in the online reviews of books, movies, and restaurants. The optional portion of this section discusses how to use reinforcement learning in recommender systems.

The third section discusses sentiment analysis, which is actually a subset of text classification. In essence, sentiment analysis attempts to assess the mood (positive, negative, or neutral) of a document (such as a review). We included Python code samples that perform sentiment analysis with Naïve Bayes, NLTK and VADER, and logistic regression.

The final part of this chapter contains a Python code sample involving a COVID-19 dataset, followed by a brief introduction to chatbots.

WHAT IS TEXT SUMMARIZATION?

Text summarization can be informally described as producing a summary of the content of a block of text. Text summarization is similar to a synopsis or an executive summary of a document. In NLP, automatic text summarization involves generating a summary of text from various sources, such as a Webpage, blog post, a document, or a set of documents.

There are two main categories of text summarization (the first is more complex than the second): abstractive summarization techniques and extractive summarization techniques. A high-level description of these summarization techniques is provided in the following subsections.

Extractive Text Summarization

Extractive text summarization algorithms extract keywords or sentences without modifying any of the words in a document and represent the content of a document. Examples of extractive text summarization tasks include producing book reviews, movie reviews, the minutes of a meeting, a document, or a blog post summary.

Note that the extractive algorithms do not generate any new text from a document, and they typically involve smaller amounts of training data. Extractive text summarization has been extensively researched, and has reached a mature stage.

Extractive summarization algorithms perform three independent tasks:

- an intermediate representation of the text
- providing sentence ranks based on the representation
- creating a summary based on some of the sentences

A representation of the input text can use a frequency-based approach, such as calculating word frequencies via tf-idf scores. A topic-based approach involves finding topics in a document and then estimating the importance of each sentence based on the number of topic-related words that appear in a given sentence or the centroids from clusters that are formed by grouping similar data together. A score is assigned to each sentence based on how well it appears to explain the topics in a document. Finally, a summary is generated, based on the highest ranked sentences from the previous step.

Abstractive Text Summarization

Abstractive text summarization tasks are difficult because they involve a complex process that requires understanding the language and the context, after which they generate new sentences. These algorithms often require large amounts of data during the training step. This type of text summarization involves generating new vocabulary that describes the content of a document in a concise and structured manner.

There are two other points that are important. First, text summarization tends to work better with documents that have a smaller number of distinct

topics. For example, lengthy books and poems can contain a wide range of topics, which can pose a challenge for accurate text summarization.

Second, human speech tends to be more casual than written language, which means that errors occur when transcribing speech to text. However, the transcription accuracy continues to improve, which in turn means that it will become more feasible to apply extractive methods to text that has been transcribed from speech.

TEXT SUMMARIZATION WITH GENSIM AND SpaCy

This section contains Python code samples that combine Gensim and spaCy to perform text summarization.

Listing 6.1 shows the contents of `text_summarization.py` that illustrate how to perform text summarization with `gensim` on a block of text.

LISTING 6.1: text_summarization.py

```
from gensim.summarization import summarize

mytext = """
Chapters five and six provide a fast-paced introduction
to several NLP-related Python libraries and related code
samples. By contrast, this chapter is primarily about
text classification, recommendation systems, and sentiment
analysis.
The first section discusses two main types of text
summarization (extractive and abstractive), as well as
text recommendation. This section also contains Python
code samples that illustrate how to use gensim and SpaCy
to perform text classification.
The second section contains a brief overview of
recommender systems, which are used in online reviews of
books, movies, restaurants, and so forth. You will also
learn how to use the Python surprise library that provides
a layer of abstraction above the tasks that are required
for recommender systems. The final (optional) portion of
this section discusses how to use reinforcement learning
in recommender systems.
"""
```

```
# Summarize the preceding text by passing it as
# an input to "summarize" that returns a summary
print("===> text summary:")
print(summarize(mytext))
print()

# the "split" option produces a list of strings
print("===> split the text summary:")
print(summarize(mytext, split = True))
print()

# 1) the "ratio" parameter changes the displayed text
                                              (default = 20%)
# 2) the "word_count" parameter: the number of words to
                                                      display
print("===> a 50-word summary:")
print(summarize(mytext, word_count = 50))
print()

from gensim.summarization import keywords
print("===> keywords:")
print(keywords(mytext))
print()
```

Listing 6.1 initializes the variable `mytext` with a test string that is passed in to the `summary()` method provided by `gensim`, after which a summary of the contents of `mytext` is displayed.

The next portion of Listing 6.1 invokes the `summary()` method with `split=True` to display the preceding output as a list of strings. The next code snippet invokes the `summary()` method again, this time with `word_count=50` to display a 50-word summary of the input text. Launch the code in Listing 6.1 to see the following output:

```
===> text summary:
You will also learn how to use the Python surprise
library that provides a layer of abstraction above the
tasks that are required for recommender systems.


===> split the text summary:
['You will also learn how to use the Python surprise
library that provides a layer of abstraction above the
tasks that are required for recommender systems.']
```

```
===> a 50-word summary:
This section also contains Python code samples that
illustrate how to use gensim and SpaCy to perform text
classification.
You will also learn how to use the Python surprise
library that provides a layer of abstraction above the
tasks that are required for recommender systems.

===> keywords:
python
text
section
movies
optional
recommendation
recommender
```

Listing 6.2 shows the content of `gensim_spacy.py` that illustrates how to perform text summarization with `gensim` and `spaCy` on text that is extracted from Wikipedia.

NOTE *Make sure that you invoke* `pip3 install Wikipedia`.

LISTING 6.2: gensim_spacy.py

```
import spacy

from gensim.summarization.summarizer import summarize
from gensim.summarization import keywords
import wikipedia

# Get wiki content for Japan:
wikisearch = wikipedia.page("Japan")
wikicontent = wikisearch.content

nlp = spacy.load('en_core_web_sm')
doc = nlp(wikicontent)

# Save the wiki content to a file:
f = open("wikicontent.txt", "w")
f.write(wikicontent)
f.close()
```

```
# Summary (0.5% of the original content):
summ_per = summarize(wikicontent, ratio = 0.05)
print("=> Percent summary:")
print(summ_per)
```

Listing 6.2 initializes the variable `wikisearch` with the result of invoking the `wikipedia.page()` method, followed by the variable `wikicontent` that contains the text from the variable `wikisearch`.

Next, the variable `nlp` is initialized as an instance of the small Web model from `spaCy`, and the variable `doc` is initialized with the result of passing `wikicontent` to the `nlp` variable. The `wikicontent` variable contains actual text, which is saved to a text file.

The final code block in Listing 6.2 provides a small (i.e., 0.5%) summary of the text in `wikicontent`. Launch the code in Listing 6.2 to see the following output:

```
=> Percent summary:
Japan (Japanese: 日本, Nippon [ɲippoˈɴ] (listen) or
Nihon [ɲihoˈɴ] (listen)) is an island country in East
Asia, located in the northwest Pacific Ocean. It is
bordered on the west by the Sea of Japan, and extends
from the Sea of Okhotsk in the north toward the East
China Sea and Taiwan in the south. Part of the Ring
of Fire, Japan spans an archipelago of 6852 islands
covering 377,975 square kilometers (145,937 sq mi);
the five main islands are Hokkaido, Honshu, Shikoku,
Kyushu, and Okinawa. Tokyo is Japan's capital and
largest city; other major cities include Yokohama,
Osaka, Nagoya, Sapporo, Fukuoka, Kobe, and Kyoto.


Japan is the eleventh-most populous country in the
world, as well as one of the most densely populated and
urbanized. About three-fourths of the country's terrain
is mountainous, concentrating its population of 125.71
million on narrow coastal plains. Japan is divided into
47 administrative prefectures and eight traditional
regions. The Greater Tokyo Area is the most populous
metropolitan area in the world, with more than 37.4
million residents.


[some content omitted for brevity]
```

```
=> Word count summary:
Although it has renounced its right to declare war, the
country maintains Self-Defense Forces that are ranked
as the world's fourth-most powerful military.
Ranked the second-highest country on the Human
Development Index in Asia after Singapore, Japan has
the world's second-highest life expectancy, though it
is experiencing a decline in population.
Despite early resistance, Buddhism was promoted by the
ruling class, including figures like Prince Shōtoku,
and gained widespread acceptance beginning in the Asuka
period (592–710).The far-reaching Taika Reforms in
645 nationalized all land in Japan, to be distributed
equally among cultivators, and ordered the compilation
of a household registry as the basis for a new system
of taxation.
During the Meiji era (1868–1912), the Empire of Japan
emerged as the most developed nation in Asia and as
an industrialized world power that pursued military
conflict to expand its sphere of influence.
```

[Some content was omitted for brevity.]

WHAT ARE RECOMMENDER SYSTEMS?

Recommender systems are a subset of information filtering systems that attempt to predict the rating or preference assigned by users to an item. Such systems personalize the information supplied to users based on their interests and the relevance of the information. Recommendation systems are used widely for many scenarios, such as the following:

- recommending movies
- articles
- restaurants
- places to visit
- items to buy

There are three major types of algorithms that are used for recommendation systems, as shown in the following list:

- collaborative filtering (similar users)
- content-based approaches (item features)
- a hybrid of the first two

In highly simplified terms, *collaborative filtering* makes recommendations to a user based on another user who has similar preferences. This technique obviously requires some users, and the initial absence of users is called the "cold start" problem. By contrast, a content-based approach recommends a new item to a user based on the similarity of the features of that item to an existing (and similar) item. A hybrid approach can start with a content-based approach (which does not suffer from the cold start problem), and then use a collaborative filtering approach.

These three approaches are discussed in greater detail in a subsequent section, after we explore some of the aspects of a movie recommender system, which is the topic of the next section.

Movie Recommender Systems

A *movie recommender system* is familiar to most people, and in simple terms, the goal of such a system is to create an accurate list of recommendations to its users. A recommendation list takes into account the movies that a user has seen, the ratings assigned to the movies by each user, and a mix of other factors.

Suppose that we have a matrix `R` with m rows (users) and n columns (movies), and each entry in matrix `R` is a movie rating, which can be a number between 1 and 5 inclusive or a floating point number between 0 and 1. Matrix `R` can have millions of rows and tens of thousands of movies. In addition, matrix `R` probably has some relationships that can provide useful information. For example, it's possible to have the following:

- two equal rows (two users have the same movie ratings)
- two equal columns (two movies have the same ratings by multiple users)
- a third row is the sum of two other rows

Here is an example of a matrix `R` that consists of four users and four movies, with movie ratings expressed as a floating point number between 0 and 1:

```
        M1   M2    M2    M4
Alice   1    -     0.2   -
Edward  -    0.5   -     0.3
Steve   1    -     1     -
David   1    -     -     0.4
```

There are some missing entries in the preceding matrix. The goal of a recommender system is to infer values for the missing entries.

Now consider the following rating matrix R in which all numeric ratings are equal to 2, along with one missing value:

```
         M1   M2   M2   M4
Alice    2    2    2    2
Edward   2    2    2    2
Steve    2    2    2    ?
David    2    2    2    2
```

The obvious choice for the missing value in the preceding matrix R is 2. Now try to infer the missing value in the following matrix R:

```
         M1   M2   M2   M4
Alice    3    1    1    2
Edward   1    2    4    3
Steve    3    1    1    ?
David    4    3    5    4
```

What is the missing value in the preceding matrix? If you guessed 2, you are correct. This value is inferred from the fact that Alice and Steve have three identical ratings, and since Alice rated M4 with the value 2, we can infer that Steve would also rate this movie with a value of 2.

Factoring the Rating Matrix R

As you learned in a previous section, the rating matrix R can be massive, which will degrade performance when processing this matrix for information. Fortunately, we can avail ourselves of a technique called *matrix factorization*:

https://en.wikipedia.org/wiki/Matrix_factorization_(recommender_systems)

Let's suppose that each row of R is a movie and there are 1,000 movies. In addition, suppose that each column of R is a user, and there are 2,000 users. Then the array R has 2,000,000 entries (and is also sparse).

However, we can decompose the matrix R into the product of two matrices M and U. Specifically, M is a 1000 × 100 matrix of consisting of movies and features for its rows and columns, respectively. In addition, matrix U is a 100 × 2000 matrix consisting of features and users for its rows and columns, respectively. The matrices M and U are compatible, because their product

is (1000 × 100) × (100 × 2000), which is a matrix of dimensionality 1000 × 2000: this is also the dimensionality of matrix `R`. Moreover, the content of the rows and columns of `M*U` matches the content of the rows and columns of matrix R.

The following section describes a new type of recommendation system, followed by the two main types of recommendation systems that were mentioned earlier in this chapter.

CONTENT-BASED RECOMMENDATION SYSTEMS

The premise of *content-based recommendation systems* is that if you like an item, you will probably like a similar item. In situations where it's easy to determine the context or properties of each item, a recommendation that's based on the similarity of the items generally works well (for example, recommending the same kind of item, such as a movie, book, song, or restaurant).

Content-based recommendation systems are classified in two broad categories: (1) a technique that analyzes only the description of the content and (2) a technique that involves building user profiles and item profiles. Both techniques are discussed in the following subsections.

Analyzing Only the Description of the Content

This technique is similar to item-based collaborative filtering. This means that the system recommends anything similar to previously items that are marked as "liked." The advantages of this technique are as follows:

- avoids the "new item problem" if the descriptions are good
- semantic information and inferences can be used
- easier to create more transparent systems

This technique also has the following disadvantages:

- content-based recommendation systems tend to overspecialize
- they will recommend items similar to those already consumed
- the preceding also has a tendency of creating a "filter bubble"

Creating User Profiles and Item Profiles

One technique for creating user profiles utilizes a description or attributes from items the user has previously interacted with to recommend similar items. This technique depends only on the user's previous choices. This

technique is robust and avoids the "cold-start problem." It's also simple to use for textual items (such as articles, news and books).

COLLABORATIVE FILTERING ALGORITHM

The premise of collaborative algorithm can be illustrated with the following simple example. Suppose that person A likes items 1, 2, and 3, whereas person B likes items 2, 3, and 4. As you can see, A and B have similar interests, and so we infer that A would like item 4 (because B likes it) and that B would like item A (because A likes it).

An example of collaborative filtering occurs when you go to a restaurant and then you ask the wait staff (or someone in your group) for a recommendation.

The collaborative filtering algorithm is

- entirely based on past behavior
- not based on the context
- is independent of any other information
- a commonly used algorithm

User–User Collaborative Filtering

User–user collaborative filtering involves searching for "look-alike" customers (based on a similarity) To offer products to the second customer based on what the first customer has chosen in the past. This is an effective algorithm that requires a significant amount of time and resources because information about *every* customer pair must be determined. In the case of very large platforms, this algorithm is difficult to implement without a very strong parallelizable system.

Item–Item Collaborative Filtering

Item–item collaborative filtering is similar to user–user collaborative filtering, with the following points:

- finds the item look-alike instead of the customer look-alike
- easy to recommend similar items to customers who have purchased items (when the item look-alike matrix is constructed)

This technique is a far less resource consuming than user–user collaborative filtering. In fact, for new customers, the item-item algorithm takes far less time than user-user collaborate as we don't need all the similarity scores

between customers. In the case of a fixed number of products, the product–product look-alike matrix is fixed over time. If need be, you can read the section in appendix A that discusses various types of distance metrics.

Recommender System with Surprise

Surprise is a simple Python recommendation system engine that is based on Sklearn. Surprise is an open source Python library for building and analyzing recommender systems:

https://surpriselib.com

Surprise provides built-in datasets, prediction algorithms, dimensionality reduction (PCA and SVD), and various metrics such as MAE, RMSE, and so forth. Navigate to its home page to learn about its features and code samples:

https://surpriselib.com

RECOMMENDER SYSTEMS AND REINFORCEMENT LEARNING (OPTIONAL)

As you have already learned, traditional recommender systems typically involve either collaborative filtering or content-based systems. Another approach to implementing recommender systems involves reinforcement learning. Before we proceed with more details, let's take a short digression to provide a high-level description of reinforcement learning.

Basic Reinforcement Learning in Five Minutes

Reinforcement learning involves an agent that seeks to maximize its expected future reward, which involves moving among various states, in a randomly selected fashion or in a deterministic fashion. Each state has a reward that can be a positive value, zero, or negative value, and finding the optimal path often involves many, many iterations. *The agent (aka the learner) is not told which actions must be taken and in which sequence they must be taken. The agent must discover those actions by itself.*

In general, greedy algorithms fail in reinforcement learning tasks, so a generalization called the "epsilon greedy" algorithm is employed. This involves the following variables:

- an initial state S
- a variable `epsilon` with initial value equal to 1
- a variable `rnd` that is a randomly generated number between 0 and 1
- a Q table whose rows are states and columns are actions

If `rnd` is less than `epsilon`, then a random action A (that is defined for the current state) is selected; otherwise, an action is selected for which `Q(S,A)` has the maximum value. In either case, the selected action A is handed to the environment, which you can think of as an "oracle" that returns a new state, a reward, and a Boolean "done" flag that equals true when the current episode (or game) has concluded. Here is a sample of the type of code that determines the new state based on the current action, where the variable env is the "oracle:"

```
import gym
# initialize state, action, epsilon, rnd, done, total_reward
# state = current state
# other details omitted

while(done = False):
  # find an action based on epsilon greedy algorithm:
  # details omitted...

  # get the next state and reward:
  next_state, reward, done, _ = env.step(action)

  # set the current state to the new state:
  state = next_state
  total_reward += reward
```

Note that the preceding code is inside a loop, which means that many actions are selected in order to arrive at different states.

Each time that a value is generated for `rnd`, the value of `epsilon` is decremented from 1 to a small number (typically 0.1). Hence, over the course of multiple iterations, the number of randomly selected actions (called "exploration") decreases and the number of greedy-style selections (called "exploitation") increases.

If `epsilon` is the constant 0, then the algorithm is simply a greedy algorithm, so the latter is actually a special case of the epsilon greedy algorithm.

However, if `epsilon` is the constant 1, then the algorithm involves only randomly selected choices, which does not provide any meaningful value. The epsilon greedy algorithm is both clever and elegant in its ability to combine random choices with deterministic choices.

One well-known technique in reinforcement learning is called *q-learning* ("quality" learning) which involves a two-dimensional matrix (aka a "q-table") with states as rows and actions as columns. The matrix cells are (state, action) pairs that are initialized to 0. After each iteration of the loop containing the epsilon greedy algorithm, the currently selected (state, action) pair is updated with a new reward value, and the agent makes a transition to the newly selected state.

However, a q-table works when there is a *fixed* number of states and actions, such as tasks involving a maze or navigating around a rectangular grid. In the case of games such as Super Mario, each time Mario moves on the screen, the new set of pixels is treated as a new state. Thus, the number of states is treated as though it's continuous rather than discrete.

However, instead of appending each new state to a q-table, we use something called deep Q-learning (DQN), which involves passing the state as the input to a neural network and the output layer is a set of possible actions. The neural network is trained via backward error propagation, and the action in the output layer that has the highest probability is selected.

One other scenario can arise when the number of states and the number of actions are both continuous. As an example, consider a moving vehicle. The number of states is continuous and the number of actions (such as turning the steering wheel) is also continuous. There are some deep reinforcement learning algorithms, such as soft actor critic (SAC) and twin delayed DDPG (TD3) that solve this type of task.

Almost every reinforcement learning task can be modeled as an Markov decision process (MDP), which is based on a Markov chain, and the latter is a nondeterministic finite automata (NFA) with probabilities for outgoing edges (whose sum equals one). There are many concepts and algorithms in reinforcement learning, including

- agent
- environment
- state
- state transitions
- actions (and probabilities)
- discount factor

- discounted future reward
- Markov chains
- MDPs
- on-policy versus off-policy
- model-based versus model-free
- Q-learning
- Bellman's equation
- policy gradient algorithms
- deep reinforcement learning algorithms

There are numerous online articles that discuss the topics in the preceding list, along with the code samples for deep reinforcement learning algorithms.

What Is RecSim?

RecSim is an open source platform that is based on reinforcement learning (RL) that creates simulations for collaborative interactive recommenders (CIRs). Recall from the previous section that RL involves an agent, a set of states, and a set of actions associated with each state that enables transitions between states. Moreover, some tasks use the epsilon greedy algorithm to select a state that is passed to the environment ("oracle") that returns a new state, a reward, and a Boolean "done" flag.

By contrast, a RecSim agent interacts with an environment that consists of a user model, a document model, and a user choice model. In RecSim, we can represent the state by the content, the action is the next best content, and the user satisfaction represents the reward. Moreover, we can use a vector embedding to represent the content. Without delving into many details, RL enables recommender systems to suggest new recommendations to users that are independent of earlier recommendations.

In fact, those new recommendations can contain random content that might be appealing to them. This approach provides users with opportunities to discover new interests that previously might not have interested them. After all, people's interests can (and do) change over time. Indeed, RL-based models continually learn and evolve as users' interests change.

This concludes the portion of the chapter pertaining to recommender systems. If you are interested in learning about recent trends in recommender systems, navigate to the following link:

https://aws.amazon.com/blogs/media/whats-new-in-recommender-systems/

WHAT IS SENTIMENT ANALYSIS?

The purpose of sentiment analysis is to determine the attitude of a person regarding a topic, the context of a document, or of a corpus of documents. In each case, sentiment analysis assesses whether the input text expresses a positive, negative, or neutral sentiment.

In high-level terms, sentiment analysis involves various steps for processing natural language, followed by training a model. The first stage processes text in a way that, when we are ready to train our model, we already know what variables the model needs to consider as inputs. The model learns how to determine the sentiment of a piece of text based on these variables.

Sentiment analysis can also be a binary classification task involving positive and negative sentiment. Sentiment analysis can be performed in many situations, some of which are listed here:

- the online sentiment for a particular product
- analyzing movie, book, and restaurant reviews
- issues logged at customer support centers

As you learned in Chapter 3, human languages are very flexible in that they enable people to express

- a mix of positive and negative sentiments
- the use of sarcasm and its nuances
- the use of slang (a negative can mean a positive)

Moreover, language is fundamentally ambiguous because our emotions can convey sarcasm, irony, and plays on words, all of which pose challenges for NLP.

There are two main ways to perform sentiment analysis: a rule-based approach (older and less powerful) and machine learning techniques.

The rule-based technique counts the number of positive words and negative words in a document and whichever of these two numbers is larger determines the sentiment of the document.

However, the rule-based approach for sentiment analysis has a drawback. This method focuses on individual words and does not examine any context. More than likely, token-based algorithms will generate a highly negative ranking for sentences with slang such as "The concert last night was the bomb," which actually means the concert was great ("the bomb"), not terrible ("a bomb").

The machine learning approach involves a classification model (chosen from various algorithms) that is trained with a labeled dataset of positive, negative, and neutral sentiments. Assign the values 1, –1, and 0 to these three sentiments, or assign a range of values that are in the interval [–1,1]. In the latter case, –1 is the most negative sentiment value and +1 is the most positive sentiment value.

As an exercise, see if you can assign a numeric value to the sentiment of each of the following sentences:

1. "Our plan was not without merit" is similar in meaning to "Our plan has merit."
2. "I like the pizza toppings, but I do not like the crust."
3. "The only thing worse than being talked about, is not being talked about" (Oscar Wilde).
4. "Everything is funny, as long as it's happening to somebody else" (Will Rogers).
5. "When ignorance is bliss, 'tis folly to be wise" (William Shakespeare).
6. "Good judgment is the result of experience and experience the result of bad judgement" (Mark Twain).

Here are some observations about the sentences in the preceding list. Example #1 contains two sentences that have approximately the same interpretation. However, the former sentence is close to a "double negative," which is a more difficult NLP sentiment analysis task. Depending on the Python library that you use, the second sentence is more likely to receive a higher positive sentiment than the first sentence.

In example #2, there is a positive and a negative sentiment. What numeric sentiment value would you assign to the entire sentence? If you compute the average of that +1 (for positive) and –1 (for negative) the result is 0, which suggests a neutral sentiment. Although the overall sentiment could be deemed neutral, the two parts of the sentence are not neutral.

Example #3 might be viewed as a pithy and ironic observation about people, and some might say that to varying degrees it's also somewhat rueful. This sentence expresses a sentiment of the form "B is worse than A," which implies that A is bad without explicitly expressing such an opinion.

Example #4 is an observation that tends to be true, but it encompasses positive as well as negative events without the use of any negative words such as "not," "bad," "worse," and so forth.

Example #5 is a famous quote from William Shakespeare that is often misquoted as "ignorance is bliss," and the full quote obviously has a much different meaning. The words "ignorance" and "folly" are two words that express a negative sentiment.

Example #6 sounds paradoxical until you've gained enough experience and wisdom to understand its meaning. The words "good" and "bad" are the only two words that express any sentiment in this sentence. Later in this chapter, you will see a Python code sample that performs sentiment analysis on the preceding list of sentences.

Useful Tools for Sentiment Analysis

The following list of sentiment analysis tools provide various features that might be well-suited for your NLP needs:

- IBM Watson Tone Analyzer
- OpenText
- Talkwalker
- Rapidminer
- Social Mention
- Textblob
- Vader

Perform an online search for documentation for these sentiment analysis tools to determine which ones are suitable for your tasks.

Aspect-Based Sentiment Analysis

Aspect-based sentiment analysis is a more advanced technique than sentiment analysis. The latter only detects the sentiment of an overall corpus, whereas the former analyzes each text to identify various aspects and the corresponding sentiment for each text.

For example, sentiment analysis might determine that a comment is negative, whereas aspect-based sentiment analysis might determine that a customer is unhappy with the battery life of a specific product. Hence, aspect-based sentiment analysis produces finer-grained analysis of a corpus, which is obviously important when handling text-based customer feedback regarding products and services.

Note that aspect-based sentiment analysis can extract sentiments (i.e., positive or negative opinions about a particular aspect) as well as aspects (the item that is the current focus).

Several libraries provide the algorithmic building blocks of NLP in real-world applications. For instance, Algorithmia provides a free API endpoint for many of these algorithms, without ever having to set up or provision servers and infrastructure.

Another machine learning toolkit is Apache OpenNLP, which provides tokenizers, sentence segmentation, part-of-speech tagging, named entity extraction, chunking, parsing, and coreference resolution.

Deep Learning and Sentiment Analysis

Deep learning models, such as RNNs and LSTMs, can be combined with NLP-based sentiment analysis, such as gauging sentiment in tweets:

https://ieeexplore.ieee.org/stamp/stamp.jsp? arnumber=8244338

You can also combine RNNs with sentiment analysis:

https://blog.openai.com/unsupervised-sentiment-neuron/
https://github.com/openai/generating-reviews- discovering-sentiment

There are several details to keep in mind. First, distributed word vector techniques have been shown to outperform BoW models (as you would probably surmise). Second, the paragraph vector algorithm preserves word order information and produces state-of-the-art results. This algorithm performs better because vector averaging and clustering lose the word order. Third, it's worthwhile to acquaint yourself with the transformer architecture and BERT-based models that are discussed in Chapter 7 before you decide to use RNNs or LSTMs for sentiment analysis.

SENTIMENT ANALYSIS WITH NAÏVE BAYES

Listing 6.3 shows the contents of `nb_sentiment.py` that illustrate how to perform sentiment analysis with NLTK.

LISTING 6.3: nb_sentiment.py

```
import pandas as pd
import matplotlib.pyplot as plt
import nltk
from nltk.tokenize import RegexpTokenizer

from sklearn import metrics
```

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import MultinomialNB

data = pd.read_csv('train.tsv', sep = '\t')
print(data.head())

print("=> Data information:")
data.info()
print("-------------------------------")
print()

print("=> Sentiment value counts:")
print(data.Sentiment.value_counts())
print("-------------------------------")
print()

#find alphanumeric patterns:
token = RegexpTokenizer(r'[a-zA-Z0-9]+')

cv = CountVectorizer(lowercase = True,stop_words = 'english',
           ngram_range = (1,1),tokenizer = token.tokenize)

phrase_counts = cv.fit_transform(data['Phrase'])

X_train, X_test, y_train, y_test = train_test_split(
  phrase_counts, data['Sentiment'], test_size = 0.3,
                                          random_state = 1)

# Multinomial Naive Bayes model:
clf = MultinomialNB().fit(X_train, y_train)
predicted = clf.predict(X_test)
print("=> MultinomialNB Accuracy:",metrics.accuracy_
                                  score(y_test,predicted))
print("-------------------------------")
print()

# second part: use tf-idf values
tf = TfidfVectorizer()
text_tf = tf.fit_transform(data['Phrase'])
print("text_tf:")
print(text_tf)
```

```
print("------------------------------")
print()

X_train, X_test, y_train, y_test = train_test_split(
  text_tf, data['Sentiment'], test_size = 0.3, random_
                                                state = 123)

# Multinomial Naive Bayes model:
clf = MultinomialNB().fit(X_train, y_train)
predicted = clf.predict(X_test)
print("=> MultinomialNB Accuracy:",metrics.accuracy_
                                    score(y_test,predicted))
print("------------------------------")
```

Listing 6.3 contains two sections of code. The first section reads the contents of the TSV file `train.tsv` into the data frame `data`. It then uses a combination of the classes `RegexpTokenizer` (which performs tokenization based on a regular expression) and `CountVectorizer` (discussed in Chapter 4) to initialize the variables `token` and `cv`, respectively, to find alphabetic strings and determine the frequency of those strings. In addition, the training and test datasets are created from the `Sentiment` column of the data frame `data`, as shown here:

```
X_train, X_test, y_train, y_test = train_test_split(
  phrase_counts, data['Sentiment'], test_size = 0.3,
                                    random_state = 1)
```

Next, the variable `clf` is instantiated as an instance of the `NaiveBayes` classification algorithm. Notice that this code is similar to Listing 8.4 in Chapter 8 after replacing the instance of the `DecisionTreeClassifier` class with the following code snippet:

```
clf = MultinomialNB().fit(X_train, y_train)
```

The next portion of Listing 6.3 invokes the `predict()` method to make the predictions on the test-related data.

The next section of Listing 6.3 instantiates the variable `tf` as an instance of the `TfidfVectorizer` class, followed by the variable `text_tf` that is the result of transforming and fitting the data in the `Phrase` column. Once again, the training and test datasets are created from the `Sentiment` column of the data frame `data`, as shown here:

```
X_train, X_test, y_train, y_test = train_test_split(
  phrase_counts, data['Sentiment'], test_size = 0.3,
                                        random_state = 1)
```

The remaining code is a duplicate of the corresponding code in the first section of this code sample. Launch the code in Listing 6.4 to see the following output so that you can compare the accuracy of the two sections in Listing 6.3:

```
=> First five rows:
PhraseId  ...  Sentiment
0         1  ...          1
1         2  ...          2
2         3  ...          2
3         4  ...          2
4         5  ...          2

[5 rows x 4 columns]

=> Data information:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 156060 entries, 0 to 156059
Data columns (total 4 columns):
PhraseId      156060 non-null int64
SentenceId    156060 non-null int64
Phrase        156060 non-null object
Sentiment     156060 non-null int64
dtypes: int64(3), object(1)
memory usage: 4.8+ MB
------------------------------

=> Sentiment value counts:
2    79582
3    32927
1    27273
4     9206
0     7072
```

```
Name: Sentiment, dtype: int64
------------------------------

=> MultinomialNB Accuracy: 0.6049169122986885
------------------------------
text_tf:
  (0, 12857)    0.12785637560254456
  (0, 8807)     0.1353879543646446
  (0, 13681)    0.07615285026452821
  (0, 593)      0.22068902883834374
  (0, 9085)     0.1898515417082945
  (0, 1879)     0.11034437734762885
  (0, 602)      0.26341877863818697
  (0, 9204)     0.19301332592202286
  (0, 14888)    0.28701927784529135
  (0, 12424)    0.1381592967010513
  (0, 5595)     0.265796263188737
  (0, 529)      0.1614381914318891
  (0, 5837)     0.22883807138484064
  (0, 5323)     0.20344769269023563
  (0, 5821)     0.2625302862532789
  (0, 7217)     0.17522921677393963
  (0, 14871)    0.1354415412970302
  (0, 13503)    0.08982508036989033
  (0, 288)      0.251134096800077
  (0, 13505)    0.17690005957760713
  (0, 3490)     0.2485059095620638
  (0, 4577)     0.278538658922562
  (0, 9227)     0.27061683772839323
  (0, 11837)    0.1761994204821687
  (1, 5837)     0.3782714454401254
  : :
  (156050, 11465)     0.670263619653983
  (156050, 625)       0.2115725833396903
```

```
  (156050, 13505)     0.18632379802617538
  (156051, 9193)      0.6987248068627274
  (156051, 11465)     0.6822102168950972
  (156051, 625)       0.21534359576868978
  (156052, 11465)     0.953619269081851
  (156052, 625)       0.3010154308931625
  (156053, 2313)      0.4917001772764322
  (156053, 1027)      0.4917001772764322
  (156053, 6245)      0.45540097827929693
  (156053, 5328)      0.3853824417825967
  (156053, 1313)      0.40068964783307426
  (156054, 2313)      0.5366653003868254
  (156054, 1027)      0.5366653003868254
  (156054, 6245)      0.4970466029897592
  (156054, 5328)      0.4206249935248471
  (156055, 6245)      1.0
  (156056, 2313)      0.618474762808639
  (156056, 1027)      0.618474762808639
  (156056, 5328)      0.4847452274521073
  (156057, 2313)      0.7071067811865476
  (156057, 1027)      0.7071067811865476
  (156058, 1027)      1.0
  (156059, 2313)      1.0
------------------------------

=> MultinomialNB Accuracy: 0.5865265496176684
------------------------------
```

SENTIMENT ANALYSIS WITH VADER AND NLTK

This section contains two Python code samples that perform sentiment analysis. Listing 6.4 shows the contents of `vader_sentiment.py` that illustrate how to perform sentiment analysis with Vader.

NOTE *Make sure that you invoke the following two commands:*

```
pip3 install vader
pip3 install vaderSentiment
```

LISTING 6.4: vader_sentiment.py

```
# pip3 install vader
# pip3 install vaderSentiment

from vaderSentiment.vaderSentiment import
SentimentIntensityAnalyzer

sia = SentimentIntensityAnalyzer()

sent = "I love Chicago deep dish pizza."
print("=> Sentence:",sent)
word_probs = sia.polarity_scores(sent)
print("=> Sentiment:",str(word_probs))
print()

sent = "I love Chicago deep dish pizza!"
print("=> Sentence:",sent)
word_probs = sia.polarity_scores(sent)
print("=> Sentiment:",str(word_probs))
print()

sent = "I love Chicago deep dish pizza!!!"
print("=> Sentence:",sent)
word_probs = sia.polarity_scores(sent)
print("=> Sentiment:",str(word_probs))
print()
```

Listing 6.4 starts with an `import` statement and then initializes the variable `sia` as an instance of the class `SentimentIntensityAnalyzer` that is available from `vaderSentiment`.

The next three code blocks initialize the variable `sent` as a text string that contains zero, one, and three exclamation points. In each case, the variable `sent` is supplied to the method `polarity_scores()` to illustrate the effect of the exclamation points on the generated polarity value. Launch the code in Listing 6.4 to see the following output:

```
=> Sentence: I love Chicago deep dish pizza.
=> Sentiment: {'neg': 0.0, 'neu': 0.543, 'pos': 0.457,
                                        'compound': 0.6369}
```

```
=> Sentence: I love Chicago deep dish pizza!
=> Sentiment: {'neg': 0.0, 'neu': 0.527, 'pos': 0.473,
'compound': 0.6696}

=> Sentence: I love Chicago deep dish pizza!!!
=> Sentiment: {'neg': 0.0, 'neu': 0.496, 'pos': 0.504,
'compound': 0.7249}
```

Notice how the positive sentiment in the preceding output increases when the number of exclamation points is increased; this makes sense because more exclamation points tends to make a statement or question more emphatic (be it positive or negative).

By way of comparison, Listing 6.5 shows the contents of the Python script `vader_nltk_sentiment.py` that uses the `SentimentIntensityAnalyzer` class from NLTK instead of `vaderSentiment` to perform sentiment analysis with NLTK.

NOTE *The following code sample downloads a 266 MB file (if it is not already available)* `sentiment-en-mix-distillbert_3.1.pt` *when you execute the code.*

LISTING 6.5: vader_nltk_sentiment.py

```
import nltk
#nltk.download('vader_lexicon')

from nltk.sentiment.vader import
SentimentIntensityAnalyzer

sia = SentimentIntensityAnalyzer()
sent = "I love Chicago deep dish pizza."
print("=> sentence:",sent)
print(sia.polarity_scores(sent))
print()

sent = "I love Chicago deep dish pizza!"
print("=> sentence:",sent)
print(sia.polarity_scores(sent))
print()

sent = "I love Chicago deep dish pizza!!!"
print("=> sentence:",sent)
print(sia.polarity_scores(sent))
print()
```

Launch the code in Listing 6.5 to see the following output that you can compare with the output from launching Listing 6.5 in the previous section.

```
=> sentence: I love Chicago deep dish pizza.
{'neg': 0.0, 'neu': 0.488, 'pos': 0.512, 'compound':
0.6369}

=> sentence: I love Chicago deep dish pizza!
{'neg': 0.0, 'neu': 0.471, 'pos': 0.529, 'compound':
0.6696}

=> sentence: I love Chicago deep dish pizza!!!
{'neg': 0.0, 'neu': 0.441, 'pos': 0.559, 'compound':
0.7249}
```

SENTIMENT ANALYSIS WITH TEXTBLOB

TextBlob is an open source Python-based library that performs various NLP-tasks, including sentiment analysis. Specifically, TextBlob provides a rule-based sentiment analyzer takes a text string as input and then returns two properties, both of which are floating point numbers.

1. *Polarity* is a number in the interval [-1,1], where -1 and +1 indicate negative and positive sentiment, respectively.
2. *Subjectivity* is number in the interval [0,1] that indicates the degree to which a sentence involves personal emotion, judgement, or opinion.

Listing 6.6 shows the contents of the Python script `textblob_sentiment.py` that use the `Textblob` package to perform sentiment analysis.

LISTING 6.6: textblob_sentiment.py

```
# pip3 install textblob
from textblob import TextBlob
sent = "I love Chicago deep dish pizza."
tb_sent = TextBlob(sent)
print("sentence:",tb_sent)
print(tb_sent.sentiment)
print()
```

```
sent = "I love Chicago deep dish pizza!"
tb_sent = TextBlob(sent)
print("sentence:",tb_sent)
print(tb_sent.sentiment)
print()

sent = "I love Chicago deep dish pizza!!!"
tb_sent = TextBlob(sent)
print("sentence:",tb_sent)
print(tb_sent.sentiment)
print()
```

Listing 6.6 contains three code blocks, analogous to the contents of Listing 6.5, but with the variable `tb_sent` that is an instance of `TextBlob`. Now launch the code in Listing 6.6 and compare this output with the output from Listing 6.5 and Listing 6.4.

```
sentence: I love Chicago deep dish pizza.
Sentiment(polarity = 0.25, subjectivity = 0.5)

sentence: I love Chicago deep dish pizza!
Sentiment(polarity = 0.25, subjectivity = 0.5)

sentence: I love Chicago deep dish pizza!!!
Sentiment(polarity = 0.25, subjectivity = 0.5)
```

Notice how the sentiment analysis in the preceding output is unaffected by the number of exclamation points in the input text.

Listing 6.7 shows the contents of `nltk_sentiment.py` that illustrate yet another example of performing sentiment analysis with NLTK.

LISTING 6.7: nltk_sentiment.py

```
import nltk
#nltk.download('vader_lexicon')
from nltk.sentiment.vader import
SentimentIntensityAnalyzer
import pandas as pd

sentiment = SentimentIntensityAnalyzer()
sentence = "I love Chicago deep dish pizza."
print("=> sentence:",sentence)
```

```
print("=> polarity:",sentiment.polarity_scores(sentence))
print()

sentences = [
  "Our plan was not without merit",
  "I like the pizza toppings but I do not like the
crust.",
  "The only thing worse than being talked about, is not
being talked about",
  "Everything is funny, as long as it's happening to
somebody else.",
  "When ignorance is bliss, 'tis folly to be wise.",
  "Good judgement is the result of experience and
experience the result of bad judgement."
   ]

scores = []
for sent in sentences:
  score = sentiment.polarity_scores(sent)
  scores.append(score)

df = pd.DataFrame(scores)
df['sentence'] = sentences
print("=> dataframe:",df)
print()

df['positive_sentiment'] = df['compound'] >= 0.5
print("=> dataframe:",df)
```

Listing 6.7 starts with two `import` statements and then initializes the variable `sentiment` as an instance of the class `SentimentIntensityAnalyzer` that is available from `nltk.sentiment.vader`.

The next code block initializes the variable `sentence` with a familiar string that you have seen in many code samples, and then displays the polarity of `sentence` by invoking the variable `sentiment` with the contents of `sentence`.

Next, the variable `sentences` is initialized as an array of several `sentences`, after which a `for` loop calculates the polarity score for each sentence and populates the array `scores` with those values.

The next portion of Listing 6.7 creates the data frame `df` with the contents of scores, and then appends the column `sentence` that is initialized with the contents of the variable `sentences`.

After displaying the contents of `df`, the final code snippet adds a new column called `positive_sentiment` consisting of the rows in `df` that have a positive sentiment (i.e., their value is at least 0.5) and then prints the new contents of `df`. Launch the code in Listing 6.7 to see the following output:

```
=> sentence: I love Chicago deep dish pizza.
=> polarity: {'neg': 0.0, 'neu': 0.488, 'pos': 0.512,
'compound': 0.6369}

=> dataframe:      neg    neu  ...  compound   sentence
0  0.000  0.716  ...    0.2466             Our plan was
not without merit
1  0.000  0.615  ...    0.6124  I like the pizza
toppings but I do not like th...
2  0.205  0.795  ...   -0.4767  The only thing worse
than being talked about, ...
3  0.000  0.775  ...    0.4404  Everything is funny, as
long as it's happening...
4  0.163  0.392  ...    0.6486    When ignorance is
bliss, 'tis folly to be wise.
5  0.190  0.652  ...   -0.1531  Good judgement is the
result of experience and...

[6 rows x 5 columns]

=> dataframe:      neg  ...  positive_sentiment
0  0.000  ...               False
1  0.000  ...                True
2  0.205  ...               False
3  0.000  ...               False
4  0.163  ...                True
5  0.190  ...               False

[6 rows x 6 columns]
```

SENTIMENT ANALYSIS WITH FLAIR

Flair is an open source Python-based library that performs various NLP-tasks, such as NER and POS tagging:

https://github.com/flairNLP/flair

In addition to sentiment analysis, Flair provides the following functionality:

- a biomedical NER library
- a text embedding library
- a PyTorch NLP framework

The final example of sentiment analysis is Listing 6.8 shows the contents of the Python script `flair_sentiment.py` that use the `Flair` library to perform sentiment analysis. Make sure that you invoke `pip3 install flair`.

NOTE *You might need to use Python 3.7 to install* `flair`.

LISTING 6.8: flair_sentiment.py

```
# pip3 install flair

from flair.models import TextClassifier
from flair.data import Sentence

classifier = TextClassifier.load('en-sentiment')

sent = "I love Chicago deep dish pizza."
ssent = Sentence(sent)
classifier.predict(ssent)
print('Sentence:  ', sent)
print('Sentiment: ', ssent.labels)
print()

sent = "I love Chicago deep dish pizza!"
ssent = Sentence(sent)
classifier.predict(ssent)
print('Sentence:  ', sent)
print('Sentiment: ', ssent.labels)
print()

sent = "I love Chicago deep dish pizza!!!"
ssent = Sentence(sent)
```

```
classifier.predict(ssent)
print('Sentence:  ', sent)
print('Sentiment: ', ssent.labels)
```

Listing 6.8 starts with two `import` statements and then initializes several variables, starting with the variable `classifier` (which is a model) as an instance of the `TextClassifier` class.

The next three code blocks in the same style as the code in Listing 6.4 initialize the variable `sent` as three text strings containing zero, one, and three exclamation points. After this, the sentiment of each sentence is calculated via the `predict()` method of the `classifier` variable and then displayed. Launch the code in Listing 6.8 to see the following output:

```
Sentence:   I love Chicago deep dish pizza.
Sentiment:  [POSITIVE (0.999)]

Sentence:   I love Chicago deep dish pizza!
Sentiment:  [POSITIVE (0.9996)]

Sentence:   I love Chicago deep dish pizza!!!
Sentiment:  [POSITIVE (0.9997)]
```

DETECTING SPAM

Spam classification has been an on-going challenge ever since the introduction of email (or soon thereafter). Spam filters often use a Naïve Bayes classifier (discussed earlier in this chapter). However, the nature of spam evolves over time, which means that spam classifiers must also evolve to handle new types of spam.

A spam classifier involves the following set of steps that are common to machine learning tasks:

- Step 1: labeling a training dataset
- Step 2: determining a set of features
- Step 3: splitting the dataset into training/validation/test data
- Step 4: training the classifier
- Step 5: making some predictions on new data

Step 1 involves a good mixture of legitimate email messages as well as spam email messages. Step 2 involves the typical tasks that are described in Chapter 3 (such as removing stop words, stemming, and calculating word frequencies). Step 3 requires a reasonably-sized dataset. If your dataset is small, you can split its content into training and testing data and omit the validation part (not the best solution).

Step 4 is a standard step, and you can use k-fold cross validation, which involves dividing the dataset into subsets (such as ten "folds") and then repeating the training on nine of the ten folds, using the omitted fold as the test data. Calculate the average error after completing the cross-fold validation. This technique is useful for small datasets.

The final step involves making predictions and determining the accuracy of those predictions.

LOGISTIC REGRESSION AND SENTIMENT ANALYSIS

Recall that classification problems involve predicting discrete outcomes, whereas regression problems involve predicting a value of a continuous variable. As you learned in a previous chapter, logistic regression is actually a classification algorithm (not a regression algorithm). Logistic regression works well when the features and the target have a relatively simple relationship.

The Sklearn `LogisticRegression` class has these arguments: `penalty`, `dual`, `tol`, `C`, `fit_intercept`, `intercept_scaling`, `class_weight`, `random_state`, `solver`, `max_iter`, `verbose`, `warm_start`, `n_jobs`, and `l1_ratio`.

Listing 6.9 shows the contents of `log_reg_spam.py` that illustrate how to perform sentiment analysis with NLTK.

LISTING 6.9: log_reg_spam.py

```
import numpy as np
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer

#for sklearn version 0.24.0:
from sklearn.linear_model import LogisticRegression

from sklearn.model_selection import train_test_split,
                                            cross_val_score
```

```
# you can download the SMSSpamCollection dataset here:
# https://archive.ics.uci.edu/ml/machine-learning-
databases/00228
# NB: header "type\ttext" was manually added to
SMSSpamCollection
df = pd.read_csv('SMSSpamCollection', delimiter = '\t')

print("First five rows (before):")
print(df.head(5))
print("-------------------------")

# map ham/spam to 0/1 values:
df['type'] = df['type'].map( {'ham':0 , 'spam':1} )

print("First five rows (after):")
print(df.head(5))
print("-------------------------")

# X contains text and y contains labels:
X = df.iloc[:, 1].values
y = df.iloc[:, 0].values

# perform train/test split on the data (75/25):
X_train, X_test, y_train, y_test = train_test_split(X, y,
                        test_size = 0.25, random_state = 0)
#print("X_train:",X_train)
#print("-------------------------")

vectorizer = TfidfVectorizer()
X_train = vectorizer.fit_transform(X_train)
X_test  = vectorizer.transform(X_test)

# train an instance of the LogisticRegression class:
classifier = LogisticRegression()
classifier.fit(X_train, y_train)

# make predictions for the X_test data:
y_pred = classifier.predict(X_test)
print("predictions:",y_pred)
print("-------------------------")
```

```
# create the confusion matrix:
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
print("confusion matrix:")
print(cm)

true_neg, false_pos = cm[0]
false_neg, true_pos = cm[1]
all_values = true_pos + true_neg + false_pos + false_neg

accuracy  = round((true_pos + true_neg) / all_values, 3)
precision = round((true_pos) / (true_pos + false_pos),3)
recall    = round((true_pos) / (true_pos + false_neg),3)
f1        = round(2 * (precision*recall) /
                                    (precision+recall),3)

print("--------------------------\n")
print('Accuracy:  {}'.format(accuracy))
print('Precision: {}'.format(precision))
print('Recall:    {}'.format(recall))
print('F1 Score:  {}'.format(f1))
```

Listing 6.9 is similar to the logistic regression code sample in Chapter 8, using Sklearn instead of Keras-based code. Specifically, the main difference involves the following code block instead of a Keras-based model from TensorFlow:

```
# train an instance of the LogisticRegression class:
classifier = LogisticRegression()
classifier.fit(X_train, y_train)
```

Launch the code in Listing 6.9 to see the following output:

```
=> First five rows (before):
   type                                               text
0   ham  Go until jurong point, crazy.. Available only ...
1   ham                      Ok lar... Joking wif u oni...
2  spam  Free entry in 2 a wkly comp to win FA Cup fina...
3   ham  U dun say so early hor... U c already then say...
4   ham  Nah I don't think he goes to usf, he lives aro...
-------------------------
```

```
=> First five rows (after):
   type                                                text
0     0  Go until jurong point, crazy.. Available only ...
1     0                      Ok lar... Joking wif u oni...
2     1  Free entry in 2 a wkly comp to win FA Cup fina...
3     0  U dun say so early hor... U c already then say...
4     0  Nah I don't think he goes to usf, he lives aro...
-------------------------

=> predictions: [0 1 0 ... 0 0 0]

-------------------------

=> confusion matrix:
[[1198    2]
 [  48  145]]

-------------------------

Accuracy:  0.964
Precision: 0.986
Recall:    0.751
F1 Score:  0.853
```

WORKING WITH COVID-19

Listing 6.10 shows the contents of `covid19.py` that illustrate how to train a model on the `covid19` dataset.

LISTING 6.10: covid19.py

```
import numpy as np
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
import matplotlib.pyplot as plt
import seaborn as sns
```

```
df = pd.read_csv("clean_covid19.csv", sep = ",")

X = df.iloc[:, [0, 1, 2, 3]].values
y = df.iloc[:, 4].values
y = y.astype('int')

print("Number of rows and columns in dataset:")
print(df.shape)

# count the number of 0 and 1 values for y:
print(df.groupby('severity_illness').count())

# the count of y values: 8 and 3351
# the dataset is highly imbalanced
# balance the target data via SMOTE:
from collections import Counter
from imblearn.over_sampling import SMOTE

sm = SMOTE(random_state = 42)
X, y = sm.fit_resample(X, y)

# split into training and test sets:
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y,
                        test_size = 0.25, random_state = 0)

# Feature Scaling
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

# libraries for performance metrics:
from sklearn.metrics import make_scorer
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import f1_score
from sklearn.model_selection import cross_validate
```

```
# dictionary with performance metrics:
scoring = {'accuracy':make_scorer(accuracy_score),
           'precision':make_scorer(precision_score),
           'recall':make_scorer(recall_score),
           'f1_score':make_scorer(f1_score)}

# instantiate classifier:
rfc_model = RandomForestClassifier()

# train model via cross-validation:
folds = 10
rfc = cross_validate(rfc_model, X, y, cv = folds, scoring
                                                = scoring)

print("accuracy: ",rfc['test_accuracy'].mean())
print("precision:",rfc['test_precision'].mean())
print("recall:  :",rfc['test_recall'].mean())
print("F1 score: ",rfc['test_f1_score'].mean())
```

Listing 6.10 is similar to Listing 8.4 in Chapter 8, albeit modified to use the following code block instead of a `DecisionTreeClassifier`:

```
from sklearn.ensemble import RandomForestClassifier
classifier = RandomForestClassifier(n_estimators = 10,
                criterion='entropy', random_state = 0)
```

In addition, notice that Listing 6.10 uses the `SMOTE` class to generate synthetic data because the `Covid19` dataset is highly imbalanced. Launch the code in Listing 6.10 to see the following output (truncated for ease of reading):

```
Number of rows and columns in dataset:
(3359, 5)

age  ...  severity_illness
0                        8
1                     3351

accuracy:  0.993136552705919
precision: 0.9920161922823759
recall:  : 0.9943283582089553
F1 score:  0.9931523221120553
```

WHAT ARE CHATBOTS?

Chatbots affect many parts of our lives. Chatbots have evolved into AI-based software programs that interact with users and attempt to provide them with satisfactory answers. Chatbots are available on devices (such as Siri, Alexa, and Google Assistant) and Websites. Chatbots can provide question-answer functionality or they provide task-oriented functionality, such as performing bookings for cars, airplanes, and hotels.

By today's standards, early chatbots had a rudimentary question–answer structure that tended to resemble flow charts. The response to a question was selected from a set of hard-coded answers, and if the answer did not satisfy the users, then the question was often routed to a human. Since chatbots vary in terms of quality and features, a metric called the *sensibleness and specificity average* (SSA) was developed to rate chatbots.

Companies use chatbots to reduce the human-based interactions with customers with the goal of streamlining customers' interactions with a company's services. Some chatbot-based services include,

- providing product-related customer support
- providing flight information
- connecting customers and their finances

Open Domain Chatbots

There are several interesting open domain chatbots available, some of which are shown in the following list:

- Cleverbot
- DialoGPT
- Meena
- Mitsuku
- XiaoIce

Meena is optimized for multiturn conversations and it scores well on the new metric (described earlier). Meena is a sequence-to-sequence model with an evolved transformer architecture that comprises 2.6 billion parameters. Meena predicts the actual response via perplexity, which is a measure of a language model's predictive ability, and it is used as its loss function. Meena avoids generating repetitive responses. The model builds multiple candidate responses and uses a classifier to select the best one.

Perform an online search for information pertaining to the other open domain chatbots in the preceding list.

Chatbot Types

Chatbots can be classified into two main groups: rule-based chatbots and self-learning chatbots.

Rule-based chatbots are trained on rules that are also used to provide answers to questions. However, these chatbots work best for simple questions and are less accurate for sophisticated questions.

By contrast, *self-learning chatbots* are trained via machine learning-based approaches and as you might expect, these chatbots provide better results. Moreover, self-learning chatbots can be divided into two broad categories: *retrieval-based* chatbots (which rely on heuristics to provide answers) and *generative* chatbots (which can generate answers beyond just pre-defined answers).

Logic Flow of Chatbots

Although chatbots can vary significantly in terms of their primary functionality, they generally perform the following sequence of steps:

1. Prepare a corpus of text-based responses.
2. Clean the data (as described in earlier chapters).
3. Select a vectorizer (such as `CountVectorizer` or `TfidfVectorizer`).
4. Prompt users for a question/query.
5. Calculate the cosine similarity of the question with the sentences in the corpus.
6. Determine which sentence has the highest cosine similarity.
7. Use the previously selected sentence as a response to users.

In case the highest cosine similarity is close to zero, there is no meaningful response for the query, so you can route the users to a human agent. Of course, the threshold value for "close to zero" is a number that you decide in advance, and perhaps it can be determined through experimentation.

Chatbot Abuses

There have been some well-known cases of attempts to crowd-source the training of chatbots that have gone awry. One such chatbot is Tay from Microsoft, which some users trained to make various types of highly inappropriate remarks about certain groups of people.

Another (more recent) example is the chatbot Lee Luda from Korea, which was designed to emulate a Korean university student. However, this chatbot was removed from Facebook because some users trained the chatbot to make derogatory slurs and hate speech that were directed toward certain groups of people.

Unfortunately, chatbots are likely to encounter these sorts of issues when the enhancement of a chatbot's capabilities involve crowd-sourced contributions.

Useful Links

If you are unfamiliar with chatbots, navigate to the following link that has thousands of registered chatbots:

https://botlist.co

Microsoft developed and open-sourced BlenderBot, which at its peak was the largest state-of-the-art chatbot, which is available online:

https://ai.facebook.com/blog/state-of-the-art-open-source-chatbot/

BlenderBot was benchmarked (and is significantly better) than Google's Meena chatbot, which is also available online:

https://github.com/google-research/google-research/tree/master/meena

If you are interested in chatbots in the health care field, here are two useful links:

https://topflightapps.com/ideas/chatbots-in-healthcare
https://emerj.com/ai-application-comparisons/chatbots-for-healthcare-comparison

If you plan to create a chatbot that is useful, then it probably needs to understand (human) natural language and be able to solve a task that might require multiple steps.

Although there are online tools that enable you to create chatbots that connect to an AI backend system (such as IBM Watson), the following link explains how to build a chatbot in Keras:

https://www.kdnuggets.com/2019/08/deep-learning-nlp-creating-chatbot-keras.html
https://analyticsindiamag.com/how-does-a-simple-chatbot-with-nltk-work/

The following link shows you how to create a chatbot based on a pre-trained transformers (discussed in Chapter 11) with PyTorch:

https://towardsdatascience.com/conversational-ai-chatbot-with-pretrained-transformers-using-pytorch-55b5e8882fd3

Finally, the following list of the top chatbots of 2021 might provide ideas and helpful insights: *https://www.netomi.com/best-ai-chatbot*.

SUMMARY

This chapter started with a description of two types of text recommendation, along with some code samples using Gensim and spaCy. Then you got a high-level description of recommendation systems, which are used in the online reviews of books, movies, and restaurants. Next, you learned about sentiment analysis, which is actually a subset of text classification. In essence, sentiment analysis attempts to assess the mood (positive, negative, or neutral) of a document (such as a review). Finally, you saw a Python-based code sample that uses logistic regression to predict spam email messages.

CHAPTER 7

TRANSFORMER, BERT, AND GPT

This chapter is primarily about the transformer architecture, the pretrained BERT model and its variants, and features of GPT-2 and GPT-3 from OpenAI. If you are familiar with some of these topics, skim through the material in this chapter and peruse the Python-based code samples.

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 transformer 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.

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 that is used in two or more sentences will have a word embedding that is specific to each sentence.

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 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 in this book, 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. 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 embeddings have zero context
2. Distributional word embeddings (word2vec, GloVe, and fastText):
   Based on the 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 (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)

Types of Attention and Algorithms

There are several types of attention mechanisms, three of which are listed here:

1. self-attention
2. global/soft
3. local/hard

Self-attention tries to determine how words in a sentence are interconnected with each other. *Multiheaded 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-based
- scaled dot product <= 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 link 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 CNN/RNN/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 six (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" encoder, 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 so-called "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 typically 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. Hence, 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 pretrained models for NLU and NLG. In fact, HuggingFace provides more than 30 pretrained 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, and XLNet.

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)

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 contents of `hf_transformer_ner.py` that illustrate how to perform an 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 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 QnA Tasks

Listing 7.2 shows the contents of `hf_transformer_qa.py` that illustrate 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 contents of `hf_transformer_sentiment.py` that illustrate 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 contents of `hf_transformer_mask.py` that illustrate 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 powerful NLP model created by Google.

WHAT IS T5?

Text-to-text transfer transformer (T5) is an encoder-decoder model that converts all NLP tasks into a text-to-text format:

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 pretrained on a multitask 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` enables you to specify configuration information, and its default values are similar to the T5-small architecture. Another useful class is `transformers.T5Tokenizer`, which 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 pretraining tasks instead of a fill-in-the-blank task

Although you can download code samples for T5, it might be simpler to experiment with T5 in this 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 pretrained model that is based on the transformer architecture that was developed in 2017 by Google. There are two versions of BERT called BERT Base and BERT Large. BERT Base consists of twelve layers (transformer blocks), twelve attention heads, and 110 million parameters. BERT

Large is a larger pretrained model that consists of 24 layers (transformer blocks), sixteen attention heads, and 340 million parameters.

BERT can be used in conjunction with the transformers library (discussed earlier in this chapter) that provides 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 sub-words that are created by WordPiece. These sub-words have the form "##s" or "##ed." The latter sub-words 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. As you would expect, the dataset is split into a training portion and a test portion, just as you have seen in code samples in previous chapters.

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 pretrained 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 Techniques

There are several important aspects of BERT that differentiate BERT from algorithms such as word2vec. First, BERT does not perform a stemming operation. Instead, BERT performs sub-word tokenization via WordPiece (discussed later in this chapter).

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. As a result, the same word that is used in a different context in two sentences will have different word embeddings. However, 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 here:

- MLM (masked language model)
- NSP (next sentence prediction)
- Special tokens ([CLS] and [SEP])
- Language mask
- WordPiece (sub-word tokenization)
- SentencePiece

Each topic in the preceding list is discussed briefly in the following subsections.

What is MLM?

MLM is an acronym for Masked Language Model. MLM is a BERT pre-training task, during which BERT processed the contents of Wikipedia (and also the BookCorpus). 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 is an acronym 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.

One of the tasks of BERT 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 sentence:

```
Pizza with four toppings and trimmings.
```

The BERT tokenization of the preceding sentence is here:

```
['[CLS]', 'pizza', 'with', 'four', 'topping', '##s',
'and', 'trim', '##ming', '##s', '.', '[SEP]']
```

Listing 7.5 shows the contents of `bert_special_tokens.py` that illustrate 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 two `import` statements and then initializes the variable `tokenizer` as an instance from a pretrained 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 contents of `bert_encoding_plus.py` that illustrate 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 the 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, 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

Out Of Vocabulary (OOV) refers to words in a corpus that do not belong to a vocabulary. When an OOV word is encountered, BERT splits the word into sub-words, which is known as *sub-word tokenization*. The same process is applied to rare words.

Sub-word tokenization algorithms are based on a heuristic (something that's intuitive and often produces the correct answer). Specifically, words that appear more frequently words are assigned unique IDs. However, lower frequency words are split into sub-words that retain the meaning of the lower frequency words. The following list contains four important sub-word tokenization algorithms:

- byte-pair encoding (BPE)
- SentencePiece

- unigram language model
- WordPiece (used in BERT)

Byte-pair encoding for sub-words represents frequent words with fewer symbols and less frequent words with more symbols. BPE is a bottom-up sub-word tokenization algorithm that learns a sub-word vocabulary of a certain size (the vocabulary size is a hyperparameter).

The first step in this technique involves splitting every word into unicode characters, each of which corresponds to a symbol in the final vocabulary. Now 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 sub-word tokenization algorithm that is very similar to BPE. 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 sub-word tokenizations 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 reconstruct the original word from its word pieces. Note that ELMo provides word-level (not sub-word) 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.

Byte pair encoding (BPE) is one of the algorithms that is used in the GPT family of models. BPE (also known as diagram 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`. Perform the replacement, which results in the following text string:

```
ZabdZabac (where Z = aa)
```

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 sub-word tokenizer and a detokenizer for NLP that performs sub-word segmentation. SentencePiece also supporting BPE and unigram language model. The original arXiv paper that describes SentencePiece in detail is available online:

https://arxiv.org/abs/1808.06226v1

SENTENCE SIMILARITY IN BERT

As you learned in a previous chapter, 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 illustrates 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, and other techniques are also available. 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 contents of `bert_context.py` that illustrate how BERT generates a different word vector for the same word that is used in a different context.

If you do not already have the transformers library installed, launch the following command in a command shell:

```
pip3 install transformers
```

NOTE *This code downloads a 536 M 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 = "cell mate"
```

```
# 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 here:

```
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 contents of `bert_tokens1.py` that illustrate 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 contents of `bert_tokens2.py` that illustrate 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, using 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`. 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 here:

```
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

A lite BERT for self-supervised learning of language representations (ALBERT) was created by Google Research and Toyota Technological Institute. 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 this 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 sentence-order prediction (SOP), which is an improvement over next sentence prediction (NSP). 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 55% 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.

As an aside, *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 some modifications of BERT's hyperparameters. 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.

Perform an online search to find more detailed information about the list of BERT-related models in the first part of this section.

Surpassing Human Accuracy: deBERTa

The deBERTa model from Microsoft recently surpassed human accuracy, as described in the following link:

https://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 manages to exceed the human performance score of 89.8.

Microsoft intends to integrate DeBERTa with the Turing natural language representation model Turing NLRv4 (also from Microsoft).

What is Google Smith?

The SMITH model from Google is a model for analyzing documents. The SMITH model is trained to understand passages within the context of the 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

Generative Pre-Training (GPT), or sometimes called generative pretraining transformers, is a pretrained NLP-based model that was developed by OpenAI. GPT is trained with unlabeled data via unsupervised pretraining (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 (the most recent version) that was released in June, 2020. Both GPT-2 and GPT-3 are discussed later in this chapter.

Coincidentally: according to the Lottery Ticket Hypothesis, in every sufficiently deep neural network, there is a smaller sub-network 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 question-and-answer tasks. 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 other Python 3.7.x versions 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 contents of `gpt2_sentiment.py` that illustrate how to perform sentiment analysis in GPT2.

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`. 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 contents of `gpt2_qna.py` that illustrate how to perform sentiment analysis in GPT2.

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 `qc_pair` is initialized as a pair of question/answer strings.

Next, the variable `result` is initialized with the value that is returned by 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'}
```

If you remove the `if` statement from Listing 7.11, you might see the following error message:

```
#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 contents of `gpt2_text_gen.py` that illustrate how to use generated text from an input string in GPT2. 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 contents of `gpt2_qna.py` that illustrate how to perform sentiment analysis in GPT2. 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 displays 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 contents of `pytorch_gpt_next_word.py` that illustrate 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 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 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 in the following output).

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
```

WHAT IS GPT-3?

GPT-3 is an extension of the GPT-2 model that involves more layers and data. For example, the largest model has 96 attention layers, each of which contains 96 × 128 dimension heads. 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.

Give GPT-3 an initial sequence of words and GPT-3 will generate various responses, such as code, news articles, poems, and 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 GPT-3 model with 175 billion parameters was trained on an unlabeled dataset consisting of almost 500 billion tokens from a variety of sources. 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.

GPT-2 and GPT-3 have similar architectures, which is to say, they both have a "vanilla" transformer. GPT-2 has 1.5 billion parameters whereas GPT-3 has 175 billion parameters, which is more than 100 times larger than GPT-2.

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 (which created GPT-3). GPT-3 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 tweeting at 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!"

[1] *https://www.businessinsider.com/elon-musk-poem-tweets-gpt-3-openai-2020-8.*

What is the Goal?

The aim of the GPT-3 pre-trained model is to directly evaluate the model on the test-related data of new tasks. 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 a 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:

- a button that increments a total by USD 3
- a button that decrements a total by USD 5
- 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 made dire predictions.

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. GPT-3 can then write brief articles that often seem to be written by humans.

However, any trained model has limitations, including GPT-3. 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 that told a fake patient (who expressed suicidal thoughts) to 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 125 M to 175 B parameters), and the smallest GPT-3 model is about the size of BERT-Base and RoBERTa-Base, with twelve attention layers that in turn have 12 × 64 dimension 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.

GPT versus BERT

There are some important differences between GPT-2 and BERT. Specifically, GPT-2 is *not* bidirectional and has no concept of masking. In addition, 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 pretraining. Moreover, the fine-tuning process necessitates task-specific sample data.

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 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 (a) sees a description of the task and (b) one example of the task. *No gradient updates are performed*.

A *few-shot* learner is a model that (a) sees a description of the task and (b) a few examples of the task. *No gradient updates are performed*, and a "few" examples can involve between ten 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 Task Performance

For most models, the task of translating sentences from English to Italian involves thousands of sentence pairs for those models to learn how 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 ten training examples.

GPT-3 is not only a few-short learner, but it can also perform as a zero-short 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.

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 under performs on multiple-choice test questions.

THE SWITCH TRANSFORMER: ONE TRILLION PARAMETERS

As this book goes to print, 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 was the former largest model ever created, as much as 4X faster than the previous record holder called T5-XXL, and recently superseded by a 12 trillion parameter model from Facebook.[2]

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 book. 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

[2] *https://www.nature.com/articles/d41586-020-03348-4.*

associated with large-scale deployment of AI systems, how algorithms contribute to 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 (coined 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, referred to as a "50-year-old problem in biology." AlphaFold won the competition to solve this problem 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 [AlphaFold] model from group 427 gave us our structure in half an hour, after we had spent a decade trying everything."[2]

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 NLP, and now 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.

APPENDIX A

INTRODUCTION TO REGULAR EXPRESSIONS

This appendix introduces you to regular expressions, which are a powerful language feature in Python. Since regular expressions are available in other programming languages (such as JavaScript and Java), the knowledge that you gain from the material in this appendix will be useful to you outside of Python.

Why would anyone be interested in learning regular expressions in a book for NLP with Python? The answer is threefold. First, the Pandas library supports regular expressions, which demonstrates that regular expressions are relevant to Pandas. Second, if you plan to use Pandas extensively or perhaps also work with NLP, then regular expressions will prove useful because of the ease with which you can solve certain types of tasks (such as removing HTML tags) with regular expressions. Third, the knowledge you gain from the material in this appendix will instantly transfer to other languages that support regular expressions.

This appendix contains a mixture of code blocks and complete code samples, with varying degrees of complexity, that are suitable for beginners as well as people who have had some exposure to regular expressions. In fact, you have probably used (albeit simple) regular expressions in a command line on a laptop, whether it be Windows, UNIX, or Linux-based systems.

The first part of this appendix shows how to define regular expressions with digits and letters (uppercase as well as lowercase), and also how to use character classes in regular expressions. We examine character sets and character classes.

The second portion discusses the Python `re` module, which contains several useful methods, such as the `re.match()` method for matching groups of characters, the `re.search()` method to perform searches in character strings, and the `findAll()` method. We show how to use character classes (and how to group them) in regular expressions.

The final portion of this appendix contains an assortment of code samples, such as modifying text strings, splitting text strings with the `re.split()` method, and substituting text strings with the `re.sub()` method.

As you read the code samples in this appendix, some concepts and facets of regular expressions might make you feel overwhelmed with the density of the material if you are a novice. However, practice and repetition will help you become comfortable with regular expressions.

WHAT ARE REGULAR EXPRESSIONS?

Regular expressions are referred to as REs, or regexes, or regex patterns, and they enable you to specify expressions that can match specific "parts" of a string. For instance, you can define a regular expression to match a single character or digit, a telephone number, a zip code, or an email address. You can use metacharacters and character classes (defined in the next section) as part of regular expressions to search text documents for specific patterns. As you learn how to use REs, you will find other ways to use them, as well.

The `re` module (added in Python 1.5) provides Perl-style regular expression patterns. Note that earlier versions of Python provided the `regex` module that was removed in Python 2.5. The `re` module provides an assortment of methods (discussed later in this appendix) for searching text strings or replacing text strings, which is similar to the basic search and/or replace functionality that is available in word processors (but usually without regular expression support). The `re` module also provides methods for splitting text strings based on regular expressions.

Before delving into the methods in the `re` module, you need to learn about metacharacters and character classes, which are the topic of the next section.

METACHARACTERS IN PYTHON

Python supports a set of metacharacters, most of which are the same as the metacharacters in other scripting languages such as Perl, as well as programming languages such as JavaScript and Java. The complete list of metacharacters in Python is here:

```
. ^ $ * + ? { } [ ] \ | ( )
```

The meaning of the preceding metacharacters is as follows:

- `?` (matches 0 or 1): the expression `a?` matches the string `a` (but not `ab`)
- `*` (matches 0 or more): the expression `a*` matches the string `aaa` (but not `baa`)
- + (matches 1 or more): the expression a+ matches aaa (but not baa)
- `^` (beginning of line): the expression `^[a]` matches the string `abc` (but not `bc`)
- `$` (end of line): `[c]$` matches the string `abc` (but not `cab`)
- `.` (a single dot): matches any character (except newline)

Sometimes you need to match the metacharacters themselves rather than their representation, which can be done in two ways. The first way involves "escaping" their symbolic meaning with the backslash ("\") character. Thus, the sequences `\?`, `\*`, `\+`, `\^`, `\$`, and `\.` represent the literal characters instead of their symbolic meaning. You can also "escape" the backslash character with the sequence "`\\.`" If you have two consecutive backslash characters, you need an additional backslash for each of them, which means that "`\\\\`" is the "escaped" sequence for "`\\.`"

The second way is to list the metacharacters inside a pair of square brackets. For example, `[+?]` treats the two characters "+" and "?" as literal characters instead of metacharacters. The second approach is obviously more compact and less prone to error (it's easy to forget a backslash in a long sequence of metacharacters). As you might surmise, the methods in the `re` module support metacharacters.

NOTE *The "^" character that is to the left (and outside) of a sequence in square brackets (such as* `^[A-Z]`*) "anchors" the regular expression to the beginning of a line, whereas the "^" character that is the first character inside a pair of square brackets* negates *the regular expression (such as* `[^A-Z]`*) inside the square brackets.*

The interpretation of the "^" character in a regular expression depends on its location in a regular expression, as shown here:

- "`^[a-z]`" means any string that starts with any lowercase letter
- "`[^a-z]`" means any string that does *not* contain any lowercase letters
- "`^[^a-z]`" means any string that starts with anything *except* a lowercase letter
- "`^[a-z]$`" means a single lowercase letter
- "`^[^a-z]$`" means a single character (including digits) that is *not* a lowercase letter

As a quick preview of the `re` module that is discussed later in this appendix, the `re.sub()` method enables you to remove characters (including metacharacters) from a text string. For example, the following code snippet removes all occurrences of a forward slash ("/") and the plus sign ("+") from the variable `str`:

```
>>> import re
>>> str = "this string has a / and + in it"
>>> str2 = re.sub("[/]+","",str)
>>> print('original:',str)
original: this string has a / and + in it
>>> print('replaced:',str2)
replaced: this string has a and + in it
```

We can easily remove occurrences of other metacharacters in a text string by listing them inside the square brackets, just as we have done in the preceding code snippet.

Listing A.1 shows the contents of `RemoveMetaChars1.py` that illustrate how to remove other metacharacters from a line of text.

LISTING A.1: RemoveMetaChars1.py

```
import re

text1 = "meta characters ? and / and + and ."
text2 = re.sub("[/\.*?=+]+","",text1)

print('text1:',text1)
print('text2:',text2)
```

Let's examine the contents of Listing A.1. First of all, the term `[/\.*?=+]` matches a forward slash ("/"), a dot (".."), a question mark ("?"), an equals sign ("="), or a plus sign ("+"). Notice that the dot "." is preceded by a backslash character "\." Doing so "escapes" the meaning of the "." metacharacter (which matches any single non-whitespace character) and treats it as a literal character.

Thus, the term `[/\.*?=+]+` means "one or more occurrences of any of the metacharacters—treated as literal characters—inside the square brackets."

Consequently, the expression `re.sub("[/\.*?=+]+","",text1)` matches any occurrence of the previously listed metacharacters, and then

replaces them with an empty string in the text string specified by the variable `text1`. The output from Listing A.1 is here:

```
text1: meta characters ? and / and + and .
text2: meta characters and and and
```

Later in this appendix, we discuss other functions in the `re` module that enable you to modify and split text strings.

CHARACTER SETS IN PYTHON

A single digit in base 10 is a number between 0 and 9 inclusive, which is represented by the sequence `[0-9]`. Similarly, a lowercase letter can be any letter between a and z, which is represented by the sequence `[a-z]`. An uppercase letter can be any letter between `A` and `Z`, which is represented by the sequence `[A-Z]`.

The following code snippets illustrate how to specify sequences of digits and sequences of character strings using a shorthand notation that is much simpler than specifying every matching digit:

- `[0-9]` matches a single digit
- `[0-9][0-9]` matches 2 consecutive digits
- `[0-9]{3}` matches 3 consecutive digits
- `[0-9]{2,4}` matches 2, 3, or 4 consecutive digits
- `[0-9]{5,}` matches 5 or more consecutive digits
- `^[0-9]+$` matches a string consisting solely of digits

You can define similar patterns using uppercase or lowercase letters in a way that is much simpler than explicitly specifying every lowercase letter or every uppercase letter:

- `[a-z][A-Z]` matches a single lowercase letter that is followed by 1 uppercase letter
- `[a-zA-Z]` matches any upper- or lowercase letter

Working with "^" and "\"

The purpose of the "^" character depends on its context in a regular expression. For example, the following expression matches a text string that starts with a digit:

```
^[0-9].
```

However, the following expression matches a text string that does *not* start with a digit because of the "^" metacharacter that is at the beginning of an expression in square brackets as well as the "^" metacharacter that is to the left (and outside) the expression in square brackets (which you learned in a previous note):

```
^[^0-9]
```

Thus, the "^" character inside a pair of matching square brackets ("[]") negates the expression immediately to its right that is also located inside the square brackets.

The backslash ("\") allows you to "escape" the meaning of a metacharacter. Consequently, a dot "." matches a single character (except for whitespace characters), whereas the sequence "\." matches the dot "." character. Other examples involving the backslash metacharacter are here:

- `\.H.*` matches the string `.Hello`
- `H.*` matches the string `Hello`
- `H.*\.` matches the string `Hello.`
- `.ell.` matches the string `Hello`
- `.*` matches the string `Hello`
- `\..*` matches the string `.Hello`

CHARACTER CLASSES IN PYTHON

Character classes are convenient expressions that are shorter and simpler than their "bare" counterparts that you saw in the previous section. Some convenient character sequences that express patterns of digits and letters are as follows:

- `\d` matches a single digit
- `\w` matches a single character (digit or letter)
- `\s` matches a single whitespace (space, newline, return, or tab)
- `\b` matches a boundary between a word and a nonword
- `\n, \r, \t` represent a newline, a return, and a tab, respectively
- `\` "escapes" any character

Based on the preceding definitions, `\d+` matches one or more digits and `\w+` matches one or more characters, both of which are more compact expressions than using character sets. In addition, we can reformulate the expressions in the previous section:

- `\d` is the same as `[0-9]` and `\D` is the same as `[^0-9]`
- `\s` is the same as `[ \t\n\r\f\v]` and it matches any non-whitespace character, whereas \S is the opposite (it matches `[^ \t\n\r\f\v]`)
- `\w` is the same as `[a-zA-Z0-9_]` and it matches any alphanumeric character, whereas `\W` is the opposite (it matches `[^a-zA-Z0-9_]`)

Additional examples are as follows:

- `\d{2}` is the same as `[0-9][0-9]`
- `\d{3}` is the same as `[0-9]{3}`
- `\d{2,4}` is the same as `[0-9]{2,4}`
- `\d{5,}` is the same as `[0-9]{5,}`
- `^\d+$` is the same as `^[0-9]+$`

The curly braces ("`{}`") are called *quantifiers*, and they specify the number (or range) of characters in the expressions that precede them.

MATCHING CHARACTER CLASSES WITH THE `re` MODULE

The `re` module provides the following methods for matching and searching one or more occurrences of a regular expression in a text string:

- `match()`: Determine if the RE matches at the *beginning* of the string
- `search()`: Scan through a string, looking for *any* location where the RE matches
- `findall()`: Find *all* substrings where the RE matches and return them as a list
- `finditer()`: Find all substrings where the RE matches and return them as an iterator

NOTE *The* `match()` *function only matches patterns to the start of string.*

The next section shows you how to use the `match()` function in the `re` module.

USING THE `re.match()` METHOD

The `re.match()` method attempts to match RE patterns in a text string (with optional flags), and it has the following syntax:

```
re.match(pattern, string, flags = 0)
```

The `pattern` parameter is the regular expression that you want to match in the `string` parameter. The flags parameter allows you to specify multiple flags using the bitwise OR operator that is represented by the pipe "|" symbol.

The `re.match` method returns a match object on success and `None` on failure. Use the `group(num)` or `groups()` function of the match object to get a matched expression.

- `group(num = 0)`: This method returns the entire match (or specific subgroup `num`)
- `groups()`: This method returns all matching subgroups in a tuple (empty if there weren't any)

NOTE *The* `re.match()` *method only matches patterns from the start of a text string, which is different from the* `re.search()` *method discussed later in this appendix.*

The following code block illustrates how to use the `group()` function in regular expressions:

```
>>> import re
>>> p = re.compile('(a(b)c)de')
>>> m = p.match('abcde')
>>> m.group(0)
'abcde'
>>> m.group(1)
'abc'
>>> m.group(2)
'b'
```

Notice that the higher numbers inside the `group()` method match more deeply nested expressions that are specified in the initial regular expression.

Listing A.2 shows the contents of `MatchGroup1.py` that illustrate how to use the `group()` function to match an alphanumeric text string and an alphabetic string.

LISTING A.2: MatchGroup1.py

```
import re

line1 = 'abcd123'
line2 = 'abcdefg'
mixed = re.compile(r"^[a-z0-9]{5,7}$")
line3 = mixed.match(line1)
line4 = mixed.match(line2)
```

```
print('line1:',line1)
print('line2:',line2)
print('line3:',line3)
print('line4:',line4)
print('line5:',line4.group(0))

line6 = 'a1b2c3d4e5f6g7'
mixed2 = re.compile(r"^([a-z]+[0-9]+){5,7}$")
line7 = mixed2.match(line6)

print('line6:',line6)
print('line7:',line7.group(0))
print('line8:',line7.group(1))

line9 = 'abc123fgh4567'
mixed3 = re.compile(r"^([a-z]*[0-9]*){5,7}$")
line10 = mixed3.match(line9)
print('line9:',line9)
print('line10:',line10.group(0))
```

The output from Listing A.2 is as follows:

```
line1: abcd123
line2: abcdefg
line3: <_sre.SRE_Match object at 0x100485440>
line4: <_sre.SRE_Match object at 0x1004854a8>
line5: abcdefg
line6: a1b2c3d4e5f6g7
line7: a1b2c3d4e5f6g7
line8: g7
line9: abc123fgh4567
line10: abc123fgh4567
```

Notice that `line3` and `line7` involve two similar but different regular expressions. The variable `mixed` specifies a sequence of lowercase letters followed by digits, where the length of the text string is also between 5 and 7. The string `abcd123` satisfies all of these conditions.

However, `mixed2` specifies a pattern consisting of one or more pairs, where each pair contains one or more lowercase letters followed by one or more digits, where the length of the matching pairs is also between 5 and 7. In this case, the string `abcd123` as well as the string `a1b2c3d4e5f6g7` both satisfy these criteria.

The third regular expression `mixed3` specifies a pair such that each pair consists of zero or more occurrences of lowercase letters and zero or more

occurrences of a digit, and also that the number of such pairs is between 5 and 7. As you can see from the output, the regular expression in `mixed3` matches lowercase letters and digits in any order.

In the preceding example, the regular expression specified a range for the length of the string, which involves a lower limit of 5 and an upper limit of 7. However, you can also specify a lower limit without an upper limit (or an upper limit without a lower limit).

Listing A.3 shows the contents of `MatchGroup2.py` that illustrate how to use a regular expression and the `group()` function to match an alphanumeric text string and an alphabetic string.

LISTING A.3: MatchGroup2.py

```
import re

alphas = re.compile(r"^[abcde]{5,}")
line1 = alphas.match("abcde").group(0)
line2 = alphas.match("edcba").group(0)
line3 = alphas.match("acbedf").group(0)
line4 = alphas.match("abcdefghi").group(0)
line5 = alphas.match("abcdefghi abcdef")

print('line1:',line1)
print('line2:',line2)
print('line3:',line3)
print('line4:',line4)
print('line5:',line5)
```

Listing A.3 initializes the variable `alphas` as a regular expression that matches any string that starts with one of the letters `a` through `e`, and consists of at least five characters. The next portion of Listing A.3 initializes the four variables `line1`, `line2`, `line3`, and `line4` by means of the `alphas` RE that is applied to various text strings. These four variables are set to the first matching group by means of the expression `group(0)`.

The output from Listing A.3 is as follows:

```
line1: abcde
line2: edcba
line3: acbed
line4: abcde
line5: <_sre.SRE_Match object at 0x1004854a8>
```

Listing A.4 shows the contents of `MatchGroup3.py` that illustrate how to use a regular expression with the `group()` function to match words in a text string.

LISTING A.4: MatchGroup3.py

```
import re

line = "Giraffes are taller than elephants";

matchObj = re.match( r'(.*) are(\.*)', line, re.M|re.I)

if matchObj:
   print("matchObj.group()  : ", matchObj.group())
   print("matchObj.group(1) : ", matchObj.group(1))
   print("matchObj.group(2) : ", matchObj.group(2))
else:
   print("matchObj does not match line:", line)
```

The code in Listing A.4 produces the following output:

```
matchObj.group()  : Giraffes are
matchObj.group(1) : Giraffes
matchObj.group(2) :
```

Listing A.4 contains a pair of delimiters separated by a pipe ("|") symbol. The first delimiter is `re.M` for "multiline" (this example contains only a single line of text), and the second delimiter `re.I` means "ignore case" during the pattern matching operation. The `re.match()` method supports additional delimiters, as discussed in the next section.

OPTIONS FOR THE `re.match()` METHOD

The `match()` method supports various optional modifiers that affect the type of matching that will be performed. As you saw in the previous example, you can also specify multiple modifiers separated by the OR ("|") symbol. Additional modifiers that are available for RE are as follows:

- `re.I` performs case-insensitive matches (see previous section)
- `re.L` interprets words according to the current locale
- `re.M` makes $ match the end of a line and makes ^ match the start of any line

- `re.S` makes a period (".") match any character (including a newline)
- `re.U` interprets letters according to the Unicode character set

Experiment with these modifiers by writing Python code that uses them in conjunction with different text strings.

MATCHING CHARACTER CLASSES WITH THE `re.search()` METHOD

As you saw earlier in this appendix, the `re.match()` method only matches from the beginning of a string, whereas the `re.search()` method can successfully match a substring anywhere in a text string.

The `re.search()` method takes two arguments, a regular expression pattern and a string, and then searches for the specified pattern in the given string. The `search()` method returns a match object (if the search was successful) or `None`.

As a simple example, the following searches for the pattern `tasty` followed by a five-letter word:

```
import re

str = 'I want a tasty pizza'
match = re.search(r'tasty \w\w\w\w\w', str)

if match:
  ## 'found tasty pizza'
  print('found', match.group())
else:
  print('Nothing tasty here')
```

The output of the preceding code block is

```
found tasty pizza
```

The following code block further illustrates the difference between the `match()` method and the `search()` method:

```
>>> import re
>>> print(re.search('this', 'this is the one').span())
(0, 4)
```

```
>>>
>>> print(re.search('the', 'this is the one').span())
(8, 11)
>>> print(re.match('this', 'this is the one').span())
(0, 4)
>>> print(re.match('the', 'this is the one').span())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'NoneType' object has no attribute 'span'
```

MATCHING CHARACTER CLASSES WITH THE `findAll()` METHOD

Listing A.5 shows the contents of the Python script `RegEx1.py` that illustrate how to define simple character classes that match various text strings.

LISTING A.5: RegEx1.py

```
import re

str1 = "123456"
matches1 = re.findall("(\d+)", str1)

print('matches1:',matches1)

str1 = "123456"
matches1 = re.findall("(\d\d\d)", str1)
print('matches1:',matches1)

str1 = "123456"
matches1 = re.findall("(\d\d)", str1)
print('matches1:',matches1)

print
str2 = "1a2b3c456"
matches2 = re.findall("(\d)", str2)
print('matches2:',matches2)
```

```
print
str2 = "1a2b3c456"
matches2 = re.findall("\d", str2)
print('matches2:',matches2)

print
str3 = "1a2b3c456"
matches3 = re.findall("(\w)", str3)
print('matches3:',matches3)
```

Listing A.5 contains simple regular expressions (which you have seen already) for matching digits in the variables `str1` and `str2`. The final code block of Listing A.5 matches every character in the string `str3`, effectively "splitting" `str3` into a list where each element consists of one character. The output from Listing A.5 is here (notice the blank lines after the first three output lines):

```
matches1: ['123456']
matches1: ['123', '456']
matches1: ['12', '34', '56']

matches2: ['1', '2', '3', '4', '5', '6']

matches2: ['1', '2', '3', '4', '5', '6']

matches3: ['1', 'a', '2', 'b', '3', 'c', '4', '5', '6']
```

Finding Capitalized Words in a String

Listing A.6 shows the contents of the Python script `FindCapitalized.py` that illustrate how to define simple character classes that match various text strings.

LISTING A.6: FindCapitalized.py

```
import re

str = "This Sentence contains Capitalized words"
caps = re.findall(r'[A-Z][\w\.-]+', str)

print('str: ',str)
print('caps:',caps)
```

Listing A.6 initializes the string variable `str` and the RE `caps` that matches any word that starts with a capital letter because the first portion of `caps` is the pattern `[A-Z]` that matches any capital letter between `A` and `Z` inclusive.

The output of Listing A.6 is here:

```
str: This Sentence contains Capitalized words
caps: ['This', 'Sentence', 'Capitalized']
```

ADDITIONAL MATCHING FUNCTION FOR REGULAR EXPRESSIONS

After invoking any of the methods `match()`, `search()`, `findAll()`, or `finditer()`, you can invoke additional methods on the "matching object." An example of this functionality using the `match()` method is as follows:

```
import re

p1 = re.compile('[a-z]+')
m1 = p1.match("hello")
```

In the preceding code block, the `p1` object represents the compiled regular expression for one or more lowercase letters, and the "matching object" `m1` object supports the following methods:

- `group()` returns the string matched by the RE
- `start()` returns the starting position of the match
- `end()` returns the ending position of the match
- `span()` returns a tuple containing the (start, end) positions of the match

As a further illustration, Listing A.7 shows the contents of `SearchFunction1.py` that illustrate how to use the `search()` method and the `group()` method.

LISTING A.7: SearchFunction1.py

```
import re

line = "Giraffes are taller than elephants";

searchObj = re.search( r'(.*) are(\.*)', line, re.M|re.I)
```

```
if searchObj:
   print("searchObj.group()  : ", searchObj.group())
   print("searchObj.group(1) : ", searchObj.group(1))
   print("searchObj.group(2) : ", searchObj.group(2))
else:
   print("searchObj does not match line:", line)
```

Listing A.7 contains the variable line that represents a text string, and the variable `searchObj` is an RE involving the `search()` method and pair of pipe-delimited modifiers (discussed in more detail in the next section). If `searchObj` is not null, the `if/else` conditional code in Listing A.7 displays the contents of the three groups resulting from the successful match with the contents of the variable line.

The output from Listing A.7 is as follows:

```
searchObj.group()  : Giraffes are
searchObj.group(1) : Giraffes
searchObj.group(2) :
```

GROUPING WITH CHARACTER CLASSES IN REGULAR EXPRESSIONS

In addition to the character classes that you have seen earlier in this appendix, you can specify subexpressions of character classes.

Listing A.8 shows the contents of `Grouping1.py` that illustrate how to use the `search()` method.

Listing A.8: Grouping1.py

```
import re

p1 = re.compile('(ab)*')

print('match1:',p1.match('abababababab').group())
print('span1: ',p1.match('abababababab').span())

p2 = re.compile('(a)b')
m2 = p2.match('ab')
print('match2:',m2.group(0))
print('match3:',m2.group(1))
```

Listing A.8 starts by defining the RE `p1` that matches zero or more occurrences of the string `ab`. The first `print()` statement displays the result of using the `match()` function of `p1` (followed by the `group()` function) against a string, and the result is a string. This illustrates the use of "method chaining," which eliminates the need for an intermediate object (as shown in the second code block). The second `print()` statement displays the result of using the `match()` function of `p1`, followed by applying the `span()` function, against a string. In this case, the result is a numeric range.

The second part of Listing A.8 defines the RE `p2` that matches an optional letter `a` followed by the letter `b`. The variable `m2` invokes the match method on `p2` using the string `ab`. The third `print()` statement displays the result of invoking `group(0)` on `m2`, and the fourth `print()` statement displays the result of involving `group(1)` on `m2`. Both results are substrings of the input string `ab`. Recall that `group(0)` returns the highest level match that occurred, and `group(1)` returns a more "specific" match that occurred, such as one that involves the parentheses in the definition of `p2`. The higher the value of the integer in the expression `group(n)`, the more specific the match.

The output from Listing A.8 is here:

```
match1: abababababab
span1: (0, 10)
match2: ab
match3: a
```

USING CHARACTER CLASSES IN REGULAR EXPRESSIONS

This section contains some examples that illustrate how to use character classes to match various strings and how to use delimiters to split a text string. For example, one common date string involves a date format of the form MM/DD/YY. Another common scenario involves records with a delimiter that separates multiple fields. Usually such records contain one delimiter, but as you will see, Python makes it very easy to split records using multiple delimiters.

Matching Strings with Multiple Consecutive Digits

Listing A.9 shows the contents of the Python script `MatchPatterns1.py` that illustrate how to define simple regular expressions to split the contents of a text string based on the occurrence of one or more consecutive digits.

Although the regular expressions `\d+/\d+/\d+` and `\d\d/\d\d/\d\d\d\d` both match the string `08/13/2014`, the first regular expression matches more patterns than the second regular expression, which is an "exact match" with respect to the number of matching digits that are allowed.

LISTING A.9: MatchPatterns1.py

```
import re

date1 = '02/28/2013'
date2 = 'February 28, 2013'

# Simple matching: \d+ means match one or more digits
if re.match(r'\d+/\d+/\d+', date1):
   print('date1 matches this pattern')
else:
   print('date1 does not match this pattern')

if re.match(r'\d+/\d+/\d+', date2):
   print('date2 matches this pattern')
else:
   print('date2 does not match this pattern')
```

The output from launching Listing A.9 is as follows:

```
date1 matches this pattern
date2 does not match this pattern
```

Reversing Words in Strings

Listing A.10 shows the contents of the Python script `ReverseWords1.py` that illustrate how to reverse a pair of words in a string.

LISTING A.10: ReverseWords1.py

```
import re

str1 = 'one two'
match = re.search('([\w.-]+) ([\w.-]+)', str1)

str2 = match.group(2) + ' ' + match.group(1)

print('str1:',str1)
```

```
print('str2:',str2)
```

The output from Listing A.10 is shown here:

```
str1: one two
str2: two one
```

Now that you understand how to define regular expressions for digits and letters, let's look at some more sophisticated regular expressions.

For example, the following expression matches a string that is any combination of digits, uppercase letters, or lowercase letters (i.e., no special characters):

```
^[a-zA-Z0-9]$
```

Here is the same expression rewritten using character classes:

```
^[\w\W\d]$
```

MODIFYING TEXT STRINGS WITH THE `re` MODULE

The Python `re` module contains several methods for modifying strings. The `split()` method uses a regular expression to "split" a string into a list. The `sub()` method finds all substrings where the regular expression matches, and then replaces them with a different string. The `subn()` method performs the same functionality as `sub()`, and returns the new string and the number of replacements. The following subsections contain examples that illustrate how to use the functions `split()`, `sub()`, and `subn()` in regular expressions.

SPLITTING TEXT STRINGS WITH THE `re.split()` METHOD

Listing A.11 shows the contents of the Python script `RegEx2.py` that illustrate how to define simple regular expressions to split the contents of a text string.

LISTING A.11: RegEx2.py

```
import re

line1 = "abc def"
result1 = re.split(r'[\s]', line1)
```

```
print('result1:',result1)

line2 = "abc1,abc2:abc3;abc4"
result2 = re.split(r'[,:;]', line2)
print('result2:',result2)

line3 = "abc1,abc2:abc3;abc4 123 456"
result3 = re.split(r'[,:;\s]', line3)
print('result3:',result3)
```

Listing A.11 contains three blocks of code, each of which uses the `split()` method in the `re` module to tokenize three different strings. The first regular expression specifies a whitespace, the second regular expression specifies three punctuation characters, and the third regular expression specifies the combination of the first two regular expressions.

The output from launching `RegEx2.py` is

```
result1: ['abc', 'def']
result2: ['abc1', 'abc2', 'abc3', 'abc4']
result3: ['abc1', 'abc2', 'abc3', 'abc4', '123', '456']
```

SPLITTING TEXT STRINGS USING DIGITS AND DELIMITERS

Listing A.12 shows the contents of `SplitCharClass1.py` that illustrate how to use a regular expression consisting of a character class, the "." character, and a whitespace to split the contents of two text strings.

LISTING A.12: SplitCharClass1.py

```
import re

line1 = '1. Section one 2. Section two 3. Section three'
line2 = '11. Section eleven 12. Section twelve 13. Section
                                                    thirteen'

print(re.split(r'\d+\. ', line1)
print(re.split(r'\d+\. ', line2)
```

Listing A.12 contains two text strings that can be split using the same regular expression `\d+\. `. Note that if you use the expression `\d\. `, only the first text string will split correctly. The result of launching Listing A.12 is here:

```
['', 'Section one ', 'Section two ', 'Section three']
['', 'Section eleven ', 'Section twelve ', 'Section
thirteen']
```

SUBSTITUTING TEXT STRINGS WITH THE `re.sub()` METHOD

Earlier in this appendix you saw a preview of using the `sub()` method to remove all the metacharacters in a text string. The following code block illustrates how to use the `re.sub()` method to substitute alphabetic characters in a text string.

```
>>> import re
>>> p = re.compile( '(one|two|three)')
>>> p.sub( 'some', 'one book two books three books')
'some book some books some books'
>>>
>>> p.sub( 'some', 'one book two books three books',
                                                count = 1)
'some book two books three books'
```

The following code block uses the `re.sub()` method to insert a line feed after each alphabetic character in a text string:

```
>>> line = 'abcde'
>>> line2 = re.sub('', '\n', line)

>>> print('line2:',line2)
line2:
a
b
c
d
e
```

MATCHING THE BEGINNING AND THE END OF TEXT STRINGS

Listing A.13 shows the contents of the Python script `RegEx3.py` that illustrate how to find substrings using the `startswith()` function and `endswith()` function.

LISTING A.13: RegEx3.py

```
import re

line2 = "abc1,Abc2:def3;Def4"
result2 = re.split(r'[,:;]', line2)

for w in result2:
  if(w.startswith('Abc')):
     print('Word starts with Abc:',w)
  elif(w.endswith('4')):
     print('Word ends with 4:',w)
  else:
     print('Word:',w)
```

Listing A.13 starts by initializing the string `line2` (with punctuation characters as word delimiters) and the RE `result2` that uses the `split()` function with a comma, colon, and semicolon as "split delimiters" to tokenize the string variable `line2`.

The output after launching Listing A.13 is as follows:

```
Word: abc1
Word starts with Abc: Abc2
Word: def3
Word ends with 4: Def4
```

Listing A.14 shows the contents of the Python script `MatchLines1.py` that illustrate how to find substrings using character classes.

LISTING A.14: MatchLines1.py

```
import re

line1 = "abcdef"
line2 = "123,abc1,abc2,abc3"
line3 = "abc1,abc2,123,456f"
```

```
if re.match("^[A-Za-z]*$", line1):
   print('line1 contains only letters:',line1)

# better than the preceding snippet:
line1[:-1].isalpha()
   print('line1 contains only letters:',line1

if re.match("^[\w]*$", line1):
   print('line1 contains only letters:',line1)

if re.match(r"^[^\W\d_]+$", line1, re.LOCALE):
   print('line1 contains only letters:',line1)
print

if re.match("^[0-9][0-9][0-9]", line2):
   print('line2 starts with 3 digits:',line2)

if re.match("^\d\d\d", line2):
   print('line2 starts with 3 digits:',line2)
print

if re.match(".*[0-9][0-9][0-9][a-z]$", line3):
   print('line3 ends with 3 digits and 1 char:',line3)

if re.match(".*[a-z]$", line3):
   print('line3 ends with 1 char:',line3)
```

Listing A.14 starts by initializing three string variables `line1`, `line2`, and `line3`. The first RE contains an expression that matches any line containing uppercase or lowercase letters (or both):

```
if re.match("^[A-Za-z]*$", line1):
```

The following two snippets also test for the same thing:

```
line1[:-1].isalpha()
```

The preceding snippet starts from the rightmost position of the string and checks if each character is alphabetic.

The next snippet checks if `line1` can be tokenized into words (a word contains only alphabetic characters):

```
if re.match("^[\w]*$", line1):
```

The next portion of Listing A.14 checks if a string contains three consecutive digits:

```
if re.match("^[0-9][0-9][0-9]", line2):
 print('line2 starts with 3 digits:',line2)

if re.match("^\d\d\d", line2):
```

The first snippet uses the pattern `[0-9]` to match a digit, whereas the second snippet uses the expression `\d` to match a digit.

The output from Listing A.14 is as follows:

```
line1 contains only letters: abcdef
line1 contains only letters: abcdef
line1 contains only letters: abcdef
line1 contains only letters: abcdef

line2 starts with 3 digits: 123,abc1,abc2,abc3
line2 starts with 3 digits: 123,abc1,abc2,abc3
```

COMPILATION FLAGS

Compilation flags modify the manner in which regular expressions work. Flags are available in the `re` module as a long name (such as `IGNORECASE`) and a short, one-letter form (such as `I`). The short form is the same as the flags in pattern modifiers in Perl. You can specify multiple flags by using the "|" symbol. For example, `re.I | re.M` sets both the `I` and `M` flags.

COMPOUND REGULAR EXPRESSIONS

Listing A.15 shows the contents of `MatchMixedCase1.py` that illustrate how to use the pipe ("|") symbol to specify two regular expressions in the same `match()` function.

LISTING A.15: MatchMixedCase1.py

```
import re

line1 = "This is a line"
line2 = "That is a line"
```

```
if re.match("^[Tt]his", line1):
   print('line1 starts with This or this:')
   print(line1)
else:
   print('no match')

if re.match("^This|That", line2):
   print('line2 starts with This or That:')
   print(line2)
else:
   print('no match')
```

Listing A.15 starts with two string variables `line1` and `line2`, followed by an `if/else` conditional code block that checks if `line1` starts with the RE `[Tt]his`, which matches the string `This` as well as the string `this`.

The second conditional code block checks if `line2` starts with the string `This` or the string `That`. Notice the "^" metacharacter, which in this context anchors the RE to the beginning of the string. The output from Listing A.15 is here:

```
line1 starts with This or this:
This is a line
line2 starts with This or That:
That is a line
```

COUNTING CHARACTER TYPES IN A STRING

You can use a regular expression to check whether a character is a digit, a letter, or some other type of character. Listing A.16 shows the contents of `CountDigitsAndChars.py` that performs this task.

LISTING A.16: CountDigitsAndChars.py

```
import re

charCount  = 0
digitCount = 0
otherCount = 0

line1 = "A line with numbers: 12 345"
```

```
for ch in line1:
  if(re.match(r'\d', ch)):
     digitCount = digitCount + 1
  elif(re.match(r'\w', ch)):
     charCount = charCount + 1
  else:
     otherCount = otherCount + 1

print('charcount:',charCount)
print('digitcount:',digitCount)
print('othercount:',otherCount)
```

Listing A.16 initializes three numeric counter-related variables, followed by the string variable `line1`. The next part of Listing A.16 contains a `for` loop that processes each character in the string `line1`. The body of the `for` loop contains a conditional code block that checks whether the current character is a digit, a letter, or some other non-alphanumeric character. Each time there is a successful match, the corresponding "counter" variable is incremented.

The output from Listing A.16 is as follows:

```
charcount: 16
digitcount: 5
othercount: 6
```

REGULAR EXPRESSIONS AND GROUPING

You can also "group" subexpressions and even refer to them symbolically. For example, the following expression matches zero or 1 occurrences of 3 consecutive letters or digits:

```
^([a-zA-Z0-9]{3,3})?
```

The following expression matches a telephone number (such as 650-555-1212) in the United States:

```
^\d{3,3}[-]\d{3,3}[-]\d{4,4}
```

The following expression matches a zip code (such as `67827` or `94343-04005`) in the United States:

```
^\d{5,5}([-]\d{5,5})?
```

The following code block partially matches an email address:

```
str = 'john.doe@google.com'
match = re.search(r'\w+@\w+', str)
if match:
 print(match.group() ## 'doe@google')
```

Exercise: Use the preceding code block as a starting point to define a regular expression for email addresses.

SIMPLE STRING MATCHES

Listing A.17 shows the contents of the Python script `RegEx4.py` that illustrate how to define regular expressions that match various text strings.

LISTING A.17: RegEx4.py

```
import re

searchString = "Testing pattern matches"

expr1 = re.compile( r"Test" )
expr2 = re.compile( r"^Test" )
expr3 = re.compile( r"Test$" )
expr4 = re.compile( r"\b\w*es\b" )
expr5 = re.compile( r"t[aeiou]", re.I )

if expr1.search( searchString ):
   print('"Test" was found.')

if expr2.match( searchString ):
   print('"Test" was found at the beginning of the line.')

if expr3.match( searchString ):
   print('"Test" was found at the end of the line.')

result = expr4.findall( searchString )

if result:
   print('There are %d words(s) ending in "es":' % \)
   ( len( result ) ),
```

```
 for item in result:
   print(" " + item,)

print

result = expr5.findall( searchString )
if result:
   print('The letter t, followed by a vowel, occurs %d
                                                 times:' % \)
   ( len( result ) ),

 for item in result:
   print(" "+item,)

print
```

Listing A.17 starts with the variable `searchString` that specifies a text string, followed by the REs `expr1`, `expr2`, and `expr3`. The RE `expr1` matches the string `Test` that occurs anywhere in `searchString`, whereas `expr2` matches `Test` if it occurs at the beginning of `searchString`, and `expr3` matches `Test` if it occurs at the end of `searchString`. The RE `expr` matches words that end in the letters `es`, and the RE `expr5` matches the letter `t` followed by a vowel.

The output from Listing A.17 is here:

```
"Test" was found.
"Test" was found at the beginning of the line.
There are 1 words(s) ending in "es": matches
      The letter t, followed by a vowel, occurs
                                        3 times: Te ti te
```

PANDAS AND REGULAR EXPRESSIONS

This section is optional because the code snippets require an understanding of the Python-based Pandas library. If you are not interested in learning about Pandas, you can skip this section with no loss of continuity. Alternatively, you can search online for various tutorials regarding Pandas, after which you can return to this portion of the appendix.

Listing A.18 shows the contents `pandas_regexs.py` that illustrate how to extract data from a Pandas data frame using regular expressions.

LISTING A.18: pandas_regexs.py

```
import pandas as pd

schedule = ["Monday: Prepare lunch at 12:30pm for VIPs",
            "Tuesday: Yoga class from 10:00am to 11:00am",
            "Wednesday: PTA meeting at library at 3pm",
            "Thursday: Happy hour at 5:45 at Julie's
            house.",
            "Friday: Prepare pizza dough for lunch at
            12:30pm.",
            "Saturday: Early shopping for the week at
            8:30am.",
            "Sunday: Neighborhood bbq block party at
            2:00pm."]

# create a Pandas dataframe:
df = pd.DataFrame(schedule, columns = ['dow_of_week'])

# convert to lowercase:
df = df.applymap(lambda s:s.lower() if type(s) == str else s)
print("df:")
print(df)
print()

# character count for each string in df['dow_of_week']:
print("string lengths:")
print(df['dow_of_week'].str.len())
print()

# the number of tokens for each string in df['dow_of_week']
print("number of tokens in each string in df['dow_of_week']:")
print(df['dow_of_week'].str.split().str.len())
print()

# the number of occurrences of digits:
print("number of digits:")
print(df['dow_of_week'].str.count(r'\d'))
print()
```

```
# display all occurrences of digits:
print("show all digits:")
print(df['dow_of_week'].str.findall(r'\d'))
print()

# display hour and minute values:
print("display (hour, minute) pairs:")
print(df['dow_of_week'].str.findall(r'(\d?\d)\d\')'))
print()

# create new columns from hour:minute value:

print ("hour and minute column:")
print(df['dow_of_week'].str.extract'r'(\d?\d):(\d\')'))
print()
```

Listing A.18 initializes the variable `schedule` with a set of strings, each of which specifies a daily to-do item for an entire week. The format for each to-do item is of the form `day:task`, where is a day of the week and task is a string that specifies what needs to be done on that particular day. Next, the data frame `df1` is initialized with the contents of `schedule`, followed by an example of defining a lambda expression that converts string-based values to lower case, as shown here:

```
df = df.applymap(lambda s:s.lower() if type(s) == str
                                                 else s)
```

The preceding code snippet is useful because you do not need to specify individual columns of a data frame. The code ignores any non-string values (such as integers and floating point values).

The next pair of code blocks involve various operations using the methods `applymap()`, `split()`, and `len()`. The next code block displays the number of digits in each to-do item by means of the regular expression in the following code snippet:

```
print(df['dow_of_week'].str.count(r'\d'))
```

The next code block displays the actual digits (instead of the number of digits) in each to-do item by means of the regular expression in the following code snippet:

```
print(df['dow_of_week'].str.findall(r'\d'))
```

The final code block displays the strings of the form `hour:minutes` by means of the regular expression in the following code snippet:

```
print(df['dow_of_week'].str.findall(r'(\d?\d):(\d\d)'))
```

As mentioned in the beginning of this section, you can learn more about regular expressions by reading one of the appendices of this book. Launch the code in Listing A.18 to see the following output:

```
=> df:
                                                dow_of_week
0              monday: prepare lunch at 12:30pm for vips
1            tuesday: yoga class from 10:00am to 11:00am
2               wednesday: pta meeting at library at 3pm
3         thursday: happy hour at 5:45 at julie's house.
4      friday: prepare pizza dough for lunch at 12:30pm.
5       saturday: early shopping for the week at 8:30am.
6        sunday: neighborhood bbq block party at 2:00pm.

=> string lengths:
0    41
1    43
2    40
3    46
4    49
5    48
6    47
Name: dow_of_week, dtype: int64

=> number of tokens in each string in df['dow_of_week']:
0    7
1    7
2    7
3    8
4    8
5    8
6    7
Name: dow_of_week, dtype: int64
```

```
=> number of digits:
0    4
1    8
2    1
3    3
4    4
5    3
6    3
Name: dow_of_week, dtype: int64

=> show all digits:
0                [1, 2, 3, 0]
1    [1, 0, 0, 0, 1, 1, 0, 0]
2                         [3]
3                   [5, 4, 5]
4                [1, 2, 3, 0]
5                   [8, 3, 0]
6                   [2, 0, 0]
Name: dow_of_week, dtype: object

=> display (hour, minute) pairs:
0              [(12, 30)]
1    [(10, 00), (11, 00)]
2                      []
3               [(5, 45)]
4              [(12, 30)]
5               [(8, 30)]
6               [(2, 00)]
Name: dow_of_week, dtype: object

=> hour and minute columns:
     0    1
0   12   30
```

```
1   10   00
2  NaN  NaN
3    5   45
4   12   30
5    8   30
6    2   00
```

SUMMARY

This appendix showed you how to create various types of regular expressions. First, you learned how to define primitive regular expressions using sequences of digits, lowercase letters, and uppercase letters. Next, you learned how to use character classes, which are more convenient and simpler expressions that can perform the same functionality. You also learned how to use the Python `re` library to compile regular expressions and then use them to see if they match substrings of text strings.

EXERCISES

- Exercise 1: Given a text string, find the list of words (if any) that start or end with a vowel, and treat upper- and lowercase vowels as distinct letters. Display this list of words in alphabetical order and also in descending order based on their frequency.
- Exercise 2: Given a text string, find the list of words (if any) that contain lowercase vowels or digits or both, but no uppercase letters. Display this list of words in alphabetical order and also in descending order based on their frequency.
- Exercise 3: There is a spelling rule in English specifying that "the letter i is before e, except after c," which means that "receive" is correct but "recieve" is incorrect. Write a Python script that checks for incorrectly spelled words in a text string.
- Exercise 4: Subject pronouns cannot follow a preposition in the English language. Thus, "between you and me" and "for you and me" are correct, whereas "between you and I" and "for you and I" are incorrect. Write a Python script that checks for incorrect grammar in a text string, and search

for the prepositions "between," "for," and "with." In addition, search for the subject pronouns "I," "you," "he," and "she." Modify and display the text with the correct grammar usage.

- Exercise 5: Find the words in a text string whose length is at most 4 and then print all the substrings of those characters. For example, if a text string contains the word "text", then print the strings "t," "te," "tex," and "text."

APPENDIX B

INTRODUCTION 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), 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. We also discuss 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. We also include a section on 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's Theorem.

WHAT IS A PROBABILITY?

If you have ever performed a science experiment in one of your classes, you might remember that 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.

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.

One other concept is called *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 dollar. 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 earnings after 100 coin tosses. However, 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 earnings `E` after 100 coin tosses is

```
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 nonnegative 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 shown here:

```
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, and 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 * 7/6 = 116.67
```

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 is also countably infinite, but the set of irrational numbers and also the set of real numbers are both uncountably infinite (proofs are available online). 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.

The frequency of traffic at a given point in a road follows a Poisson distribution (which is not symmetric). Interestingly, 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 to 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 starting 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. As you can see, the mean value is not necessarily one of the values in the set.

Keep in mind that 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 *multimodal*.

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 ranges 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
```

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 United States, 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 sample 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 very 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's been mathematically proven to be true; that is, it's not an empirical or heuristic-based result. An extensive description regarding Chebyshev's inequality (including some advanced mathematical explanations) is here:

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 are 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 so-called "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 United States 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 U.S. 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 provides 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 ten bedrooms and one bathroom, or a house that has ten bathrooms and one 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).

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 relates to processes and statistics, whereas statistical inference refers to the process by which inferences are made 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 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 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." These terms are also 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 = Iy - 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 1 and the worst 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 nonnegative 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, . . ., 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 the Gini Impurity?

The Gini impurity is defined as follows, where {p1,p2,…,pn} is a probability distribution:

```
Gini = 1 - [p1*p1 + p2*p2 + . . . + pn*pn]
     = 1 - SUM pi*pi (for all i, where 1<=i<=n)
```

Since each pi is between 0 and 1, then pi*pi <= pi, which means that

```
1 = p1 + p2 + . . . + pn
  >= p1*p1 + p2*p2 + . . . + pn*pn
   = Gini impurity
```

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 <= 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) is defined via the following formula:

```
H = (-1)*[p1*log p1 + p2 * log p2 + . . . + pn * log pn]
  = (-1)* SUM [pi * log(pi)] (for all i, where 1<=i<=n)
```

Calculating Gini Impurity and Entropy Values

For our first example, suppose that we have two 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:

```
Gini = 1 - [p1*p1 + p2*p2]
     = 1 - [64/100 + 04/100]
     = 1 - 68/100
```

```
     = 32/100
     = 0.32
```

We can also calculate the entropy for this example as follows:

```
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
```

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:

```
Gini = 1 - [p1*p1 + p2*p2 + p3*p3]
     = 1 - [25/100 + 9/100 + 04/100]
     = 1 - 38/100
     = 62/100
     = 0.62
```

We can also calculate the entropy for this example as follows:

```
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
```

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 a Gini impurity equal to 0, and 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 p1
        = (-1) * (10/10) * log(10/10)
        = (-1)*1*0 = 0
```

Multidimensional 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 multidimensional 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`:

```
PERP = b^S
```

If you compare the formula for entropy with the formula for `S`, you can see that the formulas are 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. Specifically, suppose that users arrive at a Webpage by clicking on a previous page, which was also reached by clicking on yet another Webpage. Then, on the final Webpage, users click on an ad. How much credit is given to the first and second Webpages 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)
```

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 earlier in this 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?

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) that is described online:

- *https://en.wikipedia.org/wiki/Lebesgue_measure*
- *https://en.wikipedia.org/wiki/Borel_set*

In addition to KL divergence, there is also JS divergence, also called Jenson-Shannon divergence, which was developed by Johan Jensen and Claude Shannon (who defined the formula for entropy). JS divergence is based on KL divergence, but it has some differences. JS divergence is symmetric and a true metric, whereas 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. For more information about any of these terms, perform an online search to find numerous online tutorials that provide more detailed information.

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

```
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 already in this 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 | <= 6x2 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`. Subtract `x_bar` from column `x` and subtract `y_bar` from column `y` to obtain matrix `B`:

```
B = | -2.5 -1 | <= 6x2 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:

```
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`:

```
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 is 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

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 nonzero vector `x'` is an eigenvector of the matrix `C` if there is a nonzero 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 2x1 vector)
```

Since `x'` is nonzero, 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

```
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` to 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 nonzero value for `x2`, so let's set `x2` = 1 (any nonzero value is acceptable), 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:

```
[-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 into 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 nonzero).

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 one and the coefficients of y in column two, followed by the 2×2 identity matrix, and finally 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 middle matrix is the inverse of the original matrix, and the rightmost 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 later on 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 preprocessing 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. Here is a key point about PCA:

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. 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 leftmost eigenvector has the largest eigenvalue, the next eigenvector has the second-largest eigenvalue, and continuing on in this fashion until we reach the rightmost 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 holds:

```
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, and correlation matrix, as well as the matrices `D` and `P` 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 leftmost 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` hyperparameter in the PCA class of Sklearn (a very powerful Python-based machine learning library).

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 the 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 * SDt
```

Although PCA is a useful technique for dimensionality reduction, it does have some limitations:

- 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 nonlinear 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 so-called "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 that 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:

```
                cov(X,Y)
rho(X,Y) = -------------
             std(X)*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 who 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 the unique words in a sentence and is unaffected by duplicates, whereas the 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 associates 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 constant 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 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 "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 divergence (which is based on the KL Divergence) is a true metric, which enables us to make 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's 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 the dynamic analysis of sequential data.

Bayes's 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 Equation (#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.

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's Theorem?

Bayes's 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's 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, after which you learned about Bayes's theorem.

INDEX

S

T