

Data Science Algorithms in a Week

Second Edition

Top 7 algorithms for scientific computing, data analysis, and machine learning



Packt

www.packt.com

Dávid Natingga

Data Science Algorithms in a Week

Second Edition

Top 7 algorithms for scientific computing, data analysis,
and machine learning

Dávid Natingga



BIRMINGHAM - MUMBAI

Data Science Algorithms in a Week

Second Edition

Copyright © 2018 Packt Publishing

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

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author(s), nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Commissioning Editor: Amey Varangaonkar
Acquisition Editor: Joshua Nadar
Content Development Editor: Ronnel Mathew
Technical Editor: Sneha Hanchate
Copy Editor: Safis Editing
Project Coordinator: Nidhi Joshi
Proofreader: Safis Editing
Indexer: Rekha Nair
Graphics: Tom Scaria
Production Coordinator: Aparna Bhagat

First published: August 2017
Second edition: October 2018

Production reference: 1311018

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78980-607-6

www.packtpub.com



mapt.io

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

Packt.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the author

Dávid Natingga graduated with a master's in engineering in 2014 from Imperial College London, specializing in artificial intelligence. In 2011, he worked at Infosys Labs in Bangalore, India, undertaking research into the optimization of machine learning algorithms. In 2012 and 2013, while at Palantir Technologies in USA, he developed algorithms for big data. In 2014, while working as a data scientist at Pact Coffee, London, he created an algorithm suggesting products based on the taste preferences of customers and the structures of the coffees. In order to use pure mathematics to advance the field of AI, he is a PhD candidate in Computability Theory at the University of Leeds, UK. In 2016, he spent 8 months at Japan's Advanced Institute of Science and Technology as a research visitor.

About the reviewers

Surendra Pepakayala is a hands-on, seasoned technology professional with over 20 years of experience in the US and India. He has built enterprise software products at startups and multinational companies and has built and sold a technology business after five years in operation. He currently consults for small to medium businesses, helping them leverage cloud, data science, machine learning, AI, and cutting-edge technologies to gain an advantage over their competition. In addition to being an advisory board member for a couple of startups in the technology space, he holds numerous coveted certifications such as TOGAF, CRISC, and CGEIT.

Jen Stirrup is a data strategist and technologist, Microsoft Most Valuable Professional (MVP) and Microsoft Regional Director, tech community advocate, public speaker, blogger, published author, and keynote speaker. She is the founder of a boutique consultancy based in the UK, Data Relish, which focuses on delivering successful business intelligence and artificial intelligence solutions that add real value to customers worldwide. She has featured on the BBC as a guest expert on topics related to data.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface	1
Chapter 1: Classification Using K-Nearest Neighbors	6
Mary and her temperature preferences	6
Implementation of the k-nearest neighbors algorithm	10
Map of Italy example – choosing the value of k	15
Analysis	16
House ownership – data rescaling	20
Analysis	21
Text classification – using non-Euclidean distances	22
Analysis	23
Text classification – k-NN in higher dimensions	25
Analysis	26
Summary	27
Problems	27
Mary and her temperature preference problems	28
Map of Italy – choosing the value of k	28
House ownership	28
Analysis	29
Chapter 2: Naive Bayes	32
Medical tests – basic application of Bayes' theorem	33
Analysis	33
Bayes' theorem and its extension	34
Bayes' theorem	34
Proof	34
Extended Bayes' theorem	35
Proof	35
Playing chess – independent events	36
Analysis	37
Implementation of a Naive Bayes classifier	38
Playing chess – dependent events	41
Analysis	42
Gender classification – Bayes for continuous random variables	44
Analysis	44
Summary	46
Problems	47
Analysis	49
Chapter 3: Decision Trees	55

Swim preference – representing data using a decision tree	56
Information theory	57
Information entropy	57
Coin flipping	58
Definition of information entropy	58
Information gain	59
Swim preference – information gain calculation	60
ID3 algorithm – decision tree construction	61
Swim preference – decision tree construction by the ID3 algorithm	61
Implementation	63
Classifying with a decision tree	68
Classifying a data sample with the swimming preference decision tree	69
Playing chess – analysis with a decision tree	70
Analysis	70
Classification	73
Going shopping – dealing with data inconsistencies	73
Analysis	74
Summary	75
Problems	75
Analysis	76
Chapter 4: Random Forests	79
Introduction to the random forest algorithm	80
Overview of random forest construction	80
Swim preference – analysis involving a random forest	81
Analysis	81
Random forest construction	82
Construction of random decision tree number 0	82
Construction of random decision tree number 1	84
Constructed random forest	87
Classification using random forest	87
Implementation of the random forest algorithm	88
Playing chess example	91
Analysis	92
Random forest construction	92
Classification	98
Going shopping – overcoming data inconsistencies with randomness and measuring the level of confidence	99
Analysis	100
Summary	101
Problems	102
Analysis	102
Chapter 5: Clustering into K Clusters	106
Household incomes – clustering into k clusters	106
K-means clustering algorithm	107

Picking the initial k-centroids	108
Computing a centroid of a given cluster	108
Using the k-means clustering algorithm on the household income example	108
Gender classification – clustering to classify	110
Analysis	111
Implementation of the k-means clustering algorithm	113
Input data from gender classification	116
Program output for gender classification data	116
House ownership – choosing the number of clusters	117
Analysis	118
Document clustering – understanding the number of k clusters in a semantic context	124
Analysis	125
Summary	131
Problems	131
Analysis	133
Chapter 6: Regression	140
Fahrenheit and Celsius conversion – linear regression on perfect data	141
Analysis from first principles	141
Least squares method for linear regression	142
Analysis using the least squares method in Python	143
Visualization	144
Weight prediction from height – linear regression on real-world data	145
Analysis	145
Gradient descent algorithm and its implementation	146
Gradient descent algorithm	146
Implementation	147
Visualization – comparison of the least squares method and the gradient descent algorithm	150
Flight time duration prediction based on distance	151
Analysis	151
Ballistic flight analysis – non-linear model	152
Analysis	153
Analysis by using the least squares method in Python	153
Summary	154
Problems	155
Analysis	156
Chapter 7: Time Series Analysis	160
Business profits – analyzing trends	161
Analysis	161
Analyzing trends using the least squares method in Python	161
Visualization	162
Conclusion	163

Electronics shop's sales – analyzing seasonality	163
Analysis	164
Analyzing trends using the least squares method in Python	165
Visualization	167
Analyzing seasonality	167
Conclusion	171
Summary	172
Problems	172
Analysis	173
Appendix A: Python Reference	177
Introduction	177
Python Hello World example	177
Comments	177
Data types	178
int	178
float	179
String	179
Tuple	180
List	180
Set	181
Dictionary	181
Flow control	182
Conditionals	182
For loop	183
For loop on range	183
For loop on list	183
Break and continue	184
Functions	185
Input and output	185
Program arguments	185
Reading and writing a file	186
Appendix B: Statistics	187
Basic concepts	187
Bayesian inference	188
Distributions	188
Normal distribution	188
Cross-validation	189
K-fold cross-validation	190
A/B testing	190
Appendix C: Glossary of Algorithms and Methods in Data Science	191
Other Books You May Enjoy	193
Index	196

Preface

Data science is a discipline at the intersection of machine learning, statistics, and data mining with the objective of gaining new knowledge from existing data by means of algorithmic and statistical analysis. In this book, you will learn the seven most important ways in data science of analyzing the data. Each chapter first explains its algorithm or analysis as a simple concept, supported by a trivial example. Further examples and exercises are used to build and expand your knowledge of a particular type of analysis.

Who this book is for

This book is for aspiring data science professionals who are familiar with Python and have a background of sorts in statistics. Developers who are currently implementing one or two data science algorithms and who now want to learn more to expand their skillset will find this book quite useful.

What this book covers

Chapter 1, *Classification Using K-Nearest Neighbors*, classifies a data item based on the most similar k items.

Chapter 2, *Naïve Bayes*, delves into Bayes' Theorem with a view to computing the probability a data item belonging to a certain class.

Chapter 3, *Decision Trees*, organizes your decision criteria into the branches of a tree, and uses a decision tree to classify a data item into one of the classes at the leaf node.

Chapter 4, *Random Forests*, classifies a data item with an ensemble of decision trees to improve the accuracy of the algorithm by reducing the negative impact of the bias.

Chapter 5, *Clustering into K Clusters*, divides your data into k clusters to discover the patterns and similarities between the data items and goes into how to exploit these patterns to classify new data.

Chapter 6, *Regression*, models phenomena in your data by using a function that can predict the values of the unknown data in a simple way.

Chapter 7, *Time-Series Analysis*, unveils the trends and repeating patterns in time-dependent data to predict the future of the stock market, Bitcoin prices, and other time events.

Appendix A, *Python Reference*, is a reference of the basic Python language constructs, commands, and functions used throughout the book.

Appendix B, *Statistics*, provides a summary of the statistical methods and tools that are useful to a data scientist.

Appendix C, *Glossary of Algorithms and Methods in Data Science*, provides a glossary of some of the most important and powerful algorithms and methods from the fields of data science and machine learning.

To get the most out of this book

To get the most out of this book, you require, first and foremost, an active attitude to think of the problems—a lot of new content is presented in the exercises at the end of the chapter in the section entitled *Problems*. You also then need to be able to run Python programs on the operating system of your choice. The author ran the programs on the Linux operating system using the command line

Download the example code files

You can download the example code files for this book from your account at www.packt.com. If you purchased this book elsewhere, you can visit www.packt.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at www.packt.com.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub

at <https://github.com/PacktPublishing/Data-Science-Algorithms-in-a-Week-Second-Edition>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

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

http://www.packtpub.com/sites/default/files/downloads/9781789806076_ColorImages.pdf.

Conventions used

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

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Mount the downloaded `WebStorm-10*.dmg` disk image file as another disk in your system."

A block of code is set as follows:

```
def dic_key_count(dic, key):
    if key is None:
        return 0
    if dic.get(key, None) is None:
        return 0
    else:
        return int(dic[key])
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
def construct_general_tree(verbose, heading, complete_data,
    enquired_column, m):
    available_columns = []
    for col in range(0, len(heading)):
        if col != enquired_column:
```

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

```
$ python naive_bayes.py chess.csv
```

Bold: Indicates a new term, an important word, or words that you see on screen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Select **System info** from the **Administration** panel."



Warnings or important notes appear like this.



Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

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

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packt.com/submit-errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

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

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

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packt.com.

1

Classification Using K-Nearest Neighbors

A nearest neighbor algorithm classifies a data instance based on its neighbors. The class of a data instance determined by the k -nearest neighbors algorithm is the class with the highest representation among the k -closest neighbors.

In this chapter, we will cover the following topics:

- How to implement the basics of the k -NN algorithm using the example of Mary and her temperature preferences
- How to choose a correct k value so that the algorithm can perform correctly and with the highest degree of accuracy using the example of a map of Italy
- How to rescale values and prepare them for the k -NN algorithm using the example of house preferences
- How to choose a good metric to measure distances between data points
- How to eliminate irrelevant dimensions in higher-dimensional space to ensure that the algorithm performs accurately using the text classification example

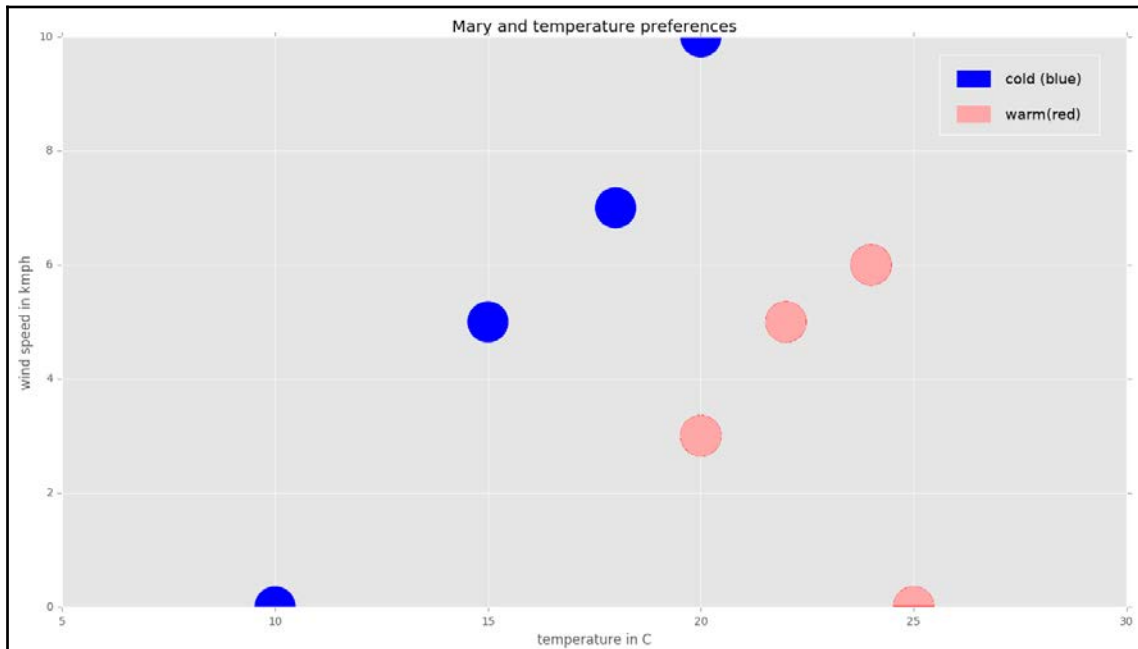
Mary and her temperature preferences

As an example, if we know that our friend, Mary, feels cold when it is 10°C , but warm when it is 25°C , then in a room where it is 22°C , the nearest neighbor algorithm would guess that our friend would feel warm, because 22 is closer to 25 than to 10.

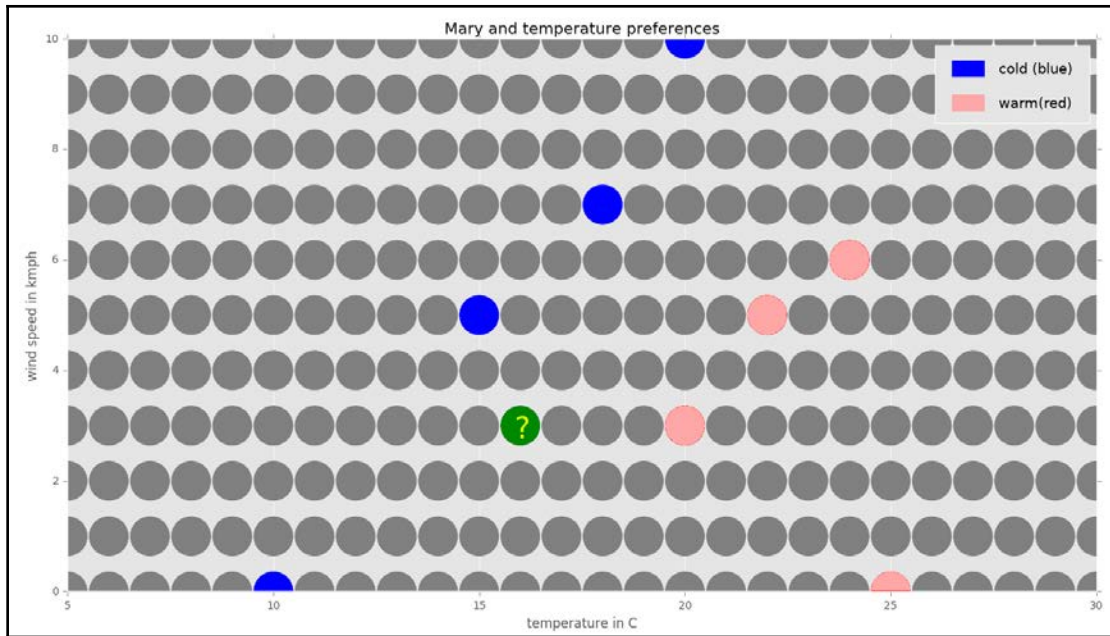
Suppose that we would like to know when Mary feels warm and when she feels cold, as in the previous example, but in addition, wind speed data is also available when Mary is asked whether she feels warm or cold:

Temperature in °C	Wind speed in km/h	Mary's perception
10	0	Cold
25	0	Warm
15	5	Cold
20	3	Warm
18	7	Cold
20	10	Cold
22	5	Warm
24	6	Warm

We could represent the data in a graph, as follows:



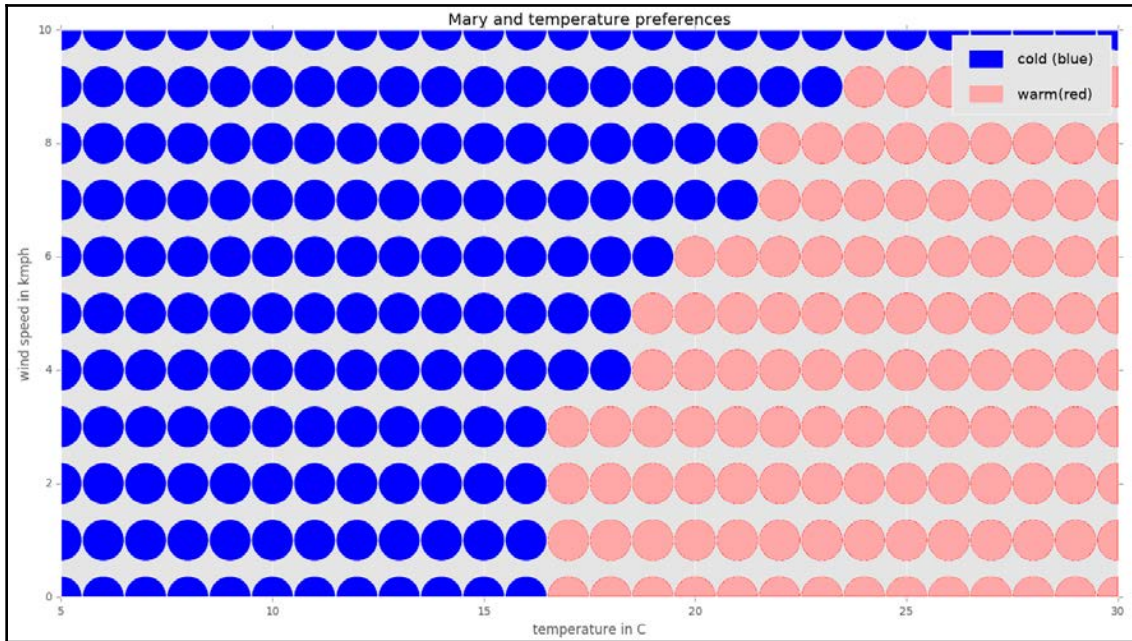
Now, suppose we would like to find out how Mary feels when the temperature is 16°C with a wind speed of 3 km/h by using the 1-NN algorithm:



For simplicity, we will use a Manhattan metric to measure the distance between the neighbors on the grid. The Manhattan distance d_{Man} of the neighbor $N_1=(x_1, y_1)$ from the neighbor $N_2=(x_2, y_2)$ is defined as $d_{Man}=|x_1-x_2|+|y_1-y_2|$.

[9]

By applying this procedure to every data point, we can complete the graph, as follows:



Note that, sometimes, a data point might be the same distance away from two known classes: for example, 20°C and 6 km/h. In such situations, we could prefer one class over the other, or ignore these boundary cases. The actual result depends on the specific implementation of an algorithm.

Implementation of the k-nearest neighbors algorithm

Now, we will implement the k-NN algorithm in Python to find Mary's temperature preference. At the end of this section, we will also implement the visualization of the data produced in the previous section, that is, *Mary and her temperature preferences*. The full, compilable code, with the input files, can be found in the source code provided with this book. The most important parts have been extracted and presented here:

```
# source_code/1/mary_and_temperature_preferences/knn_to_data.py
# Applies the knn algorithm to the input data.
# The input text file is assumed to be of the format with one line per
# every data entry consisting of the temperature in degrees Celsius,
```

```

# wind speed and then the classification cold/warm.

import sys
sys.path.append('.')
sys.path.append('../common')
import knn # noqa
import common # noqa

# Program start
# E.g. "mary_and_temperature_preferences.data"
input_file = sys.argv[1]
# E.g. "mary_and_temperature_preferences_completed.data"
output_file = sys.argv[2]
k = int(sys.argv[3])
x_from = int(sys.argv[4])
x_to = int(sys.argv[5])
y_from = int(sys.argv[6])
y_to = int(sys.argv[7])

data = common.load_3row_data_to_dic(input_file)
new_data = knn.knn_to_2d_data(data, x_from, x_to, y_from, y_to, k)
common.save_3row_data_from_dic(output_file, new_data)

# source_code/common/common.py
# ***Library with common routines and functions***
def dic_inc(dic, key):
    if key is None:
        pass
    if dic.get(key, None) is None:
        dic[key] = 1
    else:
        dic[key] = dic[key] + 1

# source_code/1/knn.py
# ***Library implementing knn algorithm***

def info_reset(info):
    info['nbhd_count'] = 0
    info['class_count'] = {}

# Find the class of a neighbor with the coordinates x,y.
# If the class is known count that neighbor.
def info_add(info, data, x, y):
    group = data.get((x, y), None)
    common.dic_inc(info['class_count'], group)
    info['nbhd_count'] += int(group is not None)

# Apply knn algorithm to the 2d data using the k-nearest neighbors with

```

```

# the Manhattan distance.
# The dictionary data comes in the form with keys being 2d coordinates
# and the values being the class.
# x,y are integer coordinates for the 2d data with the range
# [x_from,x_to] x [y_from,y_to].
def knn_to_2d_data(data, x_from, x_to, y_from, y_to, k):
    new_data = {}
    info = {}
    # Go through every point in an integer coordinate system.
    for y in range(y_from, y_to + 1):
        for x in range(x_from, x_to + 1):
            info_reset(info)
            # Count the number of neighbors for each class group for
            # every distance dist starting at 0 until at least k
            # neighbors with known classes are found.
            for dist in range(0, x_to - x_from + y_to - y_from):
                # Count all neighbors that are distanced dist from
                # the point [x,y].
                if dist == 0:
                    info_add(info, data, x, y)
                else:
                    for i in range(0, dist + 1):
                        info_add(info, data, x - i, y + dist - i)
                        info_add(info, data, x + dist - i, y - i)
                    for i in range(1, dist):
                        info_add(info, data, x + i, y + dist - i)
                        info_add(info, data, x - dist + i, y - i)
                # There could be more than k-closest neighbors if the
                # distance of more of them is the same from the point
                # [x,y]. But immediately when we have at least k of
                # them, we break from the loop.
                if info['nbhd_count'] >= k:
                    break
            class_max_count = None
            # Choose the class with the highest count of the neighbors
            # from among the k-closest neighbors.
            for group, count in info['class_count'].items():
                if group is not None and (class_max_count is None or
                    count > info['class_count'][class_max_count]):
                    class_max_count = group
            new_data[x, y] = class_max_count
    return new_data

```

Input:

The preceding program will use the following file as the source of the input data. The file contains the table with the known data about Mary's temperature preferences:

```
# source_code/1/mary_and_temperature_preferences/  
marry_and_temperature_preferences.data  
10 0 cold  
25 0 warm  
15 5 cold  
20 3 warm  
18 7 cold  
20 10 cold  
22 5 warm  
24 6 warm
```

Output:

We run the preceding implementation on the `marry_and_temperature_preferences.data` input file by using the k-NN algorithm for $k=1$ neighbors. The algorithm classifies all the points with integer coordinates in the rectangle with a size of $(30-5=25)$ by $(10-0=10)$, hence, with a size of $(25+1) * (10+1) = 286$ integer points (adding one to count points on boundaries). Using the `wc` command, we find out that the output file contains exactly 286 lines—one data item per point. Using the `head` command, we display the first 10 lines from the output file:

```
$ python knn_to_data.py marry_and_temperature_preferences.data  
marry_and_temperature_preferences_completed.data 1 5 30 0 10  
  
$ wc -l marry_and_temperature_preferences_completed.data  
286 marry_and_temperature_preferences_completed.data  
  
$ head -10 marry_and_temperature_preferences_completed.data  
7 3 cold  
6 9 cold  
12 1 cold  
16 6 cold  
16 9 cold  
14 4 cold  
13 4 cold  
19 4 warm  
18 4 cold  
15 1 cold
```

Visualization:

For the visualization depicted earlier in this chapter, the `matplotlib` library was used. A data file is loaded, and then displayed in a scatter diagram:

```
# source_code/common/common.py
# returns a dictionary of 3 lists: 1st with x coordinates,
# 2nd with y coordinates, 3rd with colors with numeric values
def get_x_y_colors(data):
    dic = {}
    dic['x'] = [0] * len(data)
    dic['y'] = [0] * len(data)
    dic['colors'] = [0] * len(data)
    for i in range(0, len(data)):
        dic['x'][i] = data[i][0]
        dic['y'][i] = data[i][1]
        dic['colors'][i] = data[i][2]
    return dic

# source_code/1/mary_and_temperature_preferences/
# mary_and_temperature_preferences_draw_graph.py
import sys
sys.path.append('../common') # noqa
import common
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
import matplotlib
matplotlib.style.use('ggplot')

data_file_name = 'mary_and_temperature_preferences_completed.data'
temp_from = 5
temp_to = 30
wind_from = 0
wind_to = 10

data = np.loadtxt(open(data_file_name, 'r'),
                  dtype={'names': ('temperature', 'wind', 'perception'),
                        'formats': ('i4', 'i4', 'S4')})

# Convert the classes to the colors to be displayed in a diagram.
for i in range(0, len(data)):
    if data[i][2] == 'cold':
        data[i][2] = 'blue'
    elif data[i][2] == 'warm':
        data[i][2] = 'red'
    else:
        data[i][2] = 'gray'
```

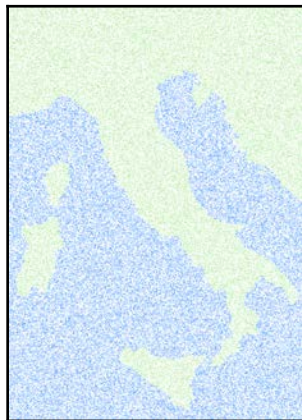
```
# Convert the array into the format ready for drawing functions.
data_processed = common.get_x_y_colors(data)

# Draw the graph.
plt.title('Mary and temperature preferences')
plt.xlabel('temperature in C')
plt.ylabel('wind speed in kmph')
plt.axis([temp_from, temp_to, wind_from, wind_to])
# Add legends to the graph.
blue_patch = mpatches.Patch(color='blue', label='cold')
red_patch = mpatches.Patch(color='red', label='warm')
plt.legend(handles=[blue_patch, red_patch])
plt.scatter(data_processed['x'], data_processed['y'],
            c=data_processed['colors'], s=[1400] * len(data))
plt.show()
```

Map of Italy example – choosing the value of **k**

In our data, we are given some points (about 1 percent) from the map of Italy and its surroundings. The blue points represent water, and the green points represent land; the white points are unknown. From the partial information that we have been given, we would like to predict whether there is water or land in the white areas.

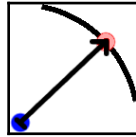
Drawing only 1% of the map data in the picture would make it almost invisible. If we were given about 33 times more data from the map of Italy and its surroundings, and drew it in the picture instead, it would look as follows:



Analysis

For this problem, we will use the k-NN algorithm— k here means that we will look at k -closest neighbors. Given a white point, it will be classified as an area of water if the majority of its k -closest neighbors are in an area of water and classified as land if the majority of its k -closest neighbors are an area of land. We will use the Euclidean metric for the distance: given two points, $X=[x_0, x_1]$ and $Y=[y_0, y_1]$, their Euclidean distance is defined as $d_{Euclidean} = \sqrt{(x_0 - y_0)^2 + (x_1 - y_1)^2}$.

The Euclidean distance is the most common metric. Given two points on a piece of paper, their Euclidean distance is just the length between the two points, as measured by a ruler, as shown in the following diagram:



To apply the k-NN algorithm to an incomplete map, we have to choose the value of k . Since the resulting class of a point is the class of the majority of the k -closest neighbors of that point, k should be odd. Let's apply this algorithm to the values of $k=1, 3, 5, 7, 9$.

Applying this algorithm to every white point on the incomplete map will result in the following complete maps:



$k=1$



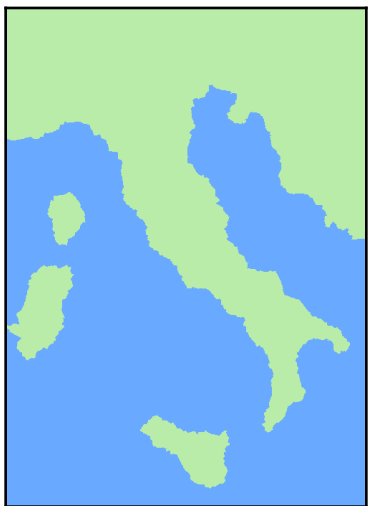
k=3



k=5



k=7



k=9

As you may have noticed, the highest value of k results in a complete map with smoother boundaries. An actual complete map of Italy is shown here:



We can use this real, complete map to calculate the percentage of incorrectly classified points for the various values of k to determine the accuracy of the k -NN algorithm for different values of k :

k	Percentage of incorrectly classified points
1	2.97
3	3.24
5	3.29
7	3.40
9	3.57

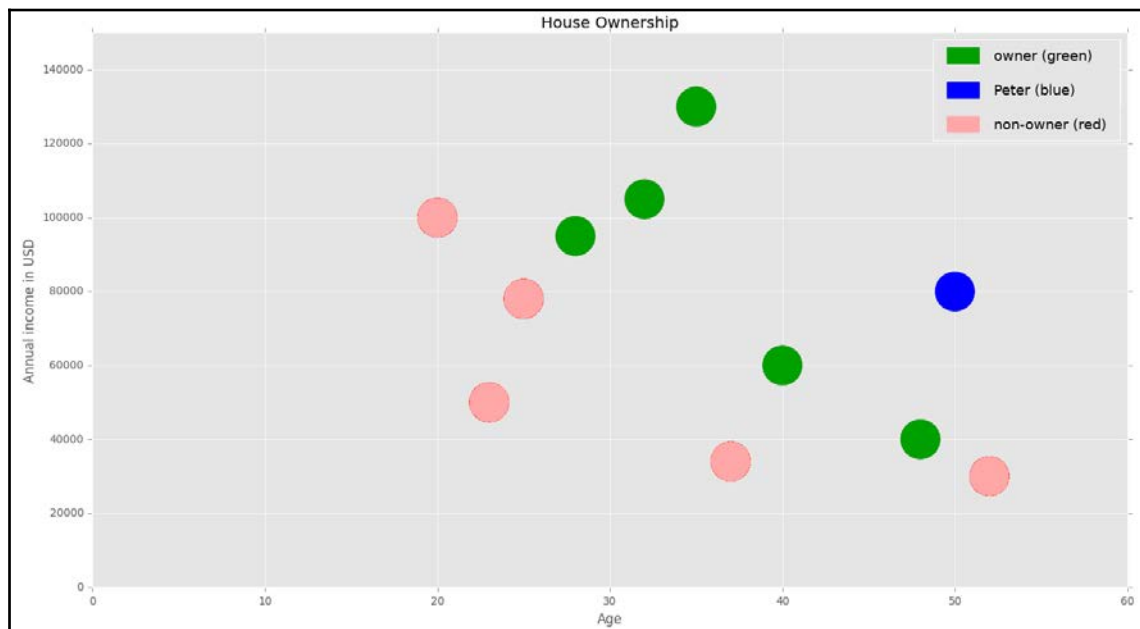
Thus, for this particular type of classification problem, the k -NN algorithm achieves the highest accuracy (least error rate) for $k=1$.

However, in real life, the problem is that we wouldn't usually have complete data or a solution. In such scenarios, we need to choose a value of k that is appropriate to the data that is partially available. For this, consult *Problem 4* at the end of this chapter.

House ownership – data rescaling

For each person, we are given their age, yearly income, and whether or not they own a house:

Age	Annual income in USD	House ownership status
23	50,000	Non-owner
37	34,000	Non-owner
48	40,000	Owner
52	30,000	Non-owner
28	95,000	Owner
25	78,000	Non-owner
35	130,000	Owner
32	105,000	Owner
20	100,000	Non-owner
40	60,000	Owner
50	80,000	Peter



House ownership and annual income

The aim is to predict whether Peter, aged 50, with an income of \$80,000 per year, owns a house and could be a potential customer for our insurance company.

Analysis

In this case, we could try to apply the 1-NN algorithm. However, we should be careful about how we measure the distances between the data points, since the income range is much wider than the age range. Income levels of USD 115 k and USD 116 k are USD 1,000 apart. The two data points for these incomes would be very far apart. However, relative to each other, the difference between these data points isn't actually that big. Because we consider both measures (age and yearly income) to be about as important as each other, we would scale both from 0 to 1 according to the following formula:

$$\text{ScaledQuantity} = (\text{ActualQuantity} - \text{MinQuantity}) / (\text{MaxQuantity} - \text{MinQuantity})$$

In our particular case, this reduces to the following:

$$\text{ScaledAge} = (\text{ActualAge} - \text{MinAge}) / (\text{MaxAge} - \text{MinAge})$$

$$\text{ScaledIncome} = (\text{ActualIncome} - \text{MinIncome}) / (\text{MaxIncome} - \text{MinIncome})$$

After scaling, we get the following data:

Age	Scaled age	Annual income in USD	Scaled annual income	House ownership status
23	0.09375	50,000	0.2	Non-owner
37	0.53125	34,000	0.04	Non-owner
48	0.875	40,000	0.1	Owner
52	1	30,000	0	Non-owner
28	0.25	95,000	0.65	Owner
25	0.15625	78,000	0.48	Non-owner
35	0.46875	130,000	1	Owner
32	0.375	105,000	0.75	Owner
20	0	100,000	0.7	Non-owner
40	0.625	60,000	0.3	Owner
50	0.9375	80,000	0.5	?

Now, if we apply the 1-NN algorithm with the Euclidean metric, we will find out that Peter more than likely owns a house. Note that, without rescaling, the algorithm would yield a different result. Refer to *Exercise 1.5* for more information.

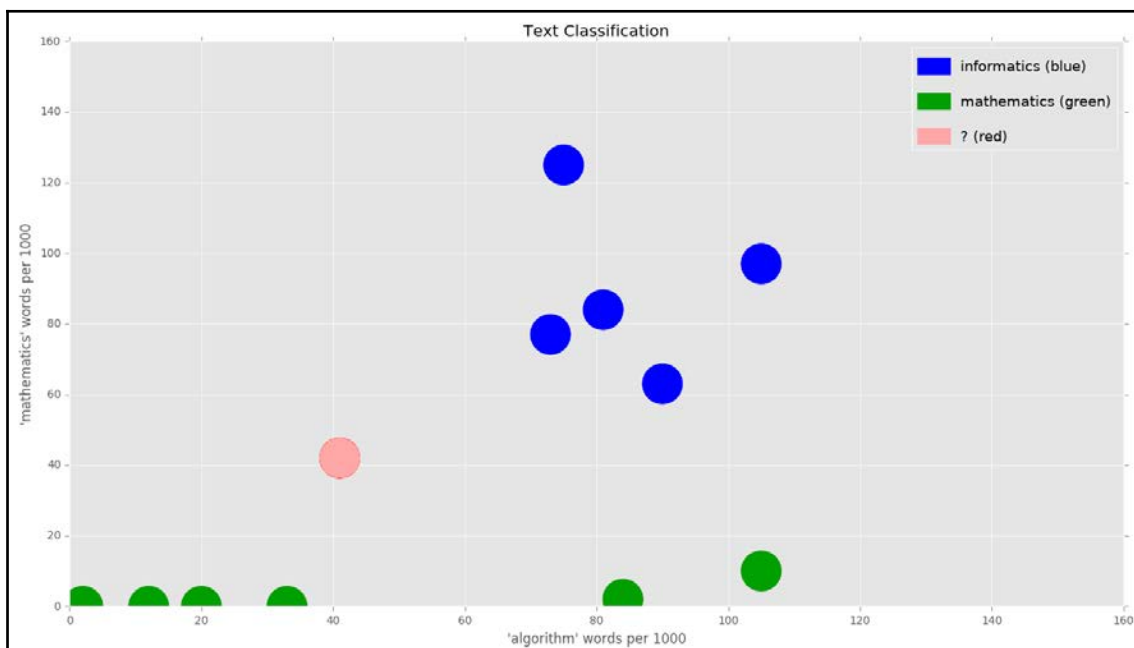
Text classification – using non-Euclidean distances

We are given the following word counts relating to the keywords **algorithm** and **computer**, for documents of the classes, in the informatics and mathematics subject classifications:

Algorithm words per 1,000	Computer words per 1,000	Subject classification
153	150	Informatics
105	97	Informatics
75	125	Informatics
81	84	Informatics
73	77	Informatics
90	63	Informatics
20	0	Mathematics
33	0	Mathematics
105	10	Mathematics
2	0	Mathematics
84	2	Mathematics
12	0	Mathematics
41	42	?

Those documents having a high incidence of the words **algorithm** and **computer** are in the `informatics` class. The `mathematics` class happens to contain documents with a high incidence of the word **algorithm** in some cases, for example, a document concerned with the Euclidean algorithm from the field of number theory. But, since the `mathematics` class tends to be applied less than `informatics` in the area of algorithms, the word **computer** comes up less frequently in the documents.

We would like to classify a document that has 41 instances of the word **algorithm** per 1,000 words, and 42 instances of the word **computer** per 1,000 words:



Analysis

Using, for example, the 1-NN algorithm and the Manhattan or Euclidean distance would result in the document in question being assigned to the `mathematics` class. However, intuitively, we should instead use a different metric to measure the distance, as the document in question has a much higher incidence of the word **computer** than other known documents in the class of `mathematics`.

Another candidate metric for this problem is a metric that would measure the proportion of the for the words or the angle between the instances in documents. Instead of the angle, you could take the cosine of the angle, $\cos(\theta)$, and then use the well-known dot product formula to calculate $\cos(\theta)$.

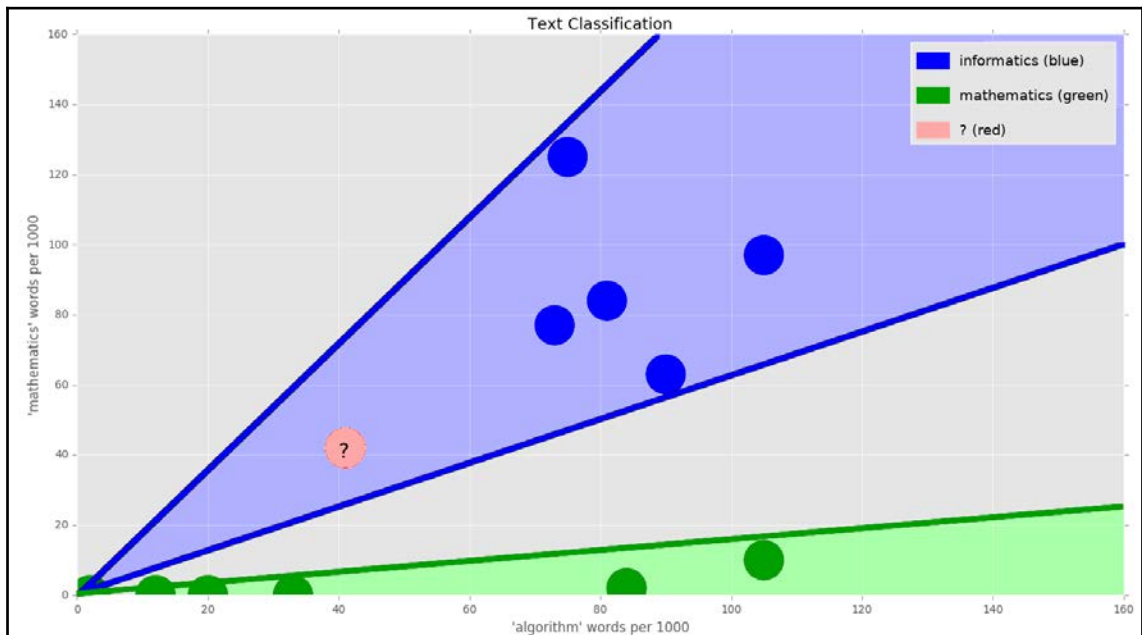
Let's use $a=(a_x, a_y)$, $b=(b_x, b_y)$. Use the following formula:

$$|a||b| \cos(\theta) = a \cdot b = a_x \cdot b_x + a_y \cdot b_y$$

This will derive the following:

$$\cos(\theta) = \frac{a_x \cdot b_x + a_y \cdot b_y}{|a||b|}$$

Using the cosine distance metric, you could classify the document in question to the `informatics` class:



Text classification – k-NN in higher dimensions

Suppose we are given documents and would like to classify other documents based on their word frequency counts. For example, the 120 most frequently occurring words found in the Project Gutenberg e-book of the King James Bible are as follows:

1. the 8.07%	41. when 0.36%	81. go 0.19%
2. and 6.51%	42. this 0.36%	82. hand 0.18%
3. of 4.37%	43. out 0.35%	83. us 0.18%
4. to 1.72%	44. were 0.35%	84. saying 0.18%
5. that 1.63%	45. upon 0.35%	85. made 0.18%
6. in 1.60%	46. man 0.34%	87. went 0.18%
7. he 1.31%	47. you 0.34%	88. even 0.18%
8. shall 1.24%	48. by 0.33%	89. do 0.17%
9. for 1.13%	49. Israel 0.32%	90. now 0.17%
10. unto 1.13%	50. king 0.32%	91. behold 0.17%
11. i 1.11%	51. son 0.30%	92. saith 0.16%
12. his 1.07%	52. up 0.30%	93. therefore 0.16%
13. a 1.04%	53. there 0.29%	94. every 0.16%
14. lord 1.00%	54. hath 0.28%	95. these 0.15%
15. they 0.93%	55. then 0.27%	96. because 0.15%
16. be 0.88%	56. people 0.27%	97. or 0.15%
17. is 0.88%	57. came 0.26%	98. after 0.15%
18. him 0.84%	58. had 0.25%	99. our 0.15%
19. not 0.83%	59. house 0.25%	100. things 0.15%
20. them 0.81%	60. on 0.25%	101. father 0.14%
21. it 0.77%	61. into 0.25%	102. down 0.14%
22. with 0.76%	62. her 0.25%	103. sons 0.14%
23. all 0.71%	63. come 0.25%	104. hast 0.13%
24. thou 0.69%	64. one 0.25%	105. David 0.13%
25. thy 0.58%	65. we 0.23%	106. o 0.13%
26. was 0.57%	66. children 0.23%	107. make 0.13%
27. god 0.56%	67. s 0.23%	108. say 0.13%
28. which 0.56%	68. before 0.23%	109. may 0.13%
29. my 0.55%	69. your 0.23%	110. over 0.13%
30. me 0.52%	70. also 0.22%	111. did 0.13%
31. said 0.50%	71. day 0.22%	112. earth 0.12%
32. but 0.50%	72. land 0.22%	113. what 0.12%
33. ye 0.50%	74. so 0.21%	114. Jesus 0.12%
34. their 0.50%	75. men 0.21%	115. she 0.12%
35. have 0.49%	76. against 0.21%	116. who 0.12%
36. will 0.48%	77. shalt 0.20%	117. great 0.12%
37. thee 0.48%	78. if 0.20%	118. name 0.12%
38. from 0.46%	79. at 0.20%	119. any 0.12%
39. as 0.44%	80. let 0.19%	120. thine 0.12%
40. are 0.37%		

The task is to design a metric that, given the word frequencies for each document, would accurately determine how semantically close those documents are. Consequently, such a metric could be used by the k-NN algorithm to classify the unknown instances in the new documents based on the existing documents.

Analysis

Suppose that we consider, for example, N most frequent words in our corpus of documents. Then, we count the word frequencies for each of the N words in a given document and put them in an N -dimensional vector that will represent that document. Then, we define a distance between two documents to be the distance (for example, Euclidean) between the two-word frequency vectors of those documents.

The problem with this solution is that only certain words represent the actual content of the book, and others need to be present in the text because of grammar rules or their general basic meaning. For example, out of the 120 most frequently encountered words in the Bible, each word is of different importance. In the following table, we have highlighted the words that have both a high frequency in the Bible and an important meaning:

• lord - used 1.00%	• Israel - 0.32%	• David - 0.13%
• god - 0.56%	• king - 0.32%	• Jesus - 0.12%

These words are less likely to be present in mathematical texts, for example, but more likely to be present in texts concerned with religion or Christianity.

However, if we just look at the six most frequent words in the Bible, they happen to be less useful with regard to detecting the meaning of the text:

• the - 8.07%	• of - 4.37%	• that - 1.63%
• and - 6.51%	• to - 1.72%	• in - 1.60%

Texts concerned with mathematics, literature, and other subjects will have similar frequencies for these words. Differences may result mostly from the writing style.

Therefore, to determine a similarity distance between two documents, we only need to look at the frequency counts of the important words. Some words are less important—these dimensions are better reduced, as their inclusion can lead to misinterpretation of the results in the end. Thus, what we are left to do is choose the words (dimensions) that are important to classify the documents in our corpus. For this, consult *Problem 6*.

Summary

In this chapter, we learned that the k -nearest neighbor algorithm is a classification algorithm that assigns the majority class among the k -nearest neighbors to a given data point. The distance between two points is measured by a metric. We covered examples of distances, including the Euclidean distance, Manhattan distance, tangential distance, and cosine distance. We also discussed how experiments with various parameters and cross-validation can help to establish which parameter, k , and which metric should be used.

We also learned that the dimensionality and position of a data point are determined by its qualities. A large number of dimensions can result in low accuracy of the k -NN algorithm. Reducing the dimensions of qualities of lesser importance can increase accuracy. Similarly, to increase accuracy further, distances for each dimension should be scaled according to the importance of the quality of that dimension.

In the next chapter, we will look at the Naive Bayes algorithm, which classifies an element based on probabilistic methods using Bayes' theorem.

Problems

In this section, we will discuss the following problems:

- Mary and her temperature preference problems
- Map of Italy – choosing the value of k
- House ownership

In order to learn from the material from this chapter in the best way possible, please analyze these problems on your own first before looking at the *Analysis* section at the end of this chapter.

Mary and her temperature preference problems

Problem 1: Imagine that you know that your friend Mary feels cold when it is -50°C , but she feels warm when it is 20°C . What would the 1-NN algorithm say about Mary? Would she feel warm or cold at temperatures of 22, 15, and -10 ? Do you think that the algorithm predicted Mary's body's perception of the temperature correctly? If not, please give your reasons and suggest why the algorithm did not give appropriate results and what would need to be improved for the algorithm to make a better classification.

Problem 2: Do you think that the 1-NN algorithm would yield better results than using the k -NN algorithm for $k > 1$?

Problem 3: We collected more data and found out that Mary feels warm at 17°C , but cold at 18°C . Using our own common sense, Mary should feel warmer when the temperature is higher. Can you explain the possible cause of the discrepancies in the data? How could we improve the analysis of our data? Should we also collect some non-temperature data? Suppose that we have the only piece of temperature data available; do you think that the 1-NN algorithm would still yield better results with data like this? How should we choose k for the k -NN algorithm to perform well?

Map of Italy – choosing the value of k

Problem 4: We are given a partial map of Italy for the Map of Italy problem. However, suppose that the complete data is not available. Thus, we cannot calculate the error rate on all of the predicted points for different values of k . How should you choose the value of k for the k -NN algorithm, to complete the map of Italy with a view to maximizing its accuracy?

House ownership

Problem 5: Using the data from the section concerned with the problem of house ownership, find the closest neighbor to Peter by using the Euclidean metric:

- a) Without rescaling the data
- b) By using the scaled data

Is the closest neighbor in:

- a) The same as the neighbor in?
- b) Which of the neighbors owns the house?

Problem 6: Suppose you would like to find books or documents in Gutenberg's corpus (www.gutenberg.org) that are similar to a selected book from the corpus (for example, the Bible) by using a certain metric and the 1-NN algorithm. How would you design a metric measuring the similarity distance between the two books?

Analysis

Problem 1: 8°C is closer to 20°C than to -50°C . So, the algorithm would classify Mary as feeling warm at -8°C . But this is not likely to be true if we use our common sense and knowledge. In more complex examples, we may be deluded by the results of the analysis and make false conclusions due to our lack of expertise. But remember that data science makes use of substantive and expert knowledge, not only data analysis. In order to draw sound conclusions, we should have a good understanding of the problem and our data.

The algorithm further says that at 22°C , Mary should feel warm, and there is no doubt in that, as 22°C is higher than 20°C , and a human being feels warmer with a higher temperature; again, a trivial use of our knowledge. For 15°C , the algorithm would deem Mary to feel warm, but if we use our common sense, we may not be that certain of this statement.

To be able to use our algorithm to yield better results, we should collect more data. For example, if we find out that Mary feels cold at 14°C , then we have a data instance that is very close to 15° and, thus, we can guess with a higher degree of certainty that Mary would feel cold at a temperature of 15° .

Problem 2: The data we are dealing with is just one-dimensional and is also partitioned into two parts, cold and warm, with the following property: the higher the temperature, the warmer a person feels. Also, even if we know how Mary feels at the temperatures -40 , -39 , ..., 39 , and 40 , we still have a very limited amount of data instances – just one for around every degree Celsius. For these reasons, it is best to just look at one closest neighbor.

Problem 3: Discrepancies in the data can be caused by inaccuracies in the tests carried out. This could be mitigated by performing more experiments.

Apart from inaccuracy, there could be other factors that influence how Mary feels: for example, the wind speed, humidity, sunshine, how warmly Mary is dressed (whether she has a coat on with jeans, or just shorts with a sleeveless top, or even a swimming suit), and whether she is wet or dry. We could add these additional dimensions (wind speed and how she is dressed) into the vectors of our data points. This would provide more, and better quality, data for the algorithm and, consequently, better results could be expected.

If we have only temperature data, but more of it (for example, 10 instances of classification for every degree Celsius), then we could increase the k value and look at more neighbors to determine the temperature more accurately. But this purely relies on the availability of the data. We could adapt the algorithm to yield the classification based on all the neighbors within a certain distance, d , rather than classifying based on the k -closest neighbors. This would make the algorithm work well in both cases when we have a lot of data within a close distance, and when we have just one data instance close to the instance that we want to classify.

Problem 4: For this purpose, you can use cross-validation (consult the *Cross-validation* section in *Appendix A – Statistics*) to determine the value of k with the highest accuracy. You could separate the available data from the partial map of Italy into learning and test data, for example, 80% of the classified pixels on the map would be given to a k-NN algorithm to complete the map. Then, the remaining 20% of the classified pixels from the partial map would be used to calculate the percentage of pixels with the correct classification according to the k-NN algorithm.

Problem 5:

- a) Without data rescaling, Peter's closest neighbor has an annual income of USD 78,000 and is aged 25. This neighbor does not own a house.
- b) After data rescaling, Peter's closest neighbor has an annual income of USD 60,000 and is aged 40. This neighbor owns a house.

Problem 6: To design a metric that accurately measures the similarity distance between the two documents, we need to select important words that will form the dimensions of the frequency vectors for the documents. Words that do not determine the semantic meaning of documents tend to have an approximately similar frequency count across all documents. Thus, instead, we could produce a list with the relative word frequency counts for a document. For example, we could use the following definition:

$$\text{relative_frequency_count}(\text{word}, \text{document}) = \frac{\text{frequency_count}(\text{word}, \text{document})}{\text{frequency_count}(\text{word}, \text{whole_corpus})}$$

Then, the document could be represented by an N -dimensional vector consisting of the word frequencies for N words with the highest relative frequency count. Such a vector will tend to consist of more important words than a vector of N words with the highest frequency count.

2

Naive Bayes

A Naive Bayes classification algorithm assigns a class to an element of a set that is most probable according to Bayes' theorem.

Let's say that A and B are probabilistic events. $P(A)$ is the probability of A being `true`. $P(A|B)$ is the conditional probability of A being `true`, given that B is `true`. If this is the case, then Bayes' theorem states the following:

$$P(A|B) = (P(B|A) * P(A)) / P(B)$$

$P(A)$ is the prior probability of A being `true` without the knowledge of the probability of $P(B)$ and $P(B|A)$. $P(A|B)$ is the posterior probability of A being `true`, taking into consideration additional knowledge about the probability of B being `true`.

In this chapter, you will learn about the following topics:

- How to apply Bayes' theorem in a basic way to compute the probability of a medical test that is correct in the simple example medical test
- How to grasp Bayes' theorem by proving its statement and its extension
- How to apply Bayes' theorem differently to independent and dependent variables in examples of playing chess
- How to apply Bayes' theorem to discrete random variables in examples of medical tests and playing chess, and for continuous random variables in an example of gender classification using the probability distribution of the continuous random variable
- How to implement an algorithm in Python to calculate the posterior probabilities of using Bayes' theorem

By the end of this chapter, you will be able to verify your understanding of Naive Bayes by solving problems. You will also be able to discern in which situations Bayes' theorem is an appropriate method of analysis, and when it is not.

Medical tests – basic application of Bayes' theorem

A patient takes a special cancer test that has an accuracy of $\text{test_accuracy}=99.9\%$ —if the result is positive, then 99.9% of the patients tested will suffer from that particular type of cancer. Conversely, 99.9% of the patients with a negative result will not suffer from that particular cancer.

Suppose that a patient is tested and the result is positive. What is the probability of that patient suffering from that particular type of cancer?

Analysis

We will use Bayes' theorem to ascertain the probability of the patient having cancer:

$$P(\text{cancer}|\text{test_positive}) = (P(\text{test_positive}|\text{cancer}) * P(\text{cancer})) / P(\text{test_positive})$$

To ascertain the prior probability that a patient has cancer, we have to find out how frequently cancer occurs among people. Say that we find out that 1 person in 100,000 suffers from this kind of cancer. Therefore, $P(\text{cancer})=1/100,000$. So, $P(\text{test_positive}|\text{cancer}) = \text{test_accuracy}=99.9\%=0.999$ given by the accuracy of the test.

$P(\text{test_positive})$ has to be computed as follows:

$$\begin{aligned} P(\text{test_positive}) &= P(\text{test_positive}|\text{cancer}) * P(\text{cancer}) + P(\text{test_positive}|\text{no_cancer}) * P(\text{no_cancer}) \\ &= \text{test_accuracy} * P(\text{cancer}) + (1 - \text{test_accuracy}) * (1 - P(\text{cancer})) \\ &= 2 * \text{test_accuracy} * P(\text{cancer}) + 1 - \text{test_accuracy} - P(\text{cancer}) \end{aligned}$$

Therefore, we can calculate the following:

$$\begin{aligned} P(\text{cancer}|\text{test_positive}) &= (\text{test_accuracy} * P(\text{cancer})) / (2 * \text{test_accuracy} * P(\text{cancer}) + 1 - \text{test_accuracy} - P(\text{cancer})) \\ &= 0.999 * 0.00001 / (2 * 0.999 * 0.00001 + 1 - 0.999 - 0.00001) \\ &= 0.00989128497 \text{ which is approximately } 1\% \end{aligned}$$

So, even if the result of the test is positive and the test has an accuracy of 99.9%, the probability of the patient having the type of cancer tested for is only approximately 1%. The probability of someone having cancer after taking the test is relatively low when compared to the high accuracy of the test, but is much higher than the probability of 1 in 100,000 (0.001%), as was known prior to taking the test, based on its occurrence in the population.

Bayes' theorem and its extension

In this section, we will state and prove both Bayes' theorem and its extension.

Bayes' theorem

Bayes' theorem states the following:

$$P(A|B) = [P(B|A) * P(A)] / P(B)$$

Proof

We can prove this theorem by using elementary set theory on the probability spaces of the events A and B . In other words, here, a probability event will be defined as a set of the possible outcomes in the probability space:

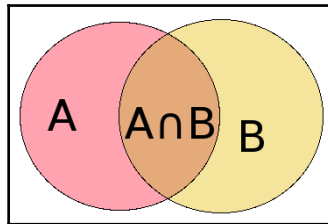


Figure 2.1: Probability space for the two events

As you can see in the preceding diagram, we can state the following relationships:

$$P(A|B) = P(A \cap B) / P(B)$$

$$P(B|A) = P(A \cap B) / P(A)$$

Rearranging these relationships, we get the following:

$$P(A \cap B) = P(A|B) * P(B)$$

$$P(A \cap B) = P(B|A) * P(A)$$

$$P(A|B) * P(B) = P(B|A) * P(A)$$

This is, in fact, Bayes' theorem:

$$P(A|B) = P(B|A) * P(A) / P(B)$$

This concludes the proof.

Extended Bayes' theorem

We can extend Bayes' theorem by taking more probability events into consideration.

Suppose that the events $B_1, \dots, B_n$ are conditionally independent given A . Let $\sim A$ denote the complement of A . Then, we have the following:

$$\begin{aligned} P(A|B_1, \dots, B_n) &= P(B_1, \dots, B_n|A) * P(A) / P(B_1, \dots, B_n) \\ &= [P(B_1|A) * \dots * P(B_n|A) * P(A)] / [P(B_1|A) * \dots * P(B_n|A) * P(A) + P(B_1|\sim A) * \dots * P(B_n|\sim A) * P(\sim A)] \end{aligned}$$

Proof

Since the events $B_1, \dots, B_n$ are conditionally independent given A (and also given $\sim A$), we get the following:

$$P(B_1, \dots, B_n|A) = P(B_1|A) * \dots * P(B_n|A)$$

Applying the simple form of Bayes' theorem and this fact, we thus have the following:

$$\begin{aligned}
 P(A|B_1, \dots, B_n) &= P(B_1, \dots, B_n|A) * P(A) / P(B_1, \dots, B_n) \\
 &= P(B_1|A) * \dots * P(B_n|A) * P(A) / [P(B_1, \dots, B_n|A) * P(A) + P(B_1, \dots, B_n|\sim A) * P(\sim A)] \\
 &= [P(B_1|A) * \dots * P(B_n|A) * P(A)] / [P(B_1|A) * \dots * P(B_n|A) * P(A) + P(B_1|\sim A) * \dots * P(B_n|\sim A) * P(\sim A)]
 \end{aligned}$$

This completes the proof.

Playing chess – independent events

Suppose we are given the following table of data. This tells us whether or not our friend will play a game of chess with us outside in the park, based on a number of weather-related conditions:

Temperature	Wind	Sunshine	Play
Cold	Strong	Cloudy	No
Warm	Strong	Cloudy	No
Warm	None	Sunny	Yes
Hot	None	Sunny	No
Hot	Breeze	Cloudy	Yes
Warm	Breeze	Sunny	Yes
Cold	Breeze	Cloudy	No
Cold	None	Sunny	Yes
Hot	Strong	Cloudy	Yes
Warm	None	Cloudy	Yes
Warm	Strong	Sunny	?

We would like to establish, by using Bayes' theorem, whether our friend would like to play a game of chess with us in the park given that the Temperature is Warm, the Wind is Strong, and it is Sunny.

Analysis

In this case, we may want to consider *Temperature*, *Wind*, and *Sunshine* as the independent random variables. The formula for the extended Bayes' theorem, when adapted, becomes the following:

$$P(\text{Play} = \text{Yes} | \text{Temperature} = \text{Warm}, \text{Wind} = \text{Strong}, \text{Sunshine} = \text{Sunny}) = R / (R + \sim R)$$

$$\text{Here, } R = P(\text{Temperature} = \text{Warm} | \text{Play} = \text{Yes}) * P(\text{Wind} = \text{Strong} | \text{Play} = \text{Yes}) *$$

$$P(\text{Sunshine} = \text{Sunny} | \text{Play} = \text{Yes}) * P(\text{Play} = \text{Yes}), \text{ and}$$

$$R = P(\text{Temperature} = \text{Warm} | \text{Play} = \text{No}) * P(\text{Wind} = \text{Strong} | \text{Play} = \text{No}) * P(\text{Sunshine} = \text{Sunny} | \text{Play} = \text{No}) \\ * P(\text{Play} = \text{No})$$

$$\sim R = P(\text{Temperature} = \text{Warm} | \text{Play} = \text{No}) * P(\text{Wind} = \text{Strong} | \text{Play} = \text{No}) \\ * P(\text{Sunshine} = \text{Sunny} | \text{Play} = \text{No}) * P(\text{Play} = \text{No})$$

Let's count the number of columns in the table with all known values to determine the individual probabilities.

$P(\text{Play} = \text{Yes}) = 6/10 = 3/5$, since there are 10 columns with complete data, and 6 of them have the value *Yes* for the attribute *Play*.

$P(\text{Temperature} = \text{Warm} | \text{Play} = \text{Yes}) = 3/6 = 1/2$, since there are 6 columns with the value *Yes* for the attribute *Play* and, of those, 3 have the value *Warm* for the attribute *Temperature*. Similarly, we have the following:

- $P(\text{Wind} = \text{Strong} | \text{Play} = \text{Yes}) = 1/6$
- $P(\text{Sunshine} = \text{Sunny} | \text{Play} = \text{Yes}) = 3/6 = 1/2$
- $P(\text{Play} = \text{No}) = 4/10 = 2/5$
- $P(\text{Temperature} = \text{Warm} | \text{Play} = \text{No}) = 1/4$
- $P(\text{Wind} = \text{Strong} | \text{Play} = \text{No}) = 2/4 = 1/2$
- $P(\text{Sunshine} = \text{Sunny} | \text{Play} = \text{No}) = 1/4$

Thus:

$$R = (1/2) * (1/6) * (1/2) * (3/5) = 1/40 \quad \text{and} \quad \sim R = (1/4) * (1/2) * (1/4) * (2/5) = 1/80$$

Therefore, we have the following:

$$P(\text{Play} = \text{Yes} | \text{Temperature} = \text{Warm}, \text{Wind} = \text{Strong}, \text{Sunshine} = \text{Sunny}) = R / (R + \sim R) = 2/3 \sim 67\%$$

This means that our friend is likely to be happy to play chess with us in the park in the stated weather conditions, with a probability of about 67%. Since this is a majority, we could classify the data vector (*Temperature=Warm, Wind=Strong, Sunshine=Sunny*) as being in the *Play=Yes* class.

Implementation of a Naive Bayes classifier

In this section, we will implement a program calculating the probability of a data item belonging to a certain class by using Bayes' theorem:

```
# source_code/2/naive_bayes.py
# A program that reads the CSV file with the data and returns
# the Bayesian probability for the unknown value denoted by ? to
# belong to a certain class.
# An input CSV file should be of the following format:
# 1. items in a row should be separated by a comma ','
# 2. the first row should be a heading - should contain a name for each
# column of the data.
# 3. the remaining rows should contain the data itself - rows with
# complete and rows with the incomplete data.
# A row with complete data is the row that has a non-empty and
# non-question mark value for each column. A row with incomplete data is
# the row that has the last column with the value of a question mark ?.
# Please, run this file on the example chess.csv to understand this help
# better:
# $ python naive_bayes.py chess.csv

import imp
import sys
sys.path.append('../common')
import common # noqa

# Calculates the Bayesian probability for the rows of incomplete data and
# returns them completed by the Bayesian probabilities. complete_data
# are the rows with the data that is complete and are used to calculate
# the conditional probabilities to complete the incomplete data.
def bayes_probability(heading, complete_data, incomplete_data,
                     enquired_column):
    conditional_counts = {}
    enquired_column_classes = {}
```

```

    for data_item in complete_data:
        common.dic_inc(enquired_column_classes,
                       data_item[enquired_column])
    for i in range(0, len(heading)):
        if i != enquired_column:
            common.dic_inc(
                conditional_counts, (
                    heading[i], data_item[i],
                    data_item[enquired_column]))

    completed_items = []
    for incomplete_item in incomplete_data:
        partial_probs = {}
        complete_probs = {}
        probs_sum = 0
        for enquired_group in enquired_column_classes.items():
            # For each class in the of the enquired variable A calculate
            # the probability  $P(A) * P(B_1|A) * P(B_2|A) * \dots * P(B_n|A)$  where
            #  $B_1, \dots, B_n$  are the remaining variables.
            probability = float(common.dic_key_count(
                enquired_column_classes,
                enquired_group[0])) / len(complete_data)
            for i in range(0, len(heading)):
                if i != enquired_column:
                    probability = probability * (float(
                        common.dic_key_count(
                            conditional_counts, (
                                heading[i], incomplete_item[i],
                                enquired_group[0])) / (
                                    common.dic_key_count(enquired_column_classes,
                                                            enquired_group[0]))))
            partial_probs[enquired_group[0]] = probability
            probs_sum += probability

        for enquired_group in enquired_column_classes.items():
            complete_probs[enquired_group[0]] = partial_probs[enquired_group[0]] / probs_sum
        incomplete_item[enquired_column] = complete_probs
        completed_items.append(incomplete_item)
    return completed_items

# Program start
if len(sys.argv) < 2:
    sys.exit('Please, input as an argument the name of the CSV file.')

(heading, complete_data, incomplete_data,
 enquired_column) = common.csv_file_to_ordered_data(sys.argv[1])

```

```
# Calculate the Bayesian probability for the incomplete data
# and output it.
completed_data = bayes_probability(
    heading, complete_data, incomplete_data, enquired_column)
print completed_data
```

We use the dictionary part from the common library:

```
# source_code/common/common.py
# Increments integer values in a dictionary.
def dic_inc(dic, key):
    if key is None:
        pass
    if dic.get(key, None) is None:
        dic[key] = 1
    else:
        dic[key] = dic[key] + 1

def dic_key_count(dic, key):
    if key is None:
        return 0
    if dic.get(key, None) is None:
        return 0
    else:
        return int(dic[key])
```

Input:

We save the data from the table, for example, the data that is related to playing chess playing chess, in the following CSV file:

```
source_code/2/naive_bayes/chess.csv
Temperature,Wind,Sunshine,Play
Cold,Strong,Cloudy,No
Warm,Strong,Cloudy,No
Warm,None,Sunny,Yes
Hot,None,Sunny,No
Hot,Breeze,Cloudy,Yes
Warm,Breeze,Sunny,Yes
Cold,Breeze,Cloudy,No
Cold,None,Sunny,Yes
Hot,Strong,Cloudy,Yes
Warm,None,Cloudy,Yes
Warm,Strong,Sunny,?
```

Output:

We provide the `chess.csv` file as the input to the Python program, calculating the probabilities of the data item (*Temperature=Warm, Wind=Strong, Sunshine=Sunny*) belonging to the classes present in the file by using `Play=Yes` and `Play=No`. As we found out earlier, the data item belongs to the `Play=Yes` class due to it having higher probability. Therefore, we classify the data item into that class:

```
$ python naive_bayes.py chess.csv
[
  ['Warm', 'Strong', 'Sunny', {
    'Yes': 0.6666666666666666,
    'No': 0.3333333333333333
  }]
]
```

Playing chess – dependent events

Suppose that we would like to find out whether our friend would like to play chess with us in a park in Cambridge, UK. But, this time, we are given different input data:

Temperature	Wind	Season	Play
Cold	Strong	Winter	No
Warm	Strong	Autumn	No
Warm	None	Summer	Yes
Hot	None	Spring	No
Hot	Breeze	Autumn	Yes
Warm	Breeze	Spring	Yes
Cold	Breeze	Winter	No
Cold	None	Spring	Yes
Hot	Strong	Summer	Yes
Warm	None	Autumn	Yes
Warm	Strong	Spring	?

So, now we are wondering how the answer to whether our friend would like to play in a park in Cambridge, UK, will change with this different data in regard to the **Temperature** being **Warm**, the **Wind** being **Strong**, and the **Season** being **Spring**.

Analysis

We may be tempted to use Bayesian probability to calculate the probability of our friend playing chess with us in the park. However, we should be careful, and ask whether the probability of the events are independent of each other.

In the previous example, where we used Bayesian probability, we were given the probability variables **Temperature**, **Wind**, and **Sunshine**. These are reasonably independent. Common sense tells us that a specific **Temperature** and **Sunshine** do not have a strong correlation to a specific **Wind** speed. It is true that sunny weather results in higher temperatures, but sunny weather is common even when the temperatures are very low. Hence, we considered even **Sunshine** and **Temperature** as being reasonably independent as random variables and applied Bayes' theorem.

However, in this example, **Temperature** and **Season** are closely related, especially in a location such as the UK, our stated location for the park. Unlike countries closer to the equator, temperatures in the UK vary greatly throughout the year. Winters are cold and summers are hot. Spring and fall have temperatures in between.

Therefore, we cannot apply Bayes' theorem here, as the random variables are dependent. However, we could still perform some analysis by using Bayes' theorem on the partial data. By eliminating sufficient dependent variables, the remaining ones could turn out to be independent. Since **Temperature** is a more specific variable than **Season**, and the two variables are dependent, let's keep just the **Temperature** variable. The remaining two variables, **Temperature** and **Wind**, are independent.

Thus, we get the following data:

Temperature	Wind	Play
Cold	Strong	No
Warm	Strong	No
Warm	None	Yes
Hot	None	No
Hot	Breeze	Yes
Warm	Breeze	Yes
Cold	Breeze	No
Cold	None	Yes
Hot	Strong	Yes
Warm	None	Yes
Warm	Strong	?

We can keep the duplicate rows, as they give us greater evidence of the occurrence of that specific data row.

Input:

Saving the table, we get the following CSV file:

```
# source_code/2/chess_reduced.csv
Temperature,Wind,Play
Cold,Strong,No
Warm,Strong,No
Warm,None,Yes
Hot,None,No
Hot,Breeze,Yes
Warm,Breeze,Yes
Cold,Breeze,No
Cold,None,Yes
Hot,Strong,Yes
Warm,None,Yes
Warm,Strong,?
```

Output:

We input the saved CSV file into the `naive_bayes.py` program and we get the following result:

```
python naive_bayes.py chess_reduced.csv
[['Warm', 'Strong', {'Yes': 0.49999999999999994, 'No': 0.5}]]
```

The first class, `Yes`, is going to be true, with a probability of 50%. The numerical difference resulted from using Python's non-exact arithmetic on the float numerical data type. The second class, `No`, has the same probability, that is, 50%, of being true. Thus, we cannot make a reasonable conclusion with the data that we have about the class of the vector (`Warm`, `Strong`). However, you have probably already noticed that this vector already occurs in the table with the resulting class `No`. Hence, our guess would be that this vector should just happen to exist in one class, `No`. But, to have greater statistical confidence, we would need more data or more independent variables to be involved.

Gender classification – Bayes for continuous random variables

So far, we have been given a probability event that belonged to one of a finite number of classes, for example, a temperature was classified as cold, warm, or hot. But how would we calculate the posterior probability if we were given the temperature in °C instead?

For this example, we are given the heights of five men and five women, as shown in the following table:

Height in cm	Gender
180	Male
174	Male
184	Male
168	Male
178	Male
170	Female
164	Female
155	Female
162	Female
166	Female
172	?

Suppose that the next person has a height of 172 cm. What gender is that person more likely to be, and with what probability?

Analysis

One approach to solving this problem could be to assign classes to the numerical values; for example, the people with a height of between 170 cm and 179 cm would be in the same class. With this approach, we may end up with a few classes that are very wide, for example, with a high cm range, or with classes that are more precise but have fewer members, and so we cannot use the full potential of the Bayes classifier. Similarly, using this method, we would not consider the classes of height interval (170, 180) and (180, 190) in centimeters, to be closer to each other than the classes (170, 180) and (190, 200).

Let's remind ourselves of the Bayes' formula here:

$$P(\text{male}|\text{height}) = P(\text{height}|\text{male}) * P(\text{male})/P(\text{height})$$

$$= P(\text{height}|\text{male}) * P(\text{male})/[P(\text{height}|\text{male}) * P(\text{male}) + P(\text{height}|\text{female}) * P(\text{female})]$$

Expressing the formula in the final form shown here removes the need to normalize the $P(\text{height}|\text{male})$ and $P(\text{height})$ to get the correct probability of a person being male based on their height.

Assuming that the height of people is distributed normally, we could use a normal probability distribution to calculate $P(\text{male}|\text{height})$. We assume that $P(\text{male})=0.5$, that is, it is equally likely that the person to be measured is of either gender. A normal probability distribution is determined by the mean, μ , and the variance, σ^2 , of the population:

$$f(x|\mu, \sigma^2) = \frac{e^{-\frac{(x-\mu)^2}{2\sigma^2}}}{\sqrt{2\sigma^2\pi}}$$

The following table shows the mean height and variance by gender:

Gender	Mean height	Height variance
Male	176.8	37.2
Female	163.4	30.8

Thus, we could calculate the following:

$$P(\text{height} = 172|\text{male}) = \exp[-(172 - 176.8)^2/(2 * 37.2)]/[\text{sqrt}(2 * 37.2 * \pi)] = 0.04798962999$$

$$P(\text{height} = 172|\text{female}) = \exp[-(172 - 163.4)^2/(2 * 30.8)]/[\text{sqrt}(2 * 30.8 * \pi)] = 0.02163711333$$

Note that these are not the probabilities, just the values of the probability density function. However, from these values, we can already observe that a person with a height of 172 cm is more likely to be male than female because, as follows:

$$P(\text{height} = 172|\text{male}) > P(\text{height} = 172|\text{female})$$

To be more precise:

$$\begin{aligned}
 & P(\text{male} | \text{height} = 172) \\
 &= P(\text{height} = 172 | \text{male}) * P(\text{male}) / [P(\text{height} = 172 | \text{male}) * P(\text{male}) + P(\text{height} = 172 | \text{female}) * P(\text{female})] \\
 &= 0.04798962999 * 0.5 / [0.04798962999 * 0.5 + 0.02163711333 * 0.5] = 0.68924134178 \sim 68.9\%
 \end{aligned}$$

Therefore, the probability of a person with a height of 172 cm being male is of 68.9%.

Summary

In this chapter, we learned about Naive Bayes and how it can be applied in different ways for different purposes.

Bayes' theorem states the following:

$$P(A|B) = (P(B|A) * P(A)) / P(B)$$

Here, $P(A|B)$ is the conditional probability of A being true, given that B is true. It is used to update the value of the probability that A is true given the new observations about other probabilistic events. This theorem can be extended to a statement with multiple random variables:

$$\begin{aligned}
 & P(A|B_1, \dots, B_n) \\
 &= [P(B_1|A) * \dots * P(B_n|A) * P(A)] / [P(B_1|A) * \dots * P(B_n|A) * P(A) + P(B_1|\sim A) * \dots * P(B_n|\sim A) * P(\sim A)]
 \end{aligned}$$

The random variables $B_1, \dots, B_n$ have to be conditionally independent given A . The random variables can be discrete or continuous and follow a probability distribution, for example, normal (Gaussian) distribution.

We also studied the discrete random variable. We learned that it is best to ensure that you have a data item for each value of a discrete random variable given any of the conditions (with a value of A) by collecting sufficient data.

The more independent random variables we have, the more accurately we can determine the posterior probability. However, the greater danger is that some of these variables could be dependent, resulting in imprecise final results. When variables are dependent, we may eliminate some of the dependent variables and consider only mutually independent variables, or consider another algorithm as an approach to solving the data science problem.

In the next chapter, we will learn how to classify an element with a decision tree algorithm.

Problems

Problem 1: A patient is tested for a virus, V . The accuracy of the test is 98%. This virus, V , is currently present in 4 out of 100 people in the region the patient lives in:

- What is the probability that a patient having the virus, V , if they test positive?
- What is the probability of a patient still having the virus if the result of the test is negative?

Problem 2: Apart from assessing whether patients are suffering from the virus, V (in *Problem 1*), by using the test, a doctor usually also checks for other symptoms. According to a doctor, about 85% of patients with symptoms such as fever, nausea, abdominal discomfort, and malaise have the virus, V :

- What is the probability of a patient having the virus, V , if they have the symptoms mentioned previously and their test result for the virus is positive?
- How likely is it that the patient has the virus if they have the symptoms mentioned previously, but the result of the test is negative?

Problem 3: On a certain island, one in two tsunamis is preceded by an earthquake. There have been 4 tsunamis and 6 earthquakes in the past 100 years. A seismological station recorded an earthquake in the ocean near the island. What is the probability that it will result in a tsunami?

Problem 4: Patients are tested with four independent tests to ascertain whether they have a certain illness:

Test1 positive	Test2 positive	Test3 positive	Test4 positive	Illness
Yes	Yes	Yes	No	Yes
Yes	Yes	No	Yes	Yes
No	Yes	No	No	No
Yes	No	No	No	No
No	No	No	No	No
Yes	Yes	Yes	Yes	Yes
Yes	No	Yes	Yes	Yes
No	Yes	No	No	No

No	No	Yes	Yes	No
Yes	Yes	No	Yes	Yes
Yes	No	Yes	No	Yes
Yes	No	No	Yes	Yes
No	Yes	Yes	No	?

We have taken a new patient, for whom the second and third tests are positive, while the first and fourth tests are negative. What is the probability that they suffer from the illness?

Problem 5: We are given the following table of words that have featured in emails and whether or not those emails were spam:

Money	Free	Rich	Naughty	Secret	Spam
No	No	Yes	No	Yes	Yes
Yes	Yes	Yes	No	No	Yes
No	No	No	No	No	No
No	Yes	No	No	No	Yes
Yes	No	No	No	No	No
No	Yes	No	Yes	Yes	Yes
No	Yes	No	Yes	No	Yes
No	No	No	Yes	No	Yes
No	Yes	No	No	No	No
No	No	No	No	Yes	No
Yes	Yes	Yes	No	Yes	Yes
Yes	No	No	No	Yes	Yes
No	Yes	Yes	No	No	No
Yes	No	Yes	No	Yes	?

a) Create a Naive Bayes algorithm to determine whether or not a given email is a spam message. What is the result when given an email that contains the words **Money**, **Rich**, and **Secret**, but does not contain the words **Free** and **Naughty**?

b) Do you agree with the result of the algorithm? Is the Naive Bayes algorithm, as used here, a good method for classifying emails? Justify your answers.

Problem 6: This is a gender classification problem. Let's assume that we are given the following data about ten people:

Height in cm	Weight in kg	Hair length	Gender
180	75	Short	Male
174	71	Short	Male
184	83	Short	Male
168	63	Short	Male
178	70	Long	Male
170	59	Long	Female
164	53	Short	Female
155	46	Long	Female
162	52	Long	Female
166	55	Long	Female
172	60	Long	?

What is the probability that the 11th person, with a height of 172 cm, a weight of 60 kg, and long hair, is a man?

Analysis

Problem 1: Before the patient is given the test, the probability that they have the virus is 4%, $P(\text{virus}) = 4\% = 0.04$. The accuracy of the test is $\text{test_accuracy} = 98\% = 0.98$. We apply the formula from the medical test example:

$$\begin{aligned}
 P(\text{test_positive}) &= P(\text{test_positive}|\text{virus}) * P(\text{virus}) + P(\text{test_positive}|\text{no_virus}) * P(\text{no_virus}) \\
 &= \text{test_accuracy} * P(\text{virus}) + (1 - \text{test_accuracy}) * (1 - P(\text{virus})) \\
 &= 2 * \text{test_accuracy} * P(\text{virus}) + 1 - \text{test_accuracy} - P(\text{virus})
 \end{aligned}$$

Therefore, we have the following:

$$\begin{aligned}
 P(\text{virus}|\text{test_positive}) &= P(\text{test_positive}|\text{virus}) * P(\text{virus}) / P(\text{test_positive}) \\
 &= \text{test_accuracy} * P(\text{virus}) / P(\text{test_positive})
 \end{aligned}$$

$$\begin{aligned}
&= test_accuracy * P(virus) / [2 * test_accuracy * P(virus) + 1 - test_accuracy - P(virus)] \\
&= 0.98 * 0.04 / [2 * 0.98 * 0.04 + 1 - 0.98 - 0.04] = 0.67123287671 \sim 67\%
\end{aligned}$$

This means that there is a probability of about 67% that the patient has the virus, V , if the result of the test is positive:

$$\begin{aligned}
P(virus|test_negative) &= P(test_negative|virus) * P(virus) / P(test_negative) \\
&= (1 - test_accuracy) * P(virus) / [1 - P(test_positive)] \\
&= (1 - test_accuracy) * P(virus) / [1 - 2 * test_accuracy * P(virus) - 1 + test_accuracy + P(virus)] \\
&= (1 - test_accuracy) * P(virus) / [test_accuracy + P(virus) - 2 * test_accuracy * P(virus)] \\
&= (1 - 0.98) * 0.04 / [0.98 + 0.04 - 2 * 0.98 * 0.04] = 0.000849617672 \sim 0.08\%
\end{aligned}$$

If the test is negative, a patient can still have the virus, V , with a probability of 0.08%.

Problem 2: Here, we can assume that symptoms and a positive test result are conditionally independent events given that a patient has the virus, V . The variables we have are as follows:

$$P(virus) = 0.04$$

$$test_accuracy = 0.98$$

$$symptoms_accuracy = 85$$

Since we have two independent random variables, we will use an extended Bayes' theorem:

$$\begin{aligned}
\text{a) Let } R &= P(test_positive|virus) * P(symptoms|virus) * P(virus) \\
&= test_accuracy * symptoms_accuracy * P(virus) \\
&= 0.98 * 0.85 * 0.04 = 0.03332 \\
\sim R &= P(test_positive| \sim virus) * P(symptoms| \sim virus) * P(\sim virus) \\
&= (1 - test_accuracy) * (1 - symptoms_accuracy) * (1 - P(virus))
\end{aligned}$$

$$= (1 - 0.98) * (1 - 0.85) * (1 - 0.04) = 0.00288$$

$$\begin{aligned} \text{Then } P(\text{virus}|\text{test_positive}, \text{symptoms}) &= R/[R+ \sim R] \\ &= 0.03332/[0.03332 + 0.00288] = 0.92044198895 \sim 92\% \end{aligned}$$

So, a patient with symptoms of the virus, V , and a positive test result for the virus has the virus with a probability of approximately 92%.

Note that in the previous question, we learned that a patient has the virus with a probability of only about 67%, given that the result of the test was positive. But after adding another independent random variable, the probability increased to 92%, even though the symptom assessment was reliable only on 85% of patients. This implies that usually, it is a very good idea to add as many independent random variables as possible to calculate the posterior probability with a higher accuracy and confidence.

b) Here, the patient has symptoms of the virus, V , but the result of the test is negative. Thus, we have the following:

$$\begin{aligned} R &= P(\text{test_negative}|\text{virus}) * P(\text{symptoms}|\text{virus}) * P(\text{virus}) \\ &= (1 - \text{test_accuracy}) * \text{symptoms_accuracy} * P(\text{virus}) \\ &= (1 - 0.98) * 0.85 * 0.04 = 0.00068 \\ \sim R &= P(\text{test_negative}|\sim \text{virus}) * P(\text{symptoms}|\sim \text{virus}) * P(\sim \text{virus}) \\ &= \text{test_accuracy} * (1 - \text{symptoms_accuracy}) * (1 - P(\text{virus})) \\ &= 0.98 * (1 - 0.85) * (1 - 0.04) = 0.14112 \\ \text{Thus } P(\text{virus}|\text{test_negative}, \text{symptoms}) &= R/[R+ \sim R] \\ &= 0.00068/[0.00068 + 0.14112] = 0.0047954866 \sim 0.48\% \end{aligned}$$

Thus, a patient testing negative with regard to the test, but with symptoms of the virus, V , has a probability of 0.48% of having the virus.

Problem 3: We apply the basic form of Bayes' theorem:

$$\begin{aligned}
 P(\text{tsunami}|\text{earthquake}) &= P(\text{earthquake}|\text{tsunami}) * P(\text{tsunami}) / P(\text{earthquake}) \\
 &\sim 0.5 * (4/(365 * 100)) / (6/(365 * 100)) \\
 &\sim 0.5 * 4/6 \sim 1/3 = 33\%
 \end{aligned}$$

There is a 33% chance that there will be a tsunami following the recorded earthquake.

Note that, here, we set $P(\text{tsunami})$ to be the probability of a tsunami happening on some particular day in the past 100 years. We used a day as a unit to calculate the probability $P(\text{earthquake})$ as well. If we changed the unit to an hour, week, month, and so on for both $P(\text{tsunami})$ and $P(\text{earthquake})$, the result would still be the same. What is important in the calculation is the ratio $P(\text{tsunami}):P(\text{earthquake})=4:6=2/3:1$, that is, that a tsunami is about 66% more likely to happen than an earthquake.

Problem 4: We put the data into the program for calculating the posterior probability from the observations and get the following result:

`[[ 'No', 'Yes', 'Yes', 'No', { 'Yes': 0.0, 'No': 1.0} ]]`

According to this calculation, a patient tested should not suffer from the virus. However, the probability of 'No' seems quite high. It may be a good idea to get more data in order to obtain a more precise estimate of the probability of the patient being healthy.

Problem 5:

a) The result of the algorithm is as follows:

`[[ 'Yes', 'No', 'Yes', 'No', 'Yes', { 'Yes': 0.8459918784779665, 'No': 0.15400812152203341} ]]`

So, according to the Naive Bayes algorithm, when applied to the data in the table, the email is spam with a probability of about 85%.

b) This method may not be as good since the occurrence of certain words in a spam email is not independent. For example, spam emails containing the word "money" would likely try to convince the victim of a spam email that they could somehow get money from the spammer, and hence, other words, such as **Rich**, **Secret**, or **Free** are more likely to be present in such an email as well. The nearest neighbor algorithm would seem to perform better at spam email classification. You could verify the actual methods by using cross-validation.

Problem 6: For this problem, we will use the extended Bayes' theorem for both continuous and discrete random variables:

$$P(\text{male}|\text{height} = 172\text{cm}, \text{weight} = 60\text{kg}, \text{hair} = \text{long}) = R/[R + \sim R]$$

$$\text{where } R = P(\text{height} = 172\text{cm}|\text{male}) * P(\text{weight} = 60\text{kg}|\text{male}) * P(\text{hair} = \text{long}|\text{male}) * P(\text{male})$$

$$\sim R = P(\text{height} = 172\text{cm}|\text{female}) * P(\text{weight} = 60\text{kg}|\text{female}) * P(\text{hair} = \text{long}|\text{female}) * P(\text{female})$$

Let's summarize the given information in the following tables:

Gender	Mean height	Variance in height
Male	176.8	37.2
Female	163.4	30.8

Gender	Mean weight	Variance in weight
Male	72.4	53.8
Female	53	22.5

From this data, let's determine the other values that are needed to determine the final probability of the person being male:

$$P(\text{height} = 172\text{cm}|\text{male}) = 0.04798962999$$

$$P(\text{weight} = 60\text{kg}|\text{male}) = \exp[-(60 - 72.4)^2 / (2 * 53.8)] / [\text{sqrt}(2 * 53.8 * \pi)] = 0.01302907931$$

$$P(\text{hair} = \text{long}|\text{male}) = 0.2$$

$$P(\text{male}) = 0.5 \text{ by assumption}$$

$$P(\text{height} = 172\text{cm}|\text{female}) = 0.02163711333$$

$$P(\text{weight} = 60\text{kg}|\text{female}) = \exp[-(60 - 53)^2 / (2 * 22.5)] / [\text{sqrt}(2 * 22.5 * \pi)] = 0.02830872899 P(\text{hair} = \text{long}|\text{female}) = 0.8$$

$P(\text{female}) = 0.5$ by assumption, Hence, we have the following :

$$R = 0.04798962999 * 0.01302907931 * 0.2 * 0.5 = 0.00006252606$$

$$\sim R = 0.02163711333 * 0.02830872899 * 0.8 * 0.5 = 0.00024500767$$

$$P(\text{male}|\text{height} = 172\text{cm}, \text{weight} = 60\text{kg}, \text{hair} = \text{long})$$

$$= 0.00006252606 / [0.00006252606 + 0.00024500767] = 0.2033144787 \sim 20.3\%$$

Therefore, the probability of a person with a height of 172 cm, a weight of 60 kg, and long hair, being a male is 20.3%. Thus, they are more likely to be female.

3

Decision Trees

A decision tree is the arrangement of data in a tree structure where, at each node, data is separated into different branches according to the value of the attribute at the node.

To construct a decision tree, we will use a standard ID3 learning algorithm that chooses an attribute that classifies data samples in the best possible way to maximize the information gain—a measure based on information entropy.

In this chapter, we will cover the following topics:

- What a decision tree is and how to represent data in a decision tree through the swim preference example
- The concepts of information entropy and information gain, theoretically in the first instance, before applying the swim preference example in practical terms in the *Information theory* section
- How to use a ID3 algorithm to construct a decision tree from the training data, and its implementation in Python
- How to classify new data items using the constructed decision tree through the swim preference example
- How to carry out an alternative analysis of the chess playing problem in [Chapter 2](#), Naive Bayes, using decision trees and how the results of the two algorithms may differ
- You will verify your understanding in the *Problems* section and look at when to use decision trees as a method of analysis
- How to deal with data inconsistencies during decision tree construction with the *Going shopping* example

Swim preference – representing data using a decision tree

We may have certain preferences that determine whether or not we would swim. These can be recorded in a table, as follows:

Swimming suit	Water temperature	Swim preference
None	Cold	No
None	Warm	No
Small	Cold	No
Small	Warm	No
Good	Cold	No
Good	Warm	Yes

The data in this table can alternatively be presented in the following decision tree:

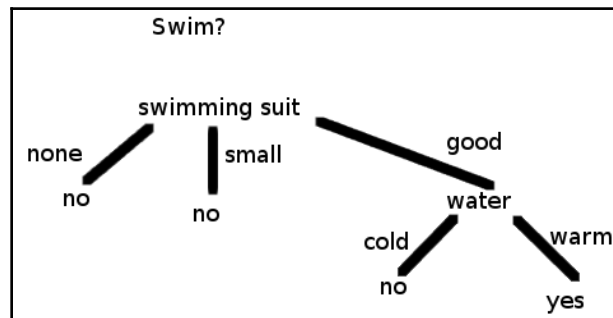


Figure 3.1: Decision tree for the Swim preference example

At the root node, we ask the question—do you have a swimming suit? The response to the question separates the available data into three groups, each with two rows. If the attribute is `swimming suit = none`, then two rows have the swim preference attribute as `no`. Therefore, there is no need to ask a question about the temperature of the water, as all the samples with the `swimming suit = none` attribute would be classified as `no`. This is also true for the `swimming suit = small` attribute. In the case of `swimming suit = good`, the remaining two rows can be divided into two classes – `no` and `yes`.

Without further information, we would not be able to classify each row correctly. Fortunately, there is one more question that can be asked about each row that classifies it correctly. For the row with the `water=cold` attribute, the swimming preference is `no`. For the row with the `water=warm` attribute, the swimming preference is `yes`.

To summarize, starting with the root node, we ask a question at every node and, based on the answer, we move down the tree until we reach a leaf node where we find the class of the data item corresponding to those answers.

This is how we can use a ready-made decision tree to classify samples of data. But it is also important to know how to construct a decision tree from data.

Which attribute has a question at which node? How does this reflect on the construction of a decision tree? If we change the order of the attributes, can the resulting decision tree classify better than another tree?

Information theory

Information theory studies the quantification of information, its storage, and communication. We introduce concepts of information entropy and information gain, which are used to construct a decision tree using the ID3 algorithm.

Information entropy

The information entropy of any given piece data is a measure of the smallest amount of information necessary to represent a data item from that data. The units of information entropy are familiar - bits, bytes, kilobytes, and so on. The lower the information entropy, the more regular the data is, and the more patterns occur in the data, thus, the smaller the quantity of information required to represent it. That is how compression tools on computers can take large text files and compress them to a much smaller size, as words and word expressions keep reoccurring, forming a pattern.

Coin flipping

Imagine we flip an unbiased coin. We would like to know whether the result is heads or tails. How much information do we need to represent the result? Both words, head, and tail, consist of four characters, and if we represent one character with one byte (8 bits), as is standard in the ASCII table, then we would need four bytes, or 32 bits, to represent the result.

But information entropy is the smallest amount of data necessary to represent the result. We know that there are only two possible results—heads or tails. If we agree to represent head with 0 and tail with 1, then 1 bit would be sufficient to communicate the result efficiently. Here, the data is the space of the possibilities of the result of the coin throw. It is the set $\{\text{head}, \text{tail}\}$ that can be represented as a set $\{0, 1\}$. The actual result is a data item from this set. It turns out that the entropy of the set is 1. This is owing to the probability of head and tail both being 50%.

Now imagine that the coin is biased and throws heads 25% of the time and tails 75% of the time. What would the entropy of the probability space $\{0, 1\}$ be this time? We could certainly represent the result with one bit of information. But can we do better? One bit is, of course, indivisible, but maybe we could generalize the concept of information to indiscete amounts.

In the previous example, we knew nothing about the previous result of the coin flip unless we looked at the coin. But in the example with the biased coin, we know that the result is more likely to be tails. If we recorded n results of coin flips in a file representing heads with 0 and tails with 1, then about 75 percent of the bits there would have the value 1, and 25 percent of them would have the value 0. The size of such a file would be n bits. But since it is more regular (a pattern of 1s prevails in it), a good compression tool should be able to compress it to less than n bits.

To learn the theory behind to compression and how much information is necessary to represent a data item, let's take a look at a precise definition of information entropy.

Definition of information entropy

Suppose that we are given a probability space, S , with the elements $1, 2, \dots, n$. The probability that an element, i , will be chosen from the probability space is p_i . The information entropy of the probability space is then defined as follows:

$$E(S) = -p_1 * \log_2(p_1) - \dots - p_n * \log_2(p_n)$$

In the preceding formula, $\log_2$ is a binary logarithm.

Hence, the information entropy of the probability space of unbiased coin throws is as follows:

$$E = -0.5 * \log_2(0.5) - 0.5 * \log_2(0.5) = 0.5 + 0.5 = 1$$

When the coin is biased, with a 25% chance of heads and a 75% chance of tails, then the information entropy of such a space is as follows:

$$E = -0.25 * \log_2(0.25) - 0.75 * \log_2(0.75) = 0.81127812445$$

This is less than 1. Thus, for example, if we had a large file with about 25% of 0 bits and 75% of 1 bits, a good compression tool should be able to compress it down to about 81.12% of its size.

Information gain

Information gain is the amount of information entropy gained as a result of a certain procedure. For example, if we would like to know the results of three fair coins, then the information entropy is 3. But if we could look at the third coin, then the information entropy of the result for the remaining two coins would be 2. Thus, by looking at the third coin, we gained one bit of information, so the information gain is 1.

We may also gain the information entropy by dividing the whole set, S , into sets, grouping them by a similar pattern. If we group elements by their value of an attribute, A , then we define the information gain as follows:

$$IG(S, A) = E(S) - \sum_{v \in \text{values}(A)} \left[\frac{|S_v|}{|S|} * E(S_v) \right]$$

Here, S_v is a set with the elements of S that have the value, v , for the attribute, A .

Swim preference – information gain calculation

Let's calculate the information gain for the six rows in the swim preference example by taking a `swimming suit` as an attribute. Because we are interested in whether a given row of data is classified as `no` or `yes` in response to the question as to whether you should go for a swim, we will use the swim preference to calculate the entropy and information gain. We partition the set S using the `swimming suit` attribute:

$$S_{none} = \{(none, cold, no), (none, warm, no)\}$$

$$S_{small} = \{(small, cold, no), (small, warm, no)\}$$

$$S_{good} = \{(good, cold, no), (good, warm, yes)\}$$

The information entropy of S is:

$$E(S) = -(1/6) * \log_2(1/6) - (5/6) * \log_2(5/6) \sim 0.65002242164$$

The information entropy of the partitions is:

$$E(S_{none}) = -(2/2) * \log_2(2/2) = -\log_2(1) = 0, \text{ since all instances have the class no.}$$

$$E(S_{small}) = 0 \text{ for a similar reason.}$$

$$E(S_{good}) = -(1/2) * \log_2(1/2) - (1/2) * \log_2(1/2) = 1$$

Therefore, the information gain is:

$$\begin{aligned} IG(S, \text{swimming suit}) &= E(S) - [(2/6) * E(S_{none}) + (2/6) * E(S_{small}) + (2/6) * E(S_{good})] \\ &= 0.65002242164 - (1/3) = 0.3166890883 \end{aligned}$$

If we chose the `water temperature` attribute to partition the set, S , what would be the information gain, $IG(S, \text{water temperature})$? The water temperature partitions the set, S , into the following sets:

$$S_{cold} = \{(none, cold, no), (small, cold, no), (good, cold, no)\}$$

$$S_{warm} = \{(none, warm, no), (small, warm, no), (good, warm, yes)\}$$

Their entropies are as follows:

$$E(S_{cold}) = 0, \text{ since all instances are classified as no.}$$

$$E(S_{warm}) = -(2/3) * \log_2(2/3) - (1/3) * \log_2(1/3) \sim 0.91829583405$$

Therefore, the information gain from partitioning the set, S , using the *water temperature* attribute is as follows:

$$\begin{aligned} IG(S, \text{water temperature}) &= E(S) - [(1/2) * E(S_{\text{cold}}) + (1/2) * E(S_{\text{warm}})] \\ &= 0.65002242164 - 0.5 * 0.91829583405 = 0.19087450461 \end{aligned}$$

This is less than $IG(S, \text{swimming suit})$. Therefore, we can gain more information about the set, S , (the classification of its instances) by partitioning it as per the *swimming suit* attribute instead of the *water temperature* attribute. This finding will be the basis of the ID3 algorithm when constructing a decision tree in the next section.

ID3 algorithm – decision tree construction

The ID3 algorithm constructs a decision tree from data based on the information gain. In the beginning, we start with the set, S . The data items in the set, S , have various properties, according to which we can partition the set, S . If an attribute, A , has the values $\{v_1, \dots, v_n\}$, then we partition the set, S , into the sets $S_1, \dots, S_n$, where the set, S_i , is a subset of the set, S , where the elements have the value, v_i , for the attribute, A .

If each element in the set, S , has the attributes $A_1, \dots, A_m$, then we can partition the set, S , according to any of the possible attributes. The ID3 algorithm partitions the set, S , according to the attribute that yields the highest information gain. Now suppose that it has the attribute, A_1 . Then, for the set, S , we have the partitions $S_1, \dots, S_n$, where A_1 has the possible values $\{v_1, \dots, v_n\}$.

Since we have not constructed a tree yet, we first place a root node. For every partition of S , we place a new branch from the root. Every branch represents one value of the selected attributes. A branch has data samples with the same value for that attribute. For every new branch, we can define a new node that will have data samples from its ancestor branch.

Once we have defined a new node, we choose another of the remaining attributes with the highest information gain for the data at that node to partition the data further at that node, and then define new branches and nodes. This process can be repeated until we run out of all the attributes for the nodes, or even earlier, until all the data at the node has the same class as our interest. In the case of the swim preference example, there are only two possible classes for the swimming preference—class *no* and class *yes*. The last node is called a **leaf node**, and decides the class of a data item from the data.

Swim preference – decision tree construction by the ID3 algorithm

Here, we describe, step by step, how an ID3 algorithm would construct a decision tree from the given data samples in the swim preference example. The initial set consists of six data samples:

```
S={ (none, cold, no) , (small, cold, no) , (good, cold, no) , (none, warm, no) , (small, warm, no) , (good, warm, yes) }
```

In the previous sections, we calculated the information gains for both, and the only non-classifying attributes, swimming suit, and water temperature, as follows:

```
IG(S, swimming suit)=0.3166890883
IG(S, water temperature)=0.19087450461
```

Hence, we would choose the `swimming suit` attribute as it has a higher information gain. There is no tree drawn yet, so we start from the root node. As the `swimming suit` attribute has three possible values – {`none`, `small`, `good`}, we draw three possible branches out of it for each. Each branch will have one partition from the partitioned set S : S_{none} , S_{small} , and S_{good} . We add nodes to the ends of the branches. S_{none} data samples have the same swimming preference class = `no`, so we do not need to branch that node with a further attribute and partition the set. Thus, the node with the data, S_{none} , is already a leaf node. The same is true for the node with the data, S_{small} .

But the node with the data, S_{good} , has two possible classes for swimming preference. Therefore, we will branch the node further. There is only one non-classifying attribute left—`water temperature`. So there is no need to calculate the information gain for that attribute with the data, S_{good} . From the node, S_{good} , we will have two branches, each with a partition from the set, S_{good} . One branch will have the set of the data sample, $S_{good, cold} = \{(good, cold, no)\}$; the other branch will have the partition, $S_{good, warm} = \{(good, warm, yes)\}$. Each of these two branches will end with a node. Each node will be a leaf node, because each node has data samples of the same value for the classifying swimming preference attribute.

The resulting decision tree has four leaf nodes and is the tree in *Figure 3.1 – Decision tree for the swim preference example*.

Implementation

We implement an ID3 algorithm that constructs a decision tree for the data given in a CSV file. All sources are in the chapter directory. The most important parts of the source code are given here:

```
# source_code/3/construct_decision_tree.py
# Constructs a decision tree from data specified in a CSV file.
# Format of a CSV file:
# Each data item is written on one line, with its variables separated
# by a comma. The last variable is used as a decision variable to
# branch a node and construct the decision tree.

import math
# anytree module is used to visualize the decision tree constructed by
# this ID3 algorithm.
from anytree import Node, RenderTree
import sys
sys.path.append('../common')
import common
import decision_tree

# Program start
csv_file_name = sys.argv[1]
verbose = int(sys.argv[2]) # verbosity level, 0 - only decision tree

# Define the enquired column to be the last one.
# I.e. a column defining the decision variable.
(heading, complete_data, incomplete_data,
 enquired_column) = common.csv_file_to_ordered_data(csv_file_name)

tree = decision_tree.constuct_decision_tree(
    verbose, heading, complete_data, enquired_column)
decision_tree.display_tree(tree)

# source_code/common/decision_tree.py
# ***Decision Tree library ***
# Used to construct a decision tree and a random forest.
import math
import random
import common
from anytree import Node, RenderTree
from common import printfv

# Node for the construction of a decision tree.
class TreeNode:

    def __init__(self, var=None, val=None):
```

```

        self.children = []
        self.var = var
        self.val = val

    def add_child(self, child):
        self.children.append(child)

    def get_children(self):
        return self.children

    def get_var(self):
        return self.var

    def get_val(self):
        return self.val

    def is_root(self):
        return self.var is None and self.val is None

    def is_leaf(self):
        return len(self.children) == 0

    def name(self):
        if self.is_root():
            return "[root]"
        return "[" + self.var + "=" + self.val + "]"

# Constructs a decision tree where heading is the heading of the table
# with the data, i.e. the names of the attributes.
# complete_data are data samples with a known value for every attribute.
# enquired_column is the index of the column (starting from zero) which
# holds the classifying attribute.
def construct_decision_tree(verbose, heading, complete_data,
enquired_column):
    return construct_general_tree(verbose, heading, complete_data,
                                  enquired_column, len(heading))

# m is the number of the classifying variables that should be at most
# considered at each node. m needed only for a random forest.
def construct_general_tree(verbose, heading, complete_data,
enquired_column, m):
    available_columns = []
    for col in range(0, len(heading)):
        if col != enquired_column:
            available_columns.append(col)
    tree = TreeNode()
    printfv(2, verbose, "We start the construction with the root node" +
            " to create the first node of the tree.\n")

```

```

        add_children_to_node(verbose, tree, heading, complete_data,
                             available_columns, enquired_column, m)

    return tree

# Splits the data samples into the groups with each having a different
# value for the attribute at the column col.
def split_data_by_col(data, col):
    data_groups = {}
    for data_item in data:
        if data_groups.get(data_item[col]) is None:
            data_groups[data_item[col]] = []
        data_groups[data_item[col]].append(data_item)
    return data_groups

# Adds a leaf node to node.
def add_leaf(verbose, node, heading, complete_data, enquired_column):
    leaf_node = TreeNode(heading[enquired_column],
                         complete_data[0][enquired_column])

    printfv(2, verbose,
            "We add the leaf node " + leaf_node.name() + ".\n")
    node.add_child(leaf_node)

# Adds all the descendants to the node.
def add_children_to_node(verbose, node, heading, complete_data,
                        available_columns, enquired_column, m):
    if len(available_columns) == 0:
        printfv(2, verbose, "We do not have any available variables " +
            "on which we could split the node further, therefore " +
            "we add a leaf node to the current branch of the tree. ")
        add_leaf(verbose, node, heading, complete_data, enquired_column)
        return -1

    printfv(2, verbose, "We would like to add children to the node " +
        node.name() + ".\n")

    selected_col = select_col(
        verbose, heading, complete_data, available_columns,
        enquired_column, m)
    for i in range(0, len(available_columns)):
        if available_columns[i] == selected_col:
            available_columns.pop(i)
            break

    data_groups = split_data_by_col(complete_data, selected_col)
    if (len(data_groups.items()) == 1):
        printfv(2, verbose, "For the chosen variable " +
            heading[selected_col] +
            " all the remaining features have the same value " +

```

```

        complete_data[0][selected_col] + ". " +
        "Thus we close the branch with a leaf node. ")
    add_leaf(verbose, node, heading, complete_data, enquired_column)
    return -1

if verbose >= 2:
    printfv(2, verbose, "Using the variable " +
            heading[selected_col] +
            " we partition the data in the current node, where" +
            " each partition of the data will be for one of the " +
            "new branches from the current node " + node.name() +
            ". " + "We have the following partitions:\n")
    for child_group, child_data in data_groups.items():
        printfv(2, verbose, "Partition for " +
                str(heading[selected_col]) + "=" +
                str(child_data[0][selected_col]) + ": " +
                str(child_data) + "\n")
    printfv(
        2, verbose, "Now, given the partitions, let us form the " +
        "branches and the child nodes.\n")
    for child_group, child_data in data_groups.items():
        child = TreeNode(heading[selected_col], child_group)
        printfv(2, verbose, "\nWe add a child node " + child.name() +
                " to the node " + node.name() + ". " +
                "This branch classifies %d feature(s): " +
                str(child_data) + "\n", len(child_data))
        add_children_to_node(verbose, child, heading, child_data, list(
            available_columns), enquired_column, m)
        node.add_child(child)
    printfv(2, verbose,
            "\nNow, we have added all the children nodes for the " +
            "node " + node.name() + ".\n")

# Selects an available column/attribute with the highest
# information gain.
def select_col(verbose, heading, complete_data, available_columns,
               enquired_column, m):
    # Consider only a subset of the available columns of size m.
    printfv(2, verbose,
            "The available variables that we have still left are " +
            str(numbers_to_strings(available_columns, heading)) + ". ")
    if len(available_columns) < m:
        printfv(
            2, verbose, "As there are fewer of them than the " +
            "parameter m=%d, we consider all of them. ", m)
        sample_columns = available_columns
    else:
        sample_columns = random.sample(available_columns, m)

```

```

    printfv(2, verbose,
            "We choose a subset of them of size m to be " +
            str(numbers_to_strings(available_columns, heading)) +
            ".")

    selected_col = -1
    selected_col_information_gain = -1
    for col in sample_columns:
        current_information_gain = col_information_gain(
            complete_data, col, enquired_column)
        # print len(complete_data), col, current_information_gain
        if current_information_gain > selected_col_information_gain:
            selected_col = col
            selected_col_information_gain = current_information_gain
    printfv(2, verbose,
            "Out of these variables, the variable with " +
            "the highest information gain is the variable " +
            heading[selected_col] +
            ". Thus we will branch the node further on this " +
            "variable. " +
            "We also remove this variable from the list of the " +
            "available variables for the children of the current node. ")
    return selected_col

# Calculates the information gain when partitioning complete_data
# according to the attribute at the column col and classifying by the
# attribute at enquired_column.
def col_information_gain(complete_data, col, enquired_column):
    data_groups = split_data_by_col(complete_data, col)
    information_gain = entropy(complete_data, enquired_column)
    for _, data_group in data_groups.items():
        information_gain -= (float(len(data_group)) / len(complete_data)
                            ) * entropy(data_group, enquired_column)
    return information_gain

# Calculates the entropy of the data classified by the attribute
# at the enquired_column.
def entropy(data, enquired_column):
    value_counts = {}
    for data_item in data:
        if value_counts.get(data_item[enquired_column]) is None:
            value_counts[data_item[enquired_column]] = 0
        value_counts[data_item[enquired_column]] += 1
    entropy = 0
    for _, count in value_counts.items():
        probability = float(count) / len(data)
        entropy -= probability * math.log(probability, 2)
    return entropy

```

Program input:

We input the data from the swim preference example into the program to construct a decision tree:

```
# source_code/3/swim.csv
swimming_suit,water_temperature,swim
None,Cold,No
None,Warm,No
Small,Cold,No
Small,Warm,No
Good,Cold,No
Good,Warm,Yes
```

Program output:

We construct a decision tree from the `swim.csv` data file, with the verbosity set to 0. The reader is encouraged to set the verbosity to 2 to see a detailed explanation of how exactly the decision tree is constructed:

```
$ python construct_decision_tree.py swim.csv 0
Root
├── [swimming_suit=Small]
│   ├── [water_temperature=Cold]
│   │   └── [swim=No]
│   └── [water_temperature=Warm]
│       └── [swim=No]
├── [swimming_suit=None]
│   ├── [water_temperature=Cold]
│   │   └── [swim=No]
│   └── [water_temperature=Warm]
│       └── [swim=No]
└── [swimming_suit=Good]
    ├── [water_temperature=Cold]
    │   └── [swim=No]
    └── [water_temperature=Warm]
        └── [swim=Yes]
```

Classifying with a decision tree

Once we have constructed a decision tree from the data with the attributes $A_1, \dots, A_m$ and the classes $\{c_1, \dots, c_k\}$, we can use this decision tree to classify a new data item with the attributes $A_1, \dots, A_m$ into one of the classes $\{c_1, \dots, c_k\}$.

Given a new data item that we would like to classify, we can think of each node, including the root, as a question for the data sample: *What value does that data sample have for the selected attribute, A_i ?* Then, based on the answer, we select a branch of the decision tree and move on to the next node. Then, another question is answered about the data sample, and another, until the data sample reaches the leaf node. A leaf node has one of the classes $\{c_1, \dots, c_k\}$ associated with it; for example, c_i . Then, the decision tree algorithm would classify the data sample into the class, c_i .

Classifying a data sample with the swimming preference decision tree

Let's construct a decision tree for the swimming preference example using the ID3 algorithm. Now that we have constructed the decision tree, we would like to use it to classify a data sample, *(good,cold,?)* into one of the two classes in the set $\{no,yes\}$.

Start with a data sample at the root of the tree. The first attribute that branches from the root is `swimming suit`, so we ask for the value of the `swimming suit` attribute of the sample *(good, cold,?)*. We learn that the value of the attribute is `swimming suit=good`; therefore, move down the rightmost branch with that value for its data samples. We arrive at the node with the `water temperature` attribute and ask the question: *What is the value of the water temperature attribute for the data sample (good, cold,?)?* We learn that for that data sample, we have `water temperature=cold`; therefore, we move down the left-hand branch into the leaf node. This leaf is associated with the `swimming preference=no` class. Therefore, the decision tree would classify the data sample *(good, cold,?)* to be in that swimming preference class; in other words, to complete it (by replacing the question mark with the exact data) to the data sample *(good, cold, no)*.

Therefore, the decision tree says that if you have a good swimming suit, but the water temperature is cold, then you would still not want to swim based on the data collected in the table.

Playing chess – analysis with a decision tree

Let's take an example from Chapter 2, *Naïve Bayes*, again:

Temperature	Wind	Sunshine	Play
Cold	Strong	Cloudy	No
Cold	Strong	Cloudy	No
Warm	None	Sunny	Yes
Hot	None	Sunny	No
Hot	Breeze	Cloudy	Yes
Warm	Breeze	Sunny	Yes
Cold	Breeze	Cloudy	No
Cold	None	Sunny	Yes
Hot	Strong	Cloudy	Yes
Warm	None	Cloudy	Yes
Warm	Strong	Sunny	?

We would like to find out whether our friend would like to play chess with us in the park. But this time, we would like to use decision trees to find the answer.

Analysis

We have the initial set, S , of the data samples, as follows:

```
S = { (Cold, Strong, Cloudy, No), (Warm, Strong, Cloudy, No), (Warm, None, Sunny, Yes),
      (Hot, None, Sunny, No), (Hot, Breeze, Cloudy, Yes), (Warm, Breeze, Sunny, Yes), (Cold, Breeze, Cloudy, No),
      (Cold, None, Sunny, Yes), (Hot, Strong, Cloudy, Yes), (Warm, None, Cloudy, Yes) }
```

First, we determine the information gain for each of the three non-classifying attributes: temperature, wind, and sunshine. The possible values for temperature are Cold, Warm, and Hot. Therefore, we will partition the set, S , into three sets:

```
Scold = { (Cold, Strong, Cloudy, No), (Cold, Breeze, Cloudy, No), (Cold, None, Sunny, Yes) }
Swarm = { (Warm, Strong, Cloudy, No), (Warm, None, Sunny, Yes), (Warm, Breeze, Sunny, Yes),
          (Warm, None, Cloudy, Yes) }
Shot = { (Hot, None, Sunny, No), (Hot, Breeze, Cloudy, Yes), (Hot, Strong, Cloudy, Yes) }
```

We calculate the information entropies for the sets first:

$$E(S) = -(4/10) * \log_2(4/10) - (6/10) * \log_2(6/10) = 0.97095059445$$

$$E(S_{cold}) = -(2/3) * \log_2(2/3) - (1/3) * \log_2(1/3) = 0.91829583405$$

$$E(S_{warm}) = -(1/4) * \log_2(1/4) - (3/4) * \log_2(3/4) = 0.81127812445$$

$$E(S_{hot}) = -(1/3) * \log_2(1/3) - (2/3) * \log_2(2/3) = 0.91829583405$$

$$\begin{aligned} \text{Thus, } IG(S, \text{temperature}) &= E(S) - [(|S_{cold}|/|S|) * E(S_{cold}) + (|S_{warm}|/|S|) * E(S_{warm}) + (|S_{hot}|/|S|) * E(S_{hot})] \\ &= 0.97095059445 - [(3/10) * 0.91829583405 + (4/10) * 0.81127812445 + (3/10) * 0.91829583405] \\ &= 0.09546184424 \end{aligned}$$

The possible values for the `wind` attribute are `None`, `Breeze`, and `Strong`. Thus, we will split the set, S , into the three partitions:

$$S_{none} = (\text{Warm, None, Sunny, Yes}), (\text{Hot, None, Sunny, No}), (\text{Cold, None, Sunny, Yes}), (\text{Warm, None, Cloudy, Yes})$$

$$S_{breeze} = (\text{Hot, Breeze, Cloudy, Yes}), (\text{Warm, Breeze, Sunny, Yes}), (\text{Cold, Breeze, Cloudy, No})$$

$$S_{strong} = (\text{Cold, Strong, Cloudy, No}), (\text{Warm, Strong, Cloudy, No}), (\text{Hot, Strong, Cloudy, Yes})$$

The information entropies of the sets are as follows:

$$E(S_{none}) = 0.81127812445$$

$$E(S_{breeze}) = 0.91829583405$$

$$E(S_{strong}) = 0.91829583405$$

$$\begin{aligned} \text{Thus, } IG(S, \text{wind}) &= E(S) - [(|S_{none}|/|S|) * E(S_{none}) + (|S_{breeze}|/|S|) * E(S_{breeze}) + (|S_{strong}|/|S|) * E(S_{strong})] \\ &= 0.97095059445 - [(4/10) * 0.81127812445 + (3/10) * 0.91829583405 + (3/10) * 0.91829583405] \\ &= 0.09546184424 \end{aligned}$$

Finally, the third attribute, `Sunshine`, has two possible values, `Cloudy` and `Sunny`. Hence, it splits the set, S , into two sets:

$$S_{cloudy} =$$

$$(\text{Cold, Strong, Cloudy, No}), (\text{Warm, Strong, Cloudy, No}), (\text{Hot, Breeze, Cloudy, Yes}), \\ (\text{Cold, Breeze, Cloudy, No}), (\text{Hot, Strong, Cloudy, Yes}), (\text{Warm, None, Cloudy, Yes})$$

$$S_{sunny} = (\text{Warm, None, Sunny, Yes}), (\text{Hot, None, Sunny, No}), (\text{Warm, Breeze, Sunny, Yes}), (\text{Cold, None, Sunny, Yes})$$

The entropies of the sets are as follows:

$$E(S_{cloudy}) = 1$$

$$E(S_{sunny}) = 0.81127812445$$

$$\begin{aligned} \text{Thus, } IG(S, \text{sunshine}) &= E(S) - [(|S_{cloudy}|/|S|) * E(S_{cloudy}) + (|S_{sunny}|/|S|) * E(S_{sunny})] \\ &= 0.97095059445 - [(6/10) * 1 + (4/10) * 0.81127812445] = 0.04643934467 \end{aligned}$$

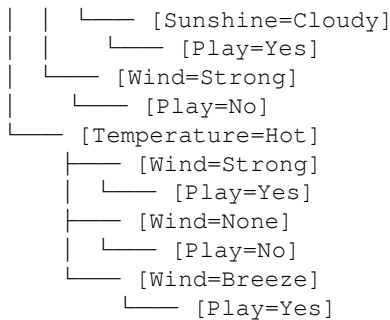
$IG(S, \text{wind})$ and $IG(S, \text{temperature})$ are greater than $IG(S, \text{sunshine})$. Both of them are equal; therefore, we can choose any of the attributes to form the three branches; for example, the first one, Temperature. In that case, each of the three branches would have the data samples S_{cold} , S_{warm} , and S_{hot} . At those branches, we could apply the algorithm further to form the rest of the decision tree. Instead, we will use the program to complete it:

Input:

```
source_code/3/chess.csv
Temperature, Wind, Sunshine, Play
Cold, Strong, Cloudy, No
Warm, Strong, Cloudy, No
Warm, None, Sunny, Yes
Hot, None, Sunny, No
Hot, Breeze, Cloudy, Yes
Warm, Breeze, Sunny, Yes
Cold, Breeze, Cloudy, No
Cold, None, Sunny, Yes
Hot, Strong, Cloudy, Yes
Warm, None, Cloudy, Yes
```

Output:

```
$ python construct_decision_tree.py chess.csv 0
Root
|--- [Temperature=Cold]
| |--- [Wind=Breeze]
| | |--- [Play=No]
| | |--- [Wind=Strong]
| | |--- [Play=No]
| | |--- [Wind=None]
| | |--- [Play=Yes]
|--- [Temperature=Warm]
| |--- [Wind=Breeze]
| | |--- [Play=Yes]
| | |--- [Wind=None]
| | |--- [Sunshine=Sunny]
| | |--- [Play=Yes]
```



Classification

Now that we have constructed the decision tree, we would like to use it to classify a data sample (*warm,strong,sunny,?*) into one of the two classes in the set $\{no,yes\}$.

We start at the root. What value does the `temperature` attribute have in that instance? Warm, so we go to the middle branch. What value does the `wind` attribute have in that instance? Strong, so the instance would fall into the class `No` since we have already arrived at the leaf node.

So, our friend would not want to play chess with us in the park, according to the decision tree classification algorithm. Please note that the Naive Bayes algorithm stated otherwise. An understanding of the problem is required to choose the best possible method. At other times, a method with greater accuracy is one that takes into consideration the results of several algorithms or several classifiers, as in the case of the random forest algorithm in Chapter 4, *Random Forests*.

Going shopping – dealing with data inconsistencies

We have the following data about the shopping preferences of our friend, Jane:

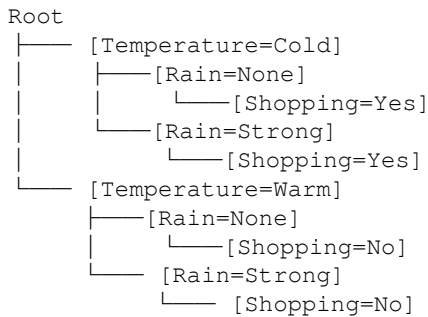
Temperature	Rain	Shopping
Cold	None	Yes
Warm	None	No
Cold	Strong	Yes
Cold	None	No

Warm	Strong	No
Warm	None	Yes
Cold	None	?

We would like to find out, using a decision tree, whether Jane would go shopping if the outside temperature was cold with no rain.

Analysis

Here, we should be careful, as there are instances of the data that have the same values for the same attributes, but different classes; that is, (cold, none, yes) and (cold, none, no). The program we made would form the following decision tree:



But at the leaf node [Rain=None] with the parent [Temperature=Cold], there are two data samples with both classes, no and yes. We cannot, therefore, classify an instance (cold, none, ?) accurately. For the decision tree algorithm to work better, we would have to provide a class at the leaf node with the greatest weight—that is, the majority class. An even better approach would be to collect values for more attributes for the data samples so that we can make a decision more accurately.

Therefore, given the available data, we are uncertain whether Jane would go shopping or not.

Summary

In this chapter, we looked at how a decision tree ID3 algorithm first constructs a decision tree from the input data and then classifies a new data instance using the constructed tree. The decision tree was constructed by selecting the attribute for branching with the highest information gain. We studied how information gain measures the amount of information that can be learned in terms of the gain in information entropy.

We also learned that the decision tree algorithm can achieve a different result from other algorithms, such as Naive Bayes.

In the next chapter, we will learn how to combine various algorithms or classifiers into a decision forest (called **random forest**) in order to achieve a more accurate result.

Problems

Problem 1: What is the information entropy of the following multisets?

a) {1,2}, b) {1,2,3}, c) {1,2,3,4}, d) {1,1,2,2}, e) {1,1,2,3}

Problem 2: What is the information entropy of the probability space induced by the biased coin that shows head with a probability of 10%, and tail with a probability of 90%?

Problem 3: Let's take another example of playing chess from Chapter 2, *Naive Bayes*:

a) What is the information gain for each of the non-classifying attributes in the table?

b) What is the decision tree constructed from the given table?

c) How would you classify a data sample (Warm, Strong, Spring, ?) according to the constructed decision tree?

Temperature	Wind	Season	Play
Cold	Strong	Winter	No
Warm	Strong	Autumn	No
Warm	None	Summer	Yes
Hot	None	Spring	No
Hot	Breeze	Autumn	Yes
Warm	Breeze	Spring	Yes
Cold	Breeze	Winter	No
Cold	None	Spring	Yes

Hot	Strong	Summer	Yes
Warm	None	Autumn	Yes
Warm	Strong	Spring	?

Problem 4: Mary and temperature preferences: Let's take the example from Chapter 1, *Classification Using K Nearest Neighbors*, regarding Mary's temperature preferences:

Temperature in °C	Wind speed in kmph	Mary's perception
10	0	Cold
25	0	Warm
15	5	Cold
20	3	Warm
18	7	Cold
20	10	Cold
22	5	Warm
24	6	Warm

We would like to use decision trees to decide whether our friend, Mary, would feel warm or cold in a room with a temperature of 16°C and a wind speed of 3 km/h.

Can you please explain how a decision tree algorithm could be used here and how beneficial it would be to use it for this example?

Analysis

Problem 1: Here are entropies of the multisets:

a) $E(1, 2) = -(1/2) * \log_2(1/2) - (1/2) * \log_2(1/2) = 1$

b) $E(1, 2, 3) = -(1/3) * \log_2(1/3) - (1/3) * \log_2(1/3) - (1/3) * \log_2(1/3) = 1.5849625$

c) $E(1, 2, 3, 4) = -(1/4) * \log_2(1/4) - (1/4) * \log_2(1/4) - (1/4) * \log_2(1/4) - (1/4) * \log_2(1/4) = 2$

d) $E(1, 1, 2, 2) = -(2/4) * \log_2(2/4) - (2/4) * \log_2(2/4) = 1$

e) $E(1, 1, 2, 3) = -(2/4) * \log_2(2/4) - (1/4) * \log_2(1/4) - (1/4) * \log_2(1/4) = 1.5$

Note here that the information entropy of the multisets that have more than two classes is greater than 1, so we need more than one bit of information to represent the result. But is this true for every multiset that has more than two classes of elements?

Problem 2:

$$E(10\% \text{ of heads}, 90\% \text{ of tails}) = -0.1 * \log_2(0.1) - (0.9) * \log_2(0.9) = 0.46899559358$$

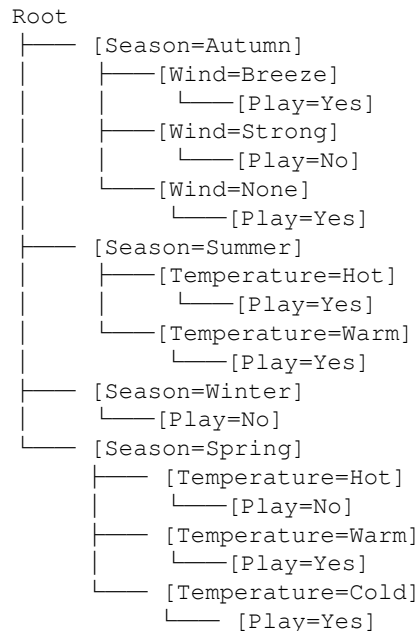
Problem 3: a) The information gains for the three attributes are as follows:

$$IG(S, \text{temperature}) = 0.0954618442383$$

$$IG(S, \text{wind}) = 0.0954618442383$$

$$IG(S, \text{season}) = 0.419973094022$$

b) Therefore, we would choose the `season` attribute to branch from the root node as it has the highest information gain. Alternatively, we can put all the input data into the program to construct a decision tree, as follows:



c) According to the decision tree constructed, we would classify the data sample (warm, strong, spring, ?) to the class `Play=Yes` by going to the bottom-most branch from the root node and then arriving at the leaf node by taking the middle branch.

Problem 4: Here, the decision tree algorithm may not perform that well without any processing of the data. If we considered every class of a temperature, then 25°C would still not occur in the decision tree as it is not in the input data, so we would not be able to classify how Mary would feel at 16°C and a wind speed of 3 km/h.

We could alternatively divide the temperature and wind speed into intervals in order to reduce the classes so that the resulting decision tree could classify the input instance. But it is this division, the intervals into which 25°C and 3 km/h should be classified, that is the fundamental part of the analysis procedure for this type of problem. Thus, decision trees without any serious modification could not analyze the problem well.

4

Random Forests

A random forest is a set of random decision trees (similar to the ones described in [Chapter 3, Decision Trees](#)), each generated on a random subset of data. A random forest classifies the features that belong to the class that is voted for by the majority of the random decision trees. Random forests tend to provide a more accurate classification of a feature than decision trees because of their decreased bias and variance.

In this chapter, we will cover the following topics:

- The tree bagging (or bootstrap aggregation) technique as part of random forest construction, but which can also be extended to other algorithms and methods in data science in order to reduce bias and variance and, hence, improve accuracy
- How to construct a random forest and classify a data item using a random forest constructed through the swim preference example
- How to implement an algorithm in Python that will construct a random forest
- The differences between the analysis of a problem using the Naive Bayes algorithm, decision trees, and random forest using the example of playing chess
- How the random forest algorithm can overcome the shortcomings of the decision tree algorithm and thus outperform it using the example of going shopping
- How a random forest can express level of confidence in its classification of a feature using the example of going shopping
- How decreasing the variance of a classifier can yield more accurate results, in the *Problems* section

Introduction to the random forest algorithm

In general, in order to construct a random forest, first we have to choose the number of trees that it will contain. A random forest does not tend to overfit (unless the data is very noisy), so choosing many decision trees will not decrease the accuracy of the prediction. A random forest does not tend to overfit (unless the data is very noisy), so having a higher number of decision trees will not decrease the accuracy of the prediction. It is important to have a sufficient number of decision trees so that more data is used for classification purposes when chosen randomly for the construction of a decision tree. On the other hand, the more decision trees there are, the more computational power is required. Also, increasing the number of decision trees fails to increase the accuracy of the classification by any significant degree.

In practice, you can run the algorithm on a specific number of decision trees, increase their number, and compare the results of the classification of smaller and bigger forests. If the results are very similar, then there is no reason to increase the number of trees.

To simplify the demonstration, throughout this book, we will use a small number of decision trees in a random forest.

Overview of random forest construction

We will describe how each tree is constructed in a random fashion. We construct a decision tree by selecting N training features randomly. This process of selecting the data randomly with a replacement for each tree is called **bootstrap aggregating**, or **tree bagging**. The purpose of bootstrap aggregating is to reduce the variance and bias in the results of the classification.

Say a feature has M variables that are used to classify the feature using the decision tree. When we have to make a branching decision at a node, in the ID3 algorithm, we choose the variable that resulted in the highest information gain. Here, in a random decision tree, at each node, we consider only at most m variables. We do not consider the ones that were already chosen sampled in a random fashion without any replacement from the given M variables. Then, of these m variables, we choose the one that results in the highest information gain.

The remainder of the construction of a random decision tree is carried out just as it was for a decision tree in Chapter 3, *Decision Trees*.

Swim preference – analysis involving a random forest

We will use the example from Chapter 3, *Decision Trees* concerning swim preferences. We have the same data table, as follows:

Swimming suit	Water temperature	Swim preference
None	Cold	No
None	Warm	No
Small	Cold	No
Small	Warm	No
Good	Cold	No
Good	Warm	Yes

We would like to construct a random forest from this data and use it to classify an item (Good, Cold, ?).

Analysis

We are given $M=3$ variables, according to which a feature can be classified. In a random forest algorithm, we usually do not use all three variables to form tree branches at each node. We only use a subset (m) of variables from M . So we choose m such that m is less than, or equal to, M . The greater m is, the stronger the classifier is in each constructed tree. However, as mentioned earlier, more data leads to more bias. But, because we use multiple trees (with a lower m), even if each constructed tree is a weak classifier, their combined classification accuracy is strong. As we want to reduce bias in a random forest, we may want to consider choosing an m parameter that is slightly less than M .

Hence, we choose the maximum number of variables considered at the node to be: $m = \min(M, \text{math.ceil}(2 * \text{math.sqrt}(M))) = \min(M, \text{math.ceil}(2 * \text{math.sqrt}(3))) = 3$.

We are given the following features:

```
[['None', 'Cold', 'No'], ['None', 'Warm', 'No'], ['Small', 'Cold', 'No'],
 ['Small', 'Warm', 'No'], ['Good', 'Cold', 'No'], ['Good', 'Warm', 'Yes']]
```

When constructing a random decision tree as part of a random forest, we will choose only a subset of these features randomly, together with their replacements.

Random forest construction

We will construct a random forest that will consist of two random decision trees.

Construction of random decision tree number 0

We are given six features as the input data. Of these, we choose six features at random with a replacement for the construction of this random decision tree:

```
[[ 'None', 'Warm', 'No'], [ 'None', 'Warm', 'No'], [ 'Small', 'Cold', 'No'],  
[ 'Good', 'Cold', 'No'], [ 'Good', 'Cold', 'No'], [ 'Good', 'Cold', 'No']]
```

We start the construction with the root node to create the first node of the tree. We would like to add children to the [root] node.

We have the following variables available: ['swimming_suit', 'water_temperature']. As there are fewer of these than the $m=3$ parameter, we consider both of them. Of these variables, the one with the highest information gain is a swimming suit.

Therefore, we will branch the node further on this variable. We will also remove this variable from the list of available variables for the children of the current node. Using the `swimming_suit` variable, we partition the data in the current node as follows:

- Partition for `swimming_suit=Small`: [['Small', 'Cold', 'No']]
- Partition for `swimming_suit=None`: [['None', 'Warm', 'No'], ['None', 'Warm', 'No']]
- Partition for `swimming_suit=Good`: [['Good', 'Cold', 'No'], ['Good', 'Cold', 'No'], ['Good', 'Cold', 'No']]

Using the preceding partitions, we create the branches and the child nodes.

We now add a child node, [`swimming_suit=Small`], to the [root] node. This branch classifies a single feature: [['Small', 'Cold', 'No']].

We would like to add children to the [`swimming_suit=Small`] node.

We have the following variable available: ['water_temperature']. As there is only one variable here, which is less than the $m=3$ parameter, we will consider this one. The one with the highest information gain is the `water_temperature` variable. Therefore, we will branch the node further on this variable. We will also remove this variable from the list of available variables for the children of the current node. For the chosen variable, `water_temperature`, all the remaining features have the same value: Cold. So, we end the branch with a leaf node, adding [swim=No].

We now add a child node, [swimming_suit=None], to the [root] node. This branch classifies two features: [['None', 'Warm', 'No'], ['None', 'Warm', 'No']].

We would like to add children to the [swimming_suit=None] node.

We have the following variable available: ['water_temperature']. As there is only one variable here, which is less than the $m=3$ parameter, we will consider this one. The one with the highest information gain is the `water_temperature` variable. Therefore, we will branch the node further on this variable. We will also remove this variable from the list of available variables for the children of the current node. For the chosen variable, `water_temperature`, all the remaining features have the same value: Warm. So, we end the branch with a leaf node, adding [swim=No].

We now add a child node, [swimming_suit=Good], to the [root] node. This branch classifies three features: [['Good', 'Cold', 'No'], ['Good', 'Cold', 'No'], ['Good', 'Cold', 'No']]

We would like to add children to the [swimming_suit=Good] node.

We have the following variable available: ['water_temperature']. As there is only one variable here, which is less than the $m=3$ parameter, we will consider this one. The one with the highest information gain is the `water_temperature` variable. Therefore, we will branch the node further on this variable. We will also remove this variable from the list of available variables for the children of the current node. For the chosen variable, `water_temperature`, all the remaining features have the same value: Cold. So, we end the branch with a leaf node, adding [swim=No].

Now, we have added all the children nodes to the [root] node.

Construction of random decision tree number 1

We are given six features as input data. Of these, we choose six features at random with a replacement for the construction of this random decision tree:

```
[[ 'Good', 'Warm', 'Yes'], [ 'None', 'Warm', 'No'], [ 'Good', 'Cold', 'No'],
[ 'None', 'Cold', 'No'], [ 'None', 'Warm', 'No'], [ 'Small', 'Warm', 'No']]
```

The remainder of the construction of random decision tree number 1 is similar to the construction of the previous random decision tree, number 0. The only difference is that the tree is built using a different randomly generated subset (as seen previously) of the initial data.

We begin construction with the root node to create the first node of the tree. We would like to add children to the `[root]` node.

We have the following variables available: `['swimming_suit', 'water_temperature']`. As there is only one variable here, which is less than the `m=3` parameter, we will consider this one. Of these variables, the one with the highest information gain is the `swimming_suit` variable.

Therefore, we will branch the node further on this variable. We will also remove this variable from the list of available variables for the children of the current node. Using the `swimming_suit` variable, we partition the data in the current node as follows:

- Partition for `swimming_suit=Small`: `[[ 'Small', 'Warm', 'No']]`
- Partition for `swimming_suit=None`: `[[ 'None', 'Warm', 'No'], [ 'None', 'Cold', 'No'], [ 'None', 'Warm', 'No']]`
- Partition for `swimming_suit=Good`: `[[ 'Good', 'Warm', 'Yes'], [ 'Good', 'Cold', 'No']]`

Now, given the partitions, let's create the branches and the child nodes. We add a child node, `[swimming_suit=Small]`, to the `[root]` node. This branch classifies a single feature: `[[ 'Small', 'Warm', 'No']]`.

We would like to add children to the `[swimming_suit=Small]` node.

We have the following variable available: `['water_temperature']`. As there is only one variable here, which is less than the `m=3` parameter, we will consider this one. The one with the highest information gain is the `water_temperature` variable. Therefore, we will branch the node further on this variable. We will also remove this variable from the list of available variables for the children of the current node. For the chosen variable, `water_temperature`, all the remaining features have the same value: Warm. So, we end the branch with a leaf node, adding `[swim=No]`.

We add a child node, `[swimming_suit=None]`, to the `[root]` node. This branch classifies three features: `[['None', 'Warm', 'No'], ['None', 'Cold', 'No'], and ['None', 'Warm', 'No']]`.

We would like to add children to the `[swimming_suit=None]` node.

We have the following variable available: `['water_temperature']`. As there is only one variable here, which is less than the `m=3` parameter, we will consider this one. The one with the highest information gain is the `water_temperature` variable. Therefore, we will branch the node further on this variable. We will also remove this variable from the list of available variables for the children of the current node. Using the `water temperature` variable, we partition the data in the current node as follows:

- Partition for `water_temperature=Cold`: `[['None', 'Cold', 'No']]`
- Partition for `water_temperature=Warm`: `[['None', 'Warm', 'No'], ['None', 'Warm', 'No']]`; now, given the partitions, let's create the branches and the child nodes

We add a child node, `[water_temperature=Cold]`, to the `[swimming_suit=None]` node. This branch classifies a single feature: `[['None', 'Cold', 'No']]`.

We do not have any available variables on which we could split the node further; therefore, we add a leaf node to the current branch of the tree. We add the `[swim=No]` leaf node. We add a child node, `[water_temperature=Warm]`, to the `[swimming_suit=None]` node. This branch classifies two features: `[['None', 'Warm', 'No'], and ['None', 'Warm', 'No']]`.

We do not have any available variables on which we could split the node further; therefore, we add a leaf node to the current branch of the tree. We add the `[swim=No]` leaf node.

Now, we have added all the children nodes to the `[swimming_suit=None]` node.

We add a child node, `[swimming_suit=Good]`, to the `[root]` node. This branch classifies two features: `[['Good', 'Warm', 'Yes'], and ['Good', 'Cold', 'No']]`

We would like to add children to the `[swimming_suit=Good]` node.

We have the following variable available: `['water_temperature']`. As there is only one variable here, which is less than the `m=3` parameter, we will consider this one. The one with the highest information gain is the `water_temperature` variable. Therefore, we will branch the node further on this variable. We will also remove this variable from the list of available variables for the children of the current node. Using the `water temperature` variable, we partition the data in the current node as follows:

- Partition for `water_temperature=Cold`: `[['Good', 'Cold', 'No']]`
- Partition for `water_temperature=Warm`: `[['Good', 'Warm', 'Yes']]`

Now, given the partitions, let's create the branches and the child nodes.

We add a child node, `[water_temperature=Cold]`, to the `[swimming_suit=Good]` node. This branch classifies a single feature: `[['Good', 'Cold', 'No']]`

We do not have any available variables on which we could split the node further; therefore, we add a leaf node to the current branch of the tree. We add the `[swim=No]` leaf node.

We add a child node, `[water_temperature=Warm]`, to the `[swimming_suit=Good]` node. This branch classifies a single feature: `[['Good', 'Warm', 'Yes']]`.

We do not have any available variables on which we could split the node further; therefore, we add a leaf node to the current branch of the tree. We add the `[swim=Yes]` leaf node.

Now, we have added all the children nodes of the `[swimming_suit=Good]` node.

We have also added all the children nodes to the `[root]` node.

Constructed random forest

We have completed the construction of the random forest, consisting of two random decision trees, as shown in the following code block:

```
Tree 0:
  Root
  |— [swimming_suit=Small]
  |   |— [swim=No]
  |— [swimming_suit=None]
  |   |— [swim=No]
  |— [swimming_suit=Good]
  |   |— [swim=No]

Tree 1:
  Root
  |— [swimming_suit=Small]
  |   |— [swim=No]
  |— [swimming_suit=None]
  |   |— [water_temperature=Cold]
  |       |— [swim=No]
  |       |— [water_temperature=Warm]
  |           |— [swim=No]
  |— [swimming_suit=Good]
  |   |— [water_temperature=Cold]
  |       |— [swim=No]
  |   |— [water_temperature=Warm]
  |       |— [swim=Yes]
```

The total number of trees in the random forest=2.

The maximum number of the variables considered at the node is m=3.

Classification using random forest

Because we use only a subset of the original data for the construction of the random decision tree, we may not have enough features to form a full tree that is able to classify every feature. In such cases, a tree will not return any class for a feature that should be classified. Therefore, we will only consider trees that classify a feature of a specific class.

The feature we would like to classify is ['Good', 'Cold', '?']. A random decision tree votes for the class to which it classifies a given feature using the same method to classify a feature as in Chapter 3, *Decision Trees*. Tree 0 votes for the No class. Tree 1 votes for the No class. The class with the maximum number of votes is No. Therefore, the constructed random forest classifies the feature ['Good', 'Cold', '?'] according to the class No.

Implementation of the random forest algorithm

We implement a random forest algorithm using a modified decision tree algorithm from the previous chapter. We also add an option to set a verbose mode within the program that can describe the whole process of how the algorithm works on a specific input—how a random forest is constructed with its random decision trees, and how this constructed random forest is used to classify other features.

You are encouraged to consult the `decision_tree.construct_general_tree` function from the previous chapter:

```
# source_code/4/random_forest.py
import math
import random
import sys
sys.path.append('../common')
import common # noqa
import decision_tree # noqa
from common import printfv # noqa

#Random forest construction
def sample_with_replacement(population, size):
    sample = []
    for i in range(0, size):
        sample.append(population[random.randint(0, len(population) - 1)])
    return sample

def construct_random_forest(verbose, heading, complete_data,
                            enquired_column, m, tree_count):
    printfv(2, verbose, "*** Random Forest construction ***\n")
    printfv(2, verbose, "We construct a random forest that will " +
            "consist of %d random decision trees.\n", tree_count)
    random_forest = []
    for i in range(0, tree_count):
        printfv(2, verbose, "\nConstruction of a random " +
            "decision tree number %d:\n", i)
        random_forest.append(construct_random_decision_tree(
            verbose, heading, complete_data, enquired_column, m))
    printfv(2, verbose, "\nTherefore we have completed the " +
            "construction of the random forest consisting of %d " +
            "random decision trees.\n", tree_count)
    return random_forest

def construct_random_decision_tree(verbose, heading, complete_data,
```

```

        enquired_column, m):
    sample = sample_with_replacement(complete_data, len(complete_data))
    printfv(2, verbose, "We are given %d features as the input data. " +
        "Out of these, we choose randomly %d features with the " +
        "replacement that we will use for the construction of " +
        "this particular random decision tree:\n" +
        str(sample) + "\n", len(complete_data),
        len(complete_data))
# The function construct_general_tree from the module decision_tree
# is written in the implementation section in the previous chapter
# on decision trees.
    return decision_tree.construct_general_tree(verbose, heading,
                                                sample,
                                                enquired_column, m)

# M is the given number of the decision variables, i.e. properties
# of one feature.
def choose_m(verbose, M):
    m = int(min(M, math.ceil(2 * math.sqrt(M))))
    printfv(2, verbose, "We are given M=" + str(M) +
        " variables according to which a feature can be " +
        "classified. ")
    printfv(3, verbose, "In random forest algorithm we usually do " +
        "not use all " + str(M) + " variables to form tree " +
        "branches at each node. ")
    printfv(3, verbose, "We use only m variables out of M. ")
    printfv(3, verbose, "So we choose m such that m is less than or " +
        "equal to M. ")
    printfv(3, verbose, "The greater m is, a stronger classifier an " +
        "individual tree constructed is. However, it is more " +
        "susceptible to a bias as more of the data is considered. " +
        "Since we in the end use multiple trees, even if each may " +
        "be a weak classifier, their combined classification " +
        "accuracy is strong. Therefore as we want to reduce a " +
        "bias in a random forest, we may want to consider to " +
        "choose a parameter m to be slightly less than M.\n")
    printfv(2, verbose, "Thus we choose the maximum number of the " +
        "variables considered at the node to be " +
        "m=min(M,math.ceil(2*math.sqrt(M))) " +
        "m=min(M,math.ceil(2*math.sqrt(%d)))=%d.\n", M, m)
    return m

#Classification
def display_classification(verbose, random_forest, heading,
                        enquired_column, incomplete_data):
    if len(incomplete_data) == 0:
        printfv(0, verbose, "No data to classify.\n")
    else:

```

```

        for incomplete_feature in incomplete_data:
            printfv(0, verbose, "\nFeature: " +
                    str(incomplete_feature) + "\n")
            display_classification_for_feature(
                verbose, random_forest, heading,
                enquired_column, incomplete_feature)

def display_classification_for_feature(verbose, random_forest, heading,
                                     enquired_column, feature):

    classification = {}
    for i in range(0, len(random_forest)):
        group = decision_tree.classify_by_tree(
            random_forest[i], heading, enquired_column, feature)
        common.dic_inc(classification, group)
        printfv(0, verbose, "Tree " + str(i) +
                " votes for the class: " + str(group) + "\n")
    printfv(0, verbose, "The class with the maximum number of votes " +
            "is '" + str(common.dic_key_max_count(classification)) +
            "'. Thus the constructed random forest classifies the " +
            "feature " + str(feature) + " into the class '" +
            str(common.dic_key_max_count(classification)) + "'.\n")

# Program start
csv_file_name = sys.argv[1]
tree_count = int(sys.argv[2])
verbose = int(sys.argv[3])

(heading, complete_data, incomplete_data,
 enquired_column) = common.csv_file_to_ordered_data(csv_file_name)
m = choose_m(verbose, len(heading))
random_forest = construct_random_forest(
    verbose, heading, complete_data, enquired_column, m, tree_count)
display_forest(verbose, random_forest)
display_classification(verbose, random_forest, heading,
                      enquired_column, incomplete_data)

```

Input:

As an input file to the implemented algorithm, we provide the data from the swim preference example:

```

# source_code/4/swim.csv
swimming_suit,water_temperature,swim
None,Cold,No
None,Warm,No
Small,Cold,No
Small,Warm,No
Good,Cold,No

```

Good, Warm, Yes
 Good, Cold, ?

Output:

We type the following command in the command line to get the output:

```
$ python random_forest.py swim.csv 2 3 > swim.out
```

2 means that we would like to construct two decision trees, and 3 is the level of the verbosity of the program, which includes detailed explanations of the construction of the random forest, the classification of the feature, and the graph of the random forest. The last part, `> swim.out`, means that the output is written to the `swim.out` file. This file can be found in the chapter directory `source_code/4`. This output of the program was used previously to write the analysis of the swim preference problem.

Playing chess example

We will again use the examples from Chapter 2, *Naive Bayes*, and Chapter 3, *Decision Tree*, as follows:

Temperature	Wind	Sunshine	Play
Cold	Strong	Cloudy	No
Warm	Strong	Cloudy	No
Warm	None	Sunny	Yes
Hot	None	Sunny	No
Hot	Breeze	Cloudy	Yes
Warm	Breeze	Sunny	Yes
Cold	Breeze	Cloudy	No
Cold	None	Sunny	Yes
Hot	Strong	Cloudy	Yes
Warm	None	Cloudy	Yes
Warm	Strong	Sunny	?

However, we would like to use a random forest consisting of four random decision trees to find the result of the classification.

Analysis

We are given $M=4$ variables from which a feature can be classified. Thus, we choose the maximum number of the variables considered at the node to:

$$m = \min(M, \text{math.ceil}(2 * \text{math.sqrt}(M))) = \min(M, \text{math.ceil}(2 * \text{math.sqrt}(4))) = 4$$

We are given the following features:

```
[['Cold', 'Strong', 'Cloudy', 'No'], ['Warm', 'Strong', 'Cloudy', 'No'],
['Warm', 'None', 'Sunny',
'Yes'], ['Hot', 'None', 'Sunny', 'No'], ['Hot', 'Breeze', 'Cloudy', 'Yes'],
['Warm', 'Breeze',
'Sunny', 'Yes'], ['Cold', 'Breeze', 'Cloudy', 'No'], ['Cold', 'None',
'Sunny', 'Yes'], ['Hot', 'Strong', 'Cloudy', 'Yes'], ['Warm', 'None',
'Cloudy', 'Yes']]
```

When constructing a random decision tree as part of a random forest, we will choose only a subset of these features randomly, together with their replacements.

Random forest construction

We will construct a random forest that will consist of four random decision trees.

Construction of random decision tree number 0

We are given 10 features as input data. Of these, we choose all features randomly with their replacements for the construction of this random decision tree:

```
[['Warm', 'Strong', 'Cloudy', 'No'], ['Cold', 'Breeze', 'Cloudy', 'No'],
['Cold', 'None', 'Sunny', 'Yes'], ['Cold', 'Breeze', 'Cloudy', 'No'],
['Hot', 'Breeze', 'Cloudy', 'Yes'], ['Warm', 'Strong', 'Cloudy', 'No'],
['Hot', 'Breeze', 'Cloudy', 'Yes'], ['Hot', 'Breeze', 'Cloudy', 'Yes'],
['Cold', 'Breeze', 'Cloudy', 'No'], ['Warm', 'Breeze', 'Sunny', 'Yes']]
```

We start the construction with the root node to create the first node of the tree. We would like to add children to the [root] node.

We have the following variables available: ['Temperature', 'Wind', 'Sunshine']. As there are fewer of them than the $m=4$ parameter, we consider all of them. Of these variables, the one with the highest information gain is the

Temperature variable. Therefore, we will branch the node further on this variable. We will also remove this variable from the list of available variables for the children of the current node. Using the Temperature variable, we partition the data in the current node as follows:

- Partition for Temperature=Cold: [['Cold', 'Breeze', 'Cloudy', 'No'], ['Cold', 'None', 'Sunny', 'Yes'], ['Cold', 'Breeze', 'Cloudy', 'No'], ['Cold', 'Breeze', 'Cloudy', 'No']]
- Partition for Temperature=Warm: [['Warm', 'Strong', 'Cloudy', 'No'], ['Warm', 'Strong', 'Cloudy', 'No'], ['Warm', 'Breeze', 'Sunny', 'Yes']]
- Partition for Temperature=Hot: [['Hot', 'Breeze', 'Cloudy', 'Yes'], ['Hot', 'Breeze', 'Cloudy', 'Yes'], ['Hot', 'Breeze', 'Cloudy', 'Yes']]

Now, given the partitions, let's create the branches and the child nodes.

We add a child node, [Temperature=Cold], to the [root] node. This branch classifies four features: [['Cold', 'Breeze', 'Cloudy', 'No'], ['Cold', 'None', 'Sunny', 'Yes'], ['Cold', 'Breeze', 'Cloudy', 'No'], and ['Cold', 'Breeze', 'Cloudy', 'No']].

We would like to add children to the [Temperature=Cold] node.

We have the following variables available: ['Wind', 'Sunshine']. As there are fewer of these than the $m=3$ parameter, we consider both of them. The one with the highest information gain is the Wind variable. Therefore, we will branch the node further on this variable. We will also remove this variable from the list of available variables for the children of the current node. Using the water Wind variable, we partition the data in the current node as follows:

- Partition for Wind=None: [['Cold', 'None', 'Sunny', 'Yes']]
- Partition for Wind=Breeze: [['Cold', 'Breeze', 'Cloudy', 'No'], ['Cold', 'Breeze', 'Cloudy', 'No'], ['Cold', 'Breeze', 'Cloudy', 'No']]

Now, given the partitions, let's create the branches and the child nodes.

We add a child node, `[Wind=None]`, to the `[Temperature=Cold]` node. This branch classifies a single feature: `[['Cold', 'None', 'Sunny', 'Yes']]`.

We would like to add children to the `[Wind=None]` node.

We have the following variable: `available['Sunshine']`. As there are fewer of these than the `m=3` parameter, we consider both of them. The one with the highest information gain is the `Sunshine` variable. Therefore, we will branch the node further on this variable. We will also remove this variable from the list of available variables for the children of the current node. For the chosen variable, `Sunshine`, all the remaining features have the same value: `Sunny`. So, we end the branch with a leaf node, adding `[Play=Yes]`.

We add a child node, `[Wind=Breeze]`, to the `[Temperature=Cold]` node. This branch classifies three features: `[['Cold', 'Breeze', 'Cloudy', 'No'], ['Cold', 'Breeze', 'Cloudy', 'No'], and ['Cold', 'Breeze', 'Cloudy', 'No']]`.

We would like to add children to the `[Wind=Breeze]` node.

We have the following variable available: `['Sunshine']`. As there are fewer of these than the `m=3` parameter, we consider both of them. The one with the highest information gain is the `Sunshine` variable. Therefore, we will branch the node further on this variable. We will also remove this variable from the list of available variables for the children of the current node. For the chosen variable, `Sunshine`, all the remaining features have the same value: `Cloudy`. So, we end the branch with a leaf node, adding `[Play=No]`.

Now, we have added all the children nodes for the `[Temperature=Cold]` node.

We add a child node, `[Temperature=Warm]`, to the `[root]` node. This branch classifies three features: `[['Warm', 'Strong', 'Cloudy', 'No'], ['Warm', 'Strong', 'Cloudy', 'No'], and ['Warm', 'Breeze', 'Sunny', 'Yes']]`.

We would like to add children to the `[Temperature=Warm]` node.

The available variables that we still have left are ['Wind', 'Sunshine']. As there are fewer of these than the $m=3$ parameter, we consider both of them. The one with the highest information gain is the Wind variable. Thus, we will branch the node further on this variable. We will also remove this variable from the list of available variables for the children of the current node. Using the Wind variable, we partition the data in the current node, where each partition of the data will be for one of the new branches from the current node, [Temperature=Warm]. We have the following partitions:

- Partition for Wind=Breeze: [['Warm', 'Breeze', 'Sunny', 'Yes']]
- Partition for Wind=Strong: [['Warm', 'Strong', 'Cloudy', 'No'], ['Warm', 'Strong', 'Cloudy', 'No']]

Now, given the partitions, let's form the branches and the child nodes.

We add a child node, [Wind=Breeze], to the [Temperature=Warm] node. This branch classifies a single feature: [['Warm', 'Breeze', 'Sunny', 'Yes']].

We would like to add children to the [Wind=Breeze] node.

We have the following variable available: ['Sunshine']. As there are fewer of these than the $m=3$ parameter, we consider both of them. The one with the highest information gain is the Sunshine variable. Therefore, we will branch the node further on this variable. We will also remove this variable from the list of available variables for the children of the current node. For the chosen variable, Sunshine, all the remaining features have the same value: Sunny. So, we end the branch with a leaf node, adding [Play=Yes].

We add a child node, [Wind=Strong], to the [Temperature=Warm] node. This branch classifies two features: [['Warm', 'Strong', 'Cloudy', 'No'], and ['Warm', 'Strong', 'Cloudy', 'No']]

We would like to add children to the [Wind=Strong] node.

We have the following variable available: ['Sunshine']. As there are fewer of these than the $m=3$ parameter, we consider both of them. The one with the highest information gain is the Sunshine variable. Therefore, we will branch the node further on this variable. We will also remove this variable from the list of available variables for the children of the current node. For the chosen variable, Sunshine, all the remaining features have the same value: Cloudy. So, we end the branch with a leaf node, adding [Play=No].

Now, we have added all the children nodes to the [Temperature=Warm] node.

We add a child node, [Temperature=Hot], to the [root] node. This branch classifies three features: [['Hot', 'Breeze', 'Cloudy', 'Yes'], ['Hot', 'Breeze', 'Cloudy', 'Yes'], and ['Hot', 'Breeze', 'Cloudy', 'Yes']].

We would like to add children to the [Temperature=Hot] node.

We have the following variables available: ['Wind', 'Sunshine']. As there are fewer of these than the $m=3$ parameter, we consider both of them. The one with the highest information gain is the Wind variable. Therefore, we will branch the node further on this variable. We will also remove this variable from the list of available variables for the children of the current node. For the chosen variable, Wind, all the remaining features have the same value: Breeze. So, we end the branch with a leaf node, adding [Play=Yes].

Now, we have added all the children nodes to the [root] node.

Construction of random decision trees numbers 1, 2, and 3

We construct the next three trees in a similar fashion. We should note that, since the construction is random, a reader who performs another correct construction may arrive at a different construction. However, if there are a sufficient number of random decision trees in a random forest, then the result of the classification should be very similar across all the random constructions.

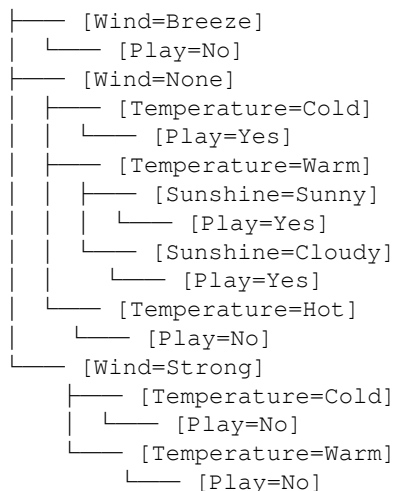
The full construction can be found in the program output in the `source_code/4/chess.out` file.

Constructed random forest:

```
Tree 0:
Root
├── [Temperature=Cold]
│   ├── [Wind=None]
│   │   └── [Play=Yes]
│   └── [Wind=Breeze]
│       └── [Play=No]
├── [Temperature=Warm]
│   ├── [Wind=Breeze]
│   │   └── [Play=Yes]
│   └── [Wind=Strong]
│       └── [Play=No]
└── [Temperature=Hot]
    └── [Play=Yes]
```

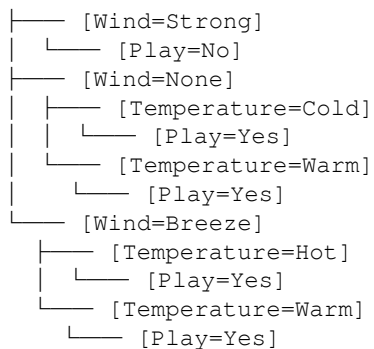
Tree 1:

Root



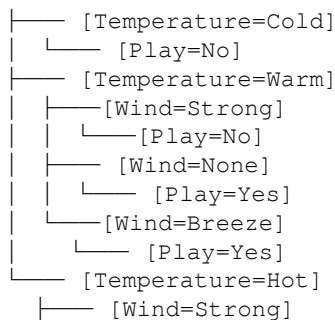
Tree 2:

Root



Tree 3:

Root



```

|   └── [Play=Yes]
└── [Wind=Breeze]
    └── [Play=Yes]

```

The total number of trees in the random forest=4.

The maximum number of the variables considered at the node is $m=4$.

Classification

Given the random forest constructed, we classify the ['Warm', 'Strong', 'Sunny', '?'] feature:

- **Tree 0 votes for the class:** No
- **Tree 1 votes for the class:** No
- **Tree 2 votes for the class:** No
- **Tree 3 votes for the class:** No

The class with the maximum number of votes is No. Thus, the constructed random forest classifies the feature ['Warm', 'Strong', 'Sunny', '?'] into the class No.

Input:

To perform the preceding analysis, we use a program implemented earlier in this chapter. First, we insert the data from the table into the following CSV file:

```

# source_code/4/chess.csv
Temperature,Wind,Sunshine,Play
Cold,Strong,Cloudy,No
Warm,Strong,Cloudy,No
Warm,None,Sunny,Yes
Hot,None,Sunny,No
Hot,Breeze,Cloudy,Yes
Warm,Breeze,Sunny,Yes
Cold,Breeze,Cloudy,No
Cold,None,Sunny,Yes
Hot,Strong,Cloudy,Yes
Warm,None,Cloudy,Yes
Warm,Strong,Sunny,?

```

Output:

We produce the output by executing the following command line:

```
$ python random_forest.py chess.csv 4 2 > chess.out
```

The number 4 here means that we want to construct four decision trees, and 2 is the level of verbosity of the program that includes the explanations of how a tree is constructed. The last part, `> chess.out`, means that the output is written to the `chess.out` file. This file can be found in the chapter directory `source_code/4`. We will not put all the output here, as it is very large and repetitive. Instead, some of it was included in the preceding analysis and in the construction of a random forest.

Going shopping – overcoming data inconsistencies with randomness and measuring the level of confidence

We take the problem from the previous chapter. We have the following data relating to the shopping preferences of our friend, Jane:

Temperature	Rain	Shopping
Cold	None	Yes
Warm	None	No
Cold	Strong	Yes
Cold	None	No
Warm	Strong	No
Warm	None	Yes
Cold	None	?

In the previous chapter, decision trees were not able to classify the feature `(Cold, None)`. So, this time, we would like to establish whether Jane would go shopping if the temperature was cold and there was no rain using the random forest algorithm.

Analysis

To perform an analysis using the random forest algorithm, we use the program implemented.

Input:

We insert the data from the table into the following CSV file:

```
# source_code/4/shopping.csv
Temperature,Rain,Shopping
Cold,None,Yes
Warm,None,No
Cold,Strong,Yes
Cold,None,No
Warm,Strong,No
Warm,None,Yes
Cold,None,?
```

Output:

We want to use a slightly higher number of trees than we used in the previous examples and explanations to obtain more accurate results. We want to construct a random forest with 20 trees with low-verbosity output – level 0. Thus, we undertake execution in a terminal:

```
$ python random_forest.py shopping.csv 20 0
***Classification***
Feature: ['Cold', 'None', '?']
Tree 0 votes for the class: Yes
Tree 1 votes for the class: No
Tree 2 votes for the class: No
Tree 3 votes for the class: No
Tree 4 votes for the class: No
Tree 5 votes for the class: Yes
Tree 6 votes for the class: Yes
Tree 7 votes for the class: Yes
Tree 8 votes for the class: No
Tree 9 votes for the class: Yes
Tree 10 votes for the class: Yes
Tree 11 votes for the class: Yes
Tree 12 votes for the class: Yes
Tree 13 votes for the class: Yes
Tree 14 votes for the class: Yes
Tree 15 votes for the class: Yes
Tree 16 votes for the class: Yes
Tree 17 votes for the class: No
```

```
Tree 18 votes for the class: No
Tree 19 votes for the class: No
The class with the maximum number of votes is 'Yes'. Thus the constructed
random forest classifies the feature ['Cold', 'None', '?'] into the class
'Yes'.
```

However, we should note that only 12 out of the 20 trees voted for the answer *Yes*. Therefore, although we have a definite answer, it might not be that certain, similar to the results we got with an ordinary decision tree. However, unlike in decision trees, where an answer was not produced because of data inconsistency, here, we have an answer.

Furthermore, by measuring the strength of the voting power for each individual class, we can measure the level of confidence that the answer is correct. In this case, the feature ['Cold', 'None', '?'] belongs to the *Yes* class with a confidence level of 12/20, or 60%. To determine the level of certainty of the classification more precisely, an even larger ensemble of random decision trees would be required.

Summary

In this chapter, we learned that a random forest is a set of decision trees, where each tree is constructed from a sample chosen randomly from the initial data. This process is called **bootstrap aggregating**. Its purpose is to reduce variance and bias in classifications made by a random forest. Bias is further reduced during the construction of a decision tree by considering only a random subset of the variables for each branch of the tree.

We also learned that once a random forest is constructed, the result of the classification of a random forest is a majority vote from among all the trees in a random forest. The level of the majority also determines the level of confidence that the answer is correct.

Since random forests consist of decision trees, it is good to use them for every problem where a decision tree is a good choice. As random forests reduce the bias and variance that exists in decision tree classifiers, they outperform decision tree algorithms.

In the next chapter, we will learn the technique of clustering data into similar clusters. We will also exploit this technique to classify data into one of the created clusters.

Problems

Problem 1: Let's take the example of playing chess from Chapter 2, *Naive Bayes*. How would you classify a (Warm, Strong, Spring, ?) data sample according to the random forest algorithm?

Temperature	Wind	Season	Play
Cold	Strong	Winter	No
Warm	Strong	Autumn	No
Warm	None	Summer	Yes
Hot	None	Spring	No
Hot	Breeze	Autumn	Yes
Warm	Breeze	Spring	Yes
Cold	Breeze	Winter	No
Cold	None	Spring	Yes
Hot	Strong	Summer	Yes
Warm	None	Autumn	Yes
Warm	Strong	Spring	?

Problem 2: Would it be a good idea to use only one tree and a random forest? Justify your answer.

Problem 3: Can cross-validation improve the results of the classification by the random forest? Justify your answer.

Analysis

Problem 1: We run the program to construct the random forest and classify the feature (Warm, Strong, Spring).

Input:

```
source_code/4/chess_with_seasons.csv
Temperature,Wind,Season,Play
Cold,Strong,Winter,No
Warm,Strong,Autumn,No
Warm,None,Summer,Yes
Hot,None,Spring,No
Hot,Breeze,Autumn,Yes
Warm,Breeze,Spring,Yes
```

```

Cold,Breeze,Winter,No
Cold,None, Spring, Yes
Hot, Strong, Summer, Yes
Warm, None, Autumn, Yes
Warm, Strong, Spring, ?

```

Output:

We construct four trees in a random forest, as follows:

```

$ python random_forest.py chess_with_seasons.csv 4 2 >
chess_with_seasons.out

```

The whole construction and the analysis are stored in the `source_code/4/chess_with_seasons.out` file. Your construction may differ because of the randomness involved. From the output, we extract the random forest graph, consisting of random decision trees, given the random numbers generated during our run.

Executing the preceding command again will most likely result in a different output and a different random forest graph. Yet there is a high probability that the results of the classification will be similar because of the multiplicity of the random decision trees and their voting power combined. The classification by one random decision tree may be subject to significant variance. However, the majority vote combines the classification from all the trees, thus reducing the variance. To verify your understanding, you can compare your classification results with the classification by the following random forest graph.

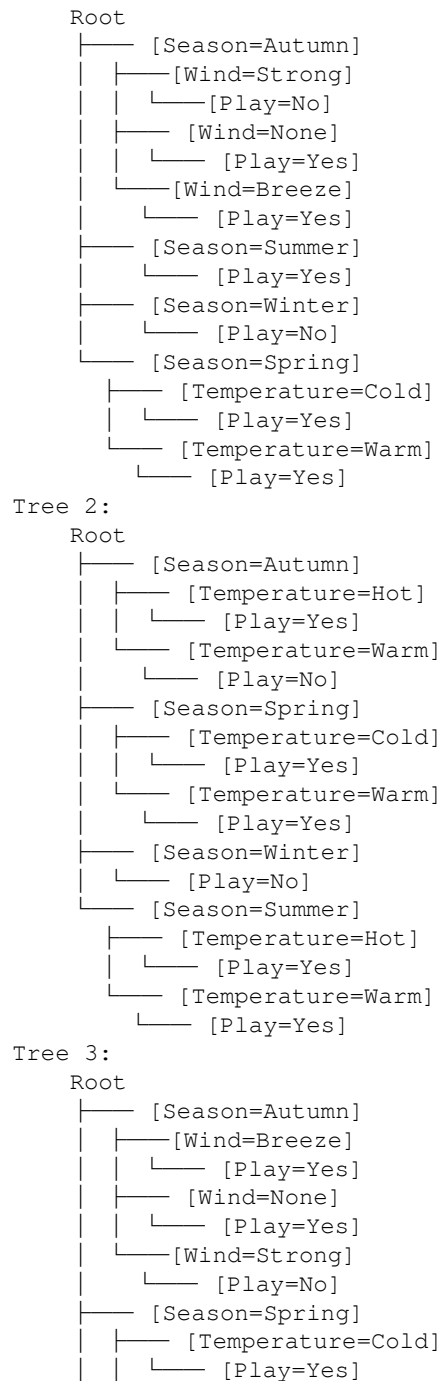
Random forest graph and classification:

Let's have a look at the output of the random forest graph and the classification of the feature:

```

Tree 0:
  Root
  |
  |--- [Wind=None]
  |   |
  |   |--- [Temperature=Cold]
  |   |   |
  |   |   |--- [Play=Yes]
  |   |   |
  |   |   |--- [Temperature=Warm]
  |   |   |   |
  |   |   |   |--- [Season=Autumn]
  |   |   |   |   |
  |   |   |   |   |--- [Play=Yes]
  |   |   |   |   |
  |   |   |   |   |--- [Season=Summer]
  |   |   |   |   |   |
  |   |   |   |   |   |--- [Play=Yes]
  |   |   |
  |   |--- [Wind=Strong]
  |   |   |
  |   |   |--- [Temperature=Cold]
  |   |   |   |
  |   |   |   |--- [Play=No]
  |   |   |
  |   |--- [Temperature=Warm]
  |   |   |
  |   |   |--- [Play=No]
Tree 1:

```



```

|   |   [Temperature=Warm]
|   |   |   [Play=Yes]
|   |   |   |   [Season=Winter]
|   |   |   |   |   [Play=No]
|   |   |   |   |   |   [Season=Summer]
|   |   |   |   |   |   |   [Play=Yes]

```

The total number of trees in the random forest=4.

The maximum number of the variables considered at the node is $m=4$.

Classification

Feature: ['Warm', 'Strong', 'Spring', '?']

Tree 0 votes for the class: No

Tree 1 votes for the class: Yes

Tree 2 votes for the class: Yes

Tree 3 votes for the class: Yes

The class with the maximum number of votes is 'Yes'. Thus the constructed random forest classifies the feature ['Warm', 'Strong', 'Spring', '?'] into the class 'Yes'.

Problem 2: When we construct a tree in a random forest, we only use a random subset of the data, with replacements. This is to eliminate classifier bias toward certain features. However, if we use only one tree, that tree may happen to contain features with bias and might be missing an important feature to enable it to provide an accurate classification. So, a random forest classifier with one decision tree would likely lead to a very poor classification. Therefore, we should construct more decision trees in a random forest to benefit from reduced bias and variance in the classification.

Problem 3: During cross-validation, we divide the data into training and testing data. Training data is used to train the classifier, and testing data is used to evaluate which parameters or methods would be the best fit for improving classification. Another advantage of cross-validation is the reduction in bias, because we only use partial data, thereby decreasing the chance of overfitting the specific dataset.

However, in a decision forest, we address problems that cross-validation addresses in an alternative way. Each random decision tree is constructed only on the subset of the data—reducing the chance of overfitting. In the end, classification is the combination of results from each of these trees. The best decision, in the end, is not made by tuning the parameters on a test dataset, but by taking the majority vote of all the trees with reduced bias.

Hence, cross-validation for a decision forest algorithm would not be of much use as it is already intrinsic within the algorithm.

5

Clustering into K Clusters

Clustering is a technique for dividing data into clusters, with the same features in the same cluster.

In this chapter, we will cover the following topics:

- How to use the *k*-means clustering algorithm, using an example involving household incomes
- How to classify features by clustering them first with the features, along with the known classes, using an example of gender classification
- How to implement the *k*-means clustering algorithm in Python in the *Implementation of k-means clustering algorithm* section
- An example of house ownership and how to choose an appropriate number of clusters for your analysis
- How to use the example of house ownership to scale a given set of numerical data appropriately to improve the accuracy of classification by using a clustering algorithm
- An understanding of how different numbers of clusters alter the meaning of the dividing line between those clusters, using an example of document clustering

Household incomes – clustering into k clusters

For example, let's look at households whose annual earnings in USD are \$40,000, \$55,000, \$70,000, \$100,000, \$115,000, \$130,000 and \$135,000. Then, if we were to combine those households into two clusters, taking their earnings as a measure of similarity, the first cluster would have those households earning 40 k, 55 k, and 70 k, while the second cluster would include those households earning 100 k, 115 k, 130 k, and 135 k.

This is because 40k and 135k are furthest away from each other so, because we want to have two clusters, these have to be in different clusters. 55 K is closer to 40 k than to 135 k, so 40 k and 55 k will be in the same cluster. Similarly, 130 k and 135 k will be in the same cluster. 70 K is closer to 40 k and 55 k than to 130 K and 135 k, so 70 k should be in the cluster with 40 k and 55 k. 115 K is closer to 130 k and 135 k than to the first cluster containing 40 k, 55 k, and 70 k, so it will be in the second cluster. Finally, 100 k is closer to the second cluster, containing 115 k, 130 k, and 135 k, so it will be located there. Therefore, the first cluster will include 40 k, 55 k, and 70 households. The second cluster will include the 100 k, 115 k, 130 k, and 135 k households.

Clustering the features of groups with similar properties and assigning a cluster to a feature is a form of classification. It is up to a data scientist to interpret the result of the clustering and what classification it induces. Here, the cluster containing the households with annual incomes of USD 40 k, 55 k, and 70 k represents a class of households with a low income. The second cluster, including households with an annual income of USD 100 k, 115 k, 130 k, and 135 k, represents a class of households with a high income.

We clustered the households into the two clusters in an informal way based on intuition and common sense. There are clustering algorithms that cluster data according to precise rules. These algorithms include a fuzzy c-means clustering algorithm, a hierarchical clustering algorithm, a Gaussian (EM) clustering algorithm, a quality threshold clustering algorithm, and a k -means clustering algorithm, which is the focus of this chapter.

K-means clustering algorithm

The k -means clustering algorithm classifies given points into k groups in such a way that the distance between members of the same group is minimized.

The k -means clustering algorithm determines the initial k -centroids (points in a cluster center)—one for each cluster. Then, each feature is classified into the cluster whose centroid is closest to that feature. After classifying all the features, we have formed an initial k clusters.

For each cluster, we recomputed the centroid to be the average of the points in that cluster. After we have moved the centroids, we recompute the classes again. The features in the classes may change. In this case, we have to recompute the centroids again. If the centroids no longer move, then the k -means clustering algorithm terminates.

Picking the initial k -centroids

We could pick the initial k -centroids to be any of the k features in the data to be classified. But, ideally, we would like to pick points that belong to different clusters from the very outset. Therefore, we may want to aim to maximize their mutual distance in a certain way. Simplifying the process, we could pick the first centroid to be any point from the features. The second could be the one that is furthest from the first. The third could be the one that is furthest from both the first and second, and so on.

Computing a centroid of a given cluster

A centroid of a cluster is just an average of the points in that cluster. If a cluster contains one-dimensional points with the coordinates $x_1, x_2, \dots, x_n$, then the centroid of that cluster

would be $(\frac{1}{n}) * (x_1 + x_2 + \dots + x_n)$. If a cluster contains two-dimensional points with the coordinates $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$, then the x -coordinate of the centroid of the cluster would have the value $(1/n) * (x_1 + x_2 + \dots + x_n)$, and the y -coordinate would have the value $(\frac{1}{n}) * (y_1 + y_2 + \dots + y_n)$.

This computation generalizes easily to higher dimensions. If the value of the higher-dimensional features for the x -coordinate are $x_1, x_2, \dots, x_n$, then the value of the x -coordinate for the centroid is $(\frac{1}{n}) * (x_1 + x_2 + \dots + x_n)$.

Using the k -means clustering algorithm on the household income example

We will apply the k -clustering algorithm to the household income example. In the beginning, we have households with incomes of \$40,000, \$55,000, \$70,000, \$100,000, \$115,000, \$130,000 and \$135,000.

The first centroid to be picked up can be any feature, for example, \$70,000. The second centroid should be the feature that is furthest from the first one, that is 135 k, since 135 k minus 70 k is 65 k, which is the greatest difference between any other feature and 70 k. Thus, 70 k is the centroid of the first cluster, while 135 k is the centroid of the second cluster.

Now, by taking the difference, 40 k, 55 k, 70 k, and 100 k are closer to 70 k than to 135 k, so they will be in the first cluster, while 115 k, 130 k, and 135 k are closer to 135 k than to 70 k, so they will be in the second cluster.

After we have classified the features according to the initial centroids, we recompute the centroids. The centroid of the first cluster is as follows:

$$\begin{aligned} & (1/4) * (40k + 55k + 70k + 100k) \\ & = (1/4) * 265k \\ & = 66.25k \end{aligned}$$

The centroid of the second cluster is as follows:

$$\begin{aligned} & (1/3) * (115k + 130k + 135k) \\ & = (1/3) * 380k \sim 126.66k \end{aligned}$$

Using the new centroids, we reclassify the features as follows:

- The first cluster containing the centroid 66.25 k will contain the features 40 k, 55 k, and 70 k
- The second cluster containing the centroid 126.66 k will contain the features 100 k, 115 k, 130 k, and 135 k

We notice that the 100 k feature moved from the first cluster to the second, since it is now closer to the centroid of the second cluster (*distance* $|100\text{ k} - 126.66\text{ k}| = 26.66\text{ k}$) than to the centroid of the first cluster (*distance* $|100\text{ k} - 66.25\text{ k}| = 33.75\text{ k}$). Since the features in the clusters have changed, we have to recompute the centroids again.

The centroid of the first cluster is as follows:

$$\begin{aligned} & (1/3) * (40k + 55k + 70k) \\ & = (1/3) * 165k \\ & = 55k \end{aligned}$$

The centroid of the second cluster is as follows:

$$\begin{aligned} & (1/4) * (100k + 115k + 130k + 135k) \\ & = (1/4) * 480k \\ & = 120k \end{aligned}$$

Using these centroids, we reclassify the features into the clusters. The first centroid, 55 k, will contain the features 40 k, 55 k, and 70 k. The second centroid, 120 k, will contain the features 100 k, 115 k, 130 k, and 135 k. Thus, when the centroids were updated, the clusters did not change. Hence, their centroids will remain the same.

Therefore, the algorithm terminates with the two clusters: the first cluster having the features 40 k, 55 k, and 70 k, and the second cluster having the features 100 k, 115 k, 130 k, and 135 k.

Gender classification – clustering to classify

The following data is taken from the gender classification example, *Problem 6, Chapter 2, Naïve Bayes*:

Height in cm	Weight in kg	Hair length	Gender
180	75	Short	Male
174	71	Short	Male
184	83	Short	Male
168	63	Short	Male
178	70	Long	Male
170	59	Long	Female
164	53	Short	Female
155	46	Long	Female
162	52	Long	Female
166	55	Long	Female
172	60	Long	?

To simplify matters, we will remove the column entitled **Hair length**. We will also remove the column entitled **Gender**, since we would like to cluster the people in the table based on their height and weight. We would like to establish whether the eleventh person in the table is more likely to be a man or a woman using clustering:

Height in cm	Weight in kg
180	75
174	71
184	83
168	63
178	70
170	59
164	53
155	46
162	52
166	55
172	60

Analysis

We may apply scaling to the initial data, but to simplify matters, we will use the unscaled data in the algorithm. We will cluster the data we have into two clusters, since there are two possibilities for gender—male or female. Then, we will aim to classify a person with a height of 172 cm and a weight of 60 kg as being more likely be a man if, and only if, there are more men in that cluster. The clustering algorithm is a very efficient technique. Thus, classifying in this way is very fast, especially if there are a lot of features to classify.

So, let's apply the k -means clustering algorithm to the data we have. First, we pick the initial centroids. Let the first centroid, for example, be a person with a height of 180 cm and a weight of 75 kg, denoted in a vector as $(180,75)$. The point that is furthest away from $(180,75)$ is $(155,46)$. So that will be the second centroid.

The points that are closer to the first centroid $(180,75)$ by taking Euclidean distance are $(180,75)$, $(174,71)$, $(184,83)$, $(168,63)$, $(178,70)$, $(170,59)$, and $(172,60)$. So these points will be in the first cluster. The points that are closer to the second centroid $(155,46)$ are $(155,46)$, $(164,53)$, $(162,52)$, and $(166,55)$. So these points will be in the second cluster. The current situation involving these two clusters is displayed in the following diagram:

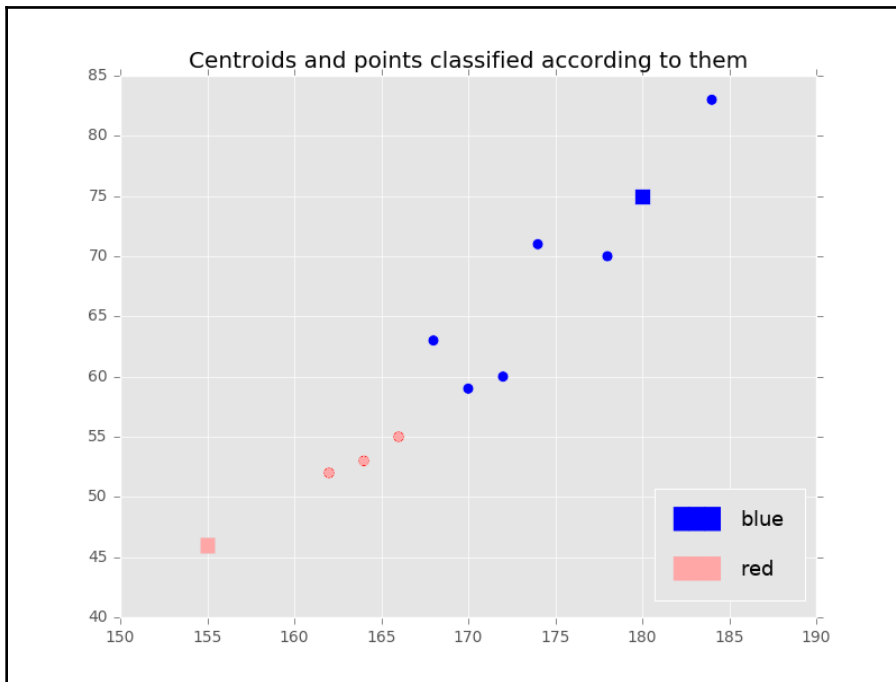


Figure 5.1: Clustering of people according to their height and weight

Let's recompute the centroids of the clusters. The blue cluster with the features (180,75), (174,71), (184,83), (168,63), (178,70), (170,59), and (172,60) will have the centroid $((180+174+184+168+178+170+172)/7, (75+71+83+63+70+59+60)/7) \sim (175.14, 68.71)$.

The red cluster with the features (155,46), (164,53), (162,52), and (166,55) will have the centroid $((155+164+162+166)/4, (46+53+52+55)/4) = (161.75, 51.5)$.

Reclassifying the points using the new centroid, the classes of the points do not change. The blue cluster will have the points (180,75), (174,71), (184,83), (168,63), (178,70), (170,59), and (172,60). The red cluster will have the points (155,46), (164,53), (162,52), and (166,55). Therefore, the clustering algorithm terminates with clusters, as depicted in the following diagram:

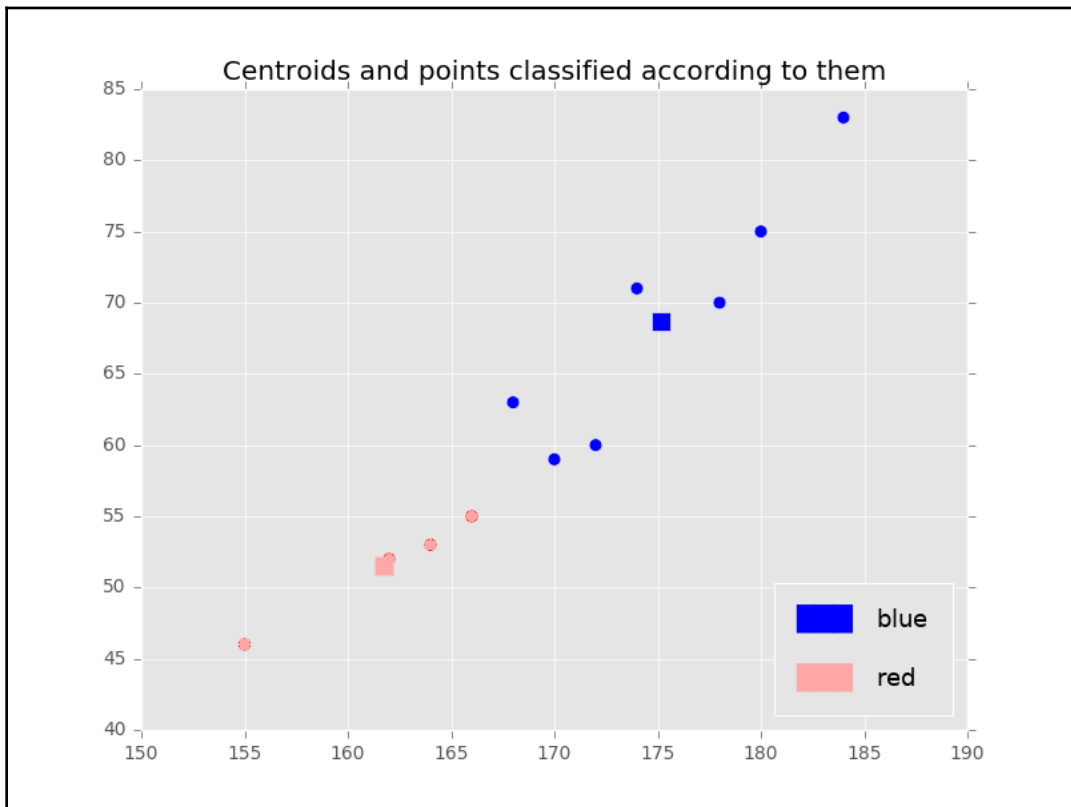


Figure 5.2: Clustering of people according to their height and weight

Now we would like to classify the instance (172,60) as to whether that person is a male or a female. The instance (172,60) is in the blue cluster, so it is similar to the features in the blue cluster. Are the remaining features in the blue cluster more likely male or female? Five out of six features are male, while only one is female. Since the majority of the features are male in the blue cluster and the person (172,60) is in the blue cluster as well, we classify a person with a height of 172 cm and a weight of 60 kg as male.

Implementation of the k -means clustering algorithm

We will now implement the k -means clustering algorithm. It takes a CSV file as input with one data item per line. A data item is converted into a point. The algorithms classify these points into the specified number of clusters. In the end, the clusters are visualized on a graph using the matplotlib library:

```
# source_code/5/k-means_clustering.py
import math
import imp
import sys
import matplotlib.pyplot as plt
import matplotlib
import sys
sys.path.append('../common')
import common # noqa
matplotlib.style.use('ggplot')

# Returns k initial centroids for the given points.
def choose_init_centroids(points, k):
    centroids = []
    centroids.append(points[0])
    while len(centroids) < k:
        # Find the centroid that with the greatest possible distance
        # to the closest already chosen centroid.
        candidate = points[0]
        candidate_dist = min_dist(points[0], centroids)
        for point in points:
            dist = min_dist(point, centroids)
            if dist > candidate_dist:
                candidate = point
                candidate_dist = dist
        centroids.append(candidate)
    return centroids
```

```

# Returns the distance of a point from the closest point in points.
def min_dist(point, points):
    min_dist = euclidean_dist(point, points[0])
    for point2 in points:
        dist = euclidean_dist(point, point2)
        if dist < min_dist:
            min_dist = dist
    return min_dist

# Returns an Euclidean distance of two 2-dimensional points.
def euclidean_dist((x1, y1), (x2, y2)):
    return math.sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2))

# PointGroup is a tuple that contains in the first coordinate a 2d point
# and in the second coordinate a group which a point is classified to.
def choose_centroids(point_groups, k):
    centroid_xs = [0] * k
    centroid_ys = [0] * k
    group_counts = [0] * k
    for ((x, y), group) in point_groups:
        centroid_xs[group] += x
        centroid_ys[group] += y
        group_counts[group] += 1
    centroids = []
    for group in range(0, k):
        centroids.append((
            float(centroid_xs[group]) / group_counts[group],
            float(centroid_ys[group]) / group_counts[group]))
    return centroids

# Returns the number of the centroid which is closest to the point.
# This number of the centroid is the number of the group where
# the point belongs to.
def closest_group(point, centroids):
    selected_group = 0
    selected_dist = euclidean_dist(point, centroids[0])
    for i in range(1, len(centroids)):
        dist = euclidean_dist(point, centroids[i])
        if dist < selected_dist:
            selected_group = i
            selected_dist = dist
    return selected_group

# Reassigns the groups to the points according to which centroid
# a point is closest to.
def assign_groups(point_groups, centroids):
    new_point_groups = []
    for (point, group) in point_groups:

```

```

        new_point_groups.append(
            (point, closest_group(point, centroids)))
    return new_point_groups

# Returns a list of pointgroups given a list of points.
def points_to_point_groups(points):
    point_groups = []
    for point in points:
        point_groups.append((point, 0))
    return point_groups

# Clusters points into the k groups adding every stage
# of the algorithm to the history which is returned.
def cluster_with_history(points, k):
    history = []
    centroids = choose_init_centroids(points, k)
    point_groups = points_to_point_groups(points)
    while True:
        point_groups = assign_groups(point_groups, centroids)
        history.append((point_groups, centroids))
        new_centroids = choose_centroids(point_groups, k)
        done = True
        for i in range(0, len(centroids)):
            if centroids[i] != new_centroids[i]:
                done = False
                break
        if done:
            return history
        centroids = new_centroids

# Program start
csv_file = sys.argv[1]
k = int(sys.argv[2])
everything = False
# The third argument sys.argv[3] represents the number of the step of the
# algorithm starting from 0 to be shown or "last" for displaying the last
# step and the number of the steps.
if sys.argv[3] == "last":
    everything = True
else:
    step = int(sys.argv[3])

data = common.csv_file_to_list(csv_file)
points = data_to_points(data) # Represent every data item by a point.
history = cluster_with_history(points, k)
if everything:
    print "The total number of steps:", len(history)
    print "The history of the algorithm:"

```

```

    (point_groups, centroids) = history[len(history) - 1]
    # Print all the history.
    print_cluster_history(history)
    # But display the situation graphically at the last step only.
    draw(point_groups, centroids)
else:
    (point_groups, centroids) = history[step]
    print "Data for the step number", step, ":"
    print point_groups, centroids
    draw(point_groups, centroids)

```

Input data from gender classification

We save data from the gender classification example into the CSV file as follows:

```

# source_code/5/persons_by_height_and_weight.csv
180,75
174,71
184,83
168,63
178,70
170,59
164,53
155,46
162,52
166,55
172,60

```

Program output for gender classification data

We run the program, implementing the k -means clustering algorithm on the data from the gender classification example. The numerical argument 2 means that we would like to cluster the data into two clusters, as can be seen in the following block of code:

```

$ python k-means_clustering.py persons_by_height_weight.csv 2 last
The total number of steps: 2
The history of the algorithm:
Step number 0: point_groups = [((180.0, 75.0), 0), ((174.0, 71.0), 0),
((184.0, 83.0), 0), ((168.0, 63.0), 0), ((178.0, 70.0), 0), ((170.0, 59.0),
0), ((164.0, 53.0), 1), ((155.0, 46.0), 1), ((162.0, 52.0), 1), ((166.0,
55.0), 1), ((172.0, 60.0), 0)]
centroids = [(180.0, 75.0), (155.0, 46.0)]
Step number 1: point_groups = [((180.0, 75.0), 0), ((174.0, 71.0), 0),
((184.0, 83.0), 0), ((168.0, 63.0), 0), ((178.0, 70.0), 0), ((170.0, 59.0),
0), ((164.0, 53.0), 1), ((155.0, 46.0), 1), ((162.0, 52.0), 1), ((166.0,

```

```
55.0), 1), ((172.0, 60.0), 0)]
centroids = [(175.14285714285714, 68.71428571428571), (161.75, 51.5)]
```

The program also outputs a graph, visible in *Figure 5.2*. The `last` parameter means that we would like the program to perform clustering until the final step. If we would like to display only the first step (step 0), we could change the last to 0 in order to run, as can be seen in the following code:

```
$ python k-means_clustering.py persons_by_height_weight.csv 2 0
```

Upon execution of the program, we would get a graph of the clusters and their centroids in the initial step, as in *Figure 5.1*.

House ownership – choosing the number of clusters

Let's take the example from the first chapter regarding house ownership:

Age	Annual income in USD	House ownership status
23	50,000	Non-owner
37	34,000	Non-owner
48	40,000	Owner
52	30,000	Non-owner
28	95,000	Owner
25	78,000	Non-owner
35	13,0000	Owner
32	10,5000	Owner
20	10,0000	Non-owner
40	60,000	Owner
50	80,000	Peter

We would like to predict whether Peter is a house owner using clustering.

Analysis

Just as in the first chapter, we will have to scale the data, since the *income* axis is significantly greater and thus would diminish the impact of the *age* axis, which actually has a good predictive power in this kind of problem. This is because it is expected that older people have had more time to settle down, save money, and buy a house, as compared to younger people.

We apply the same rescaling from Chapter 1, *Classification Using K Nearest Neighbors*, and obtain the following table:

Age	Scaled age	Annual income in USD	Scaled annual income	House ownership status
23	0.09375	50000	0.2	non-owner
37	0.53125	34000	0.04	non-owner
48	0.875	40000	0.1	owner
52	1	30000	0	non-owner
28	0.25	95000	0.65	owner
25	0.15625	78000	0.48	non-owner
35	0.46875	130000	1	owner
32	0.375	105000	0.75	owner
20	0	100000	0.7	non-owner
40	0.625	60000	0.3	owner
50	0.9375	80000	0.5	?

Given the table, we produce the input file for the algorithm and execute it, clustering the features into two clusters.

Input:

```
# source_code/5/house_ownership2.csv
0.09375,0.2
0.53125,0.04
0.875,0.1
1,0
0.25,0.65
0.15625,0.48
0.46875,1
0.375,0.75
0,0.7
0.625,0.3
0.9375,0.5
```

Output for two clusters:

```
$ python k-means_clustering.py house_ownership2.csv 2 last
```

```
The total number of steps: 3
```

```
The history of the algorithm:
```

```
Step number 0: point_groups = [((0.09375, 0.2), 0), ((0.53125, 0.04), 0),  
((0.875, 0.1), 1), ((1.0, 0.0), 1), ((0.25, 0.65), 0), ((0.15625, 0.48),  
0), ((0.46875, 1.0), 0), ((0.375, 0.75), 0), ((0.0, 0.7), 0), ((0.625,  
0.3), 1), ((0.9375, 0.5), 1)]
```

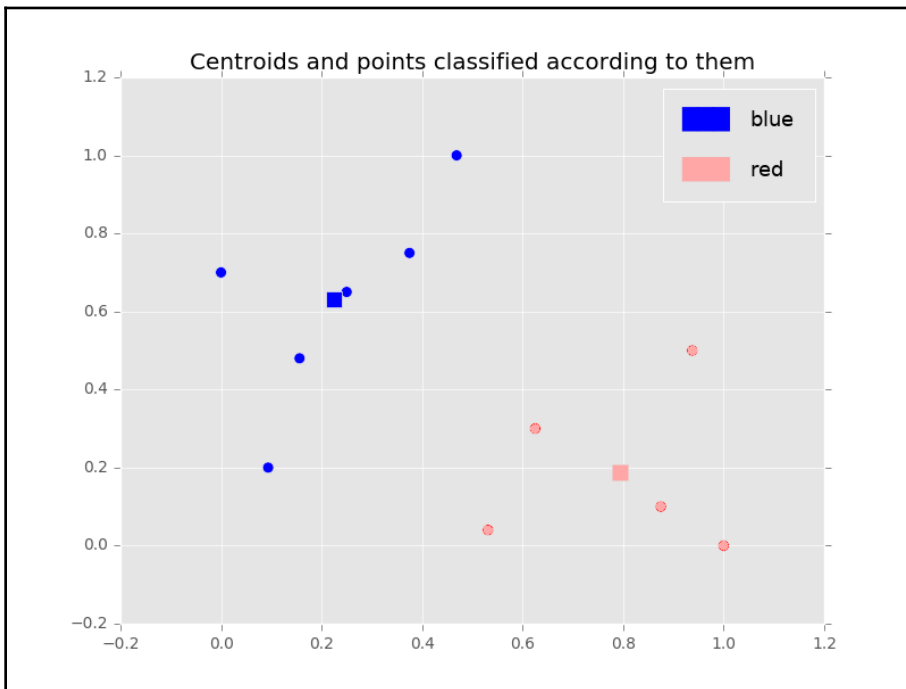
```
centroids = [(0.09375, 0.2), (1.0, 0.0)]
```

```
Step number 1: point_groups = [((0.09375, 0.2), 0), ((0.53125, 0.04), 1),  
((0.875, 0.1), 1), ((1.0, 0.0), 1), ((0.25, 0.65), 0), ((0.15625, 0.48),  
0), ((0.46875, 1.0), 0), ((0.375, 0.75), 0), ((0.0, 0.7), 0), ((0.625,  
0.3), 1), ((0.9375, 0.5), 1)]
```

```
centroids = [(0.26785714285714285, 0.5457142857142857), (0.859375, 0.225)]
```

```
Step number 2: point_groups = [((0.09375, 0.2), 0), ((0.53125, 0.04), 1),  
((0.875, 0.1), 1), ((1.0, 0.0), 1), ((0.25, 0.65), 0), ((0.15625, 0.48),  
0), ((0.46875, 1.0), 0), ((0.375, 0.75), 0), ((0.0, 0.7), 0), ((0.625,  
0.3), 1), ((0.9375, 0.5), 1)]
```

```
centroids = [(0.22395833333333334, 0.63), (0.79375, 0.188)]
```

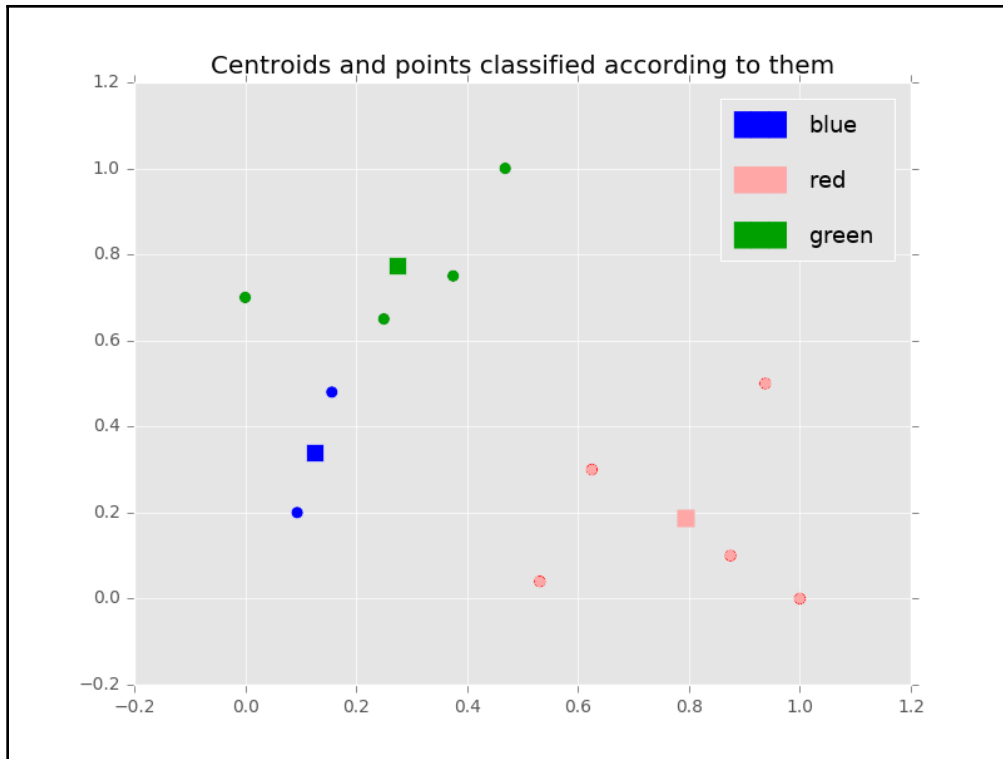


The blue cluster contains scaled features – $(0.09375, 0.2)$, $(0.25, 0.65)$, $(0.15625, 0.48)$, $(0.46875, 1)$, $(0.375, 0.75)$, and $(0, 0.7)$, and unscaled ones – $(23, 50000)$, $(28, 95000)$, $(25, 78000)$, $(35, 130000)$, $(32, 105000)$, and $(20, 100000)$. The red cluster contains scaled features – $(0.53125, 0.04)$, $(0.875, 0.1)$, $(1, 0)$, $(0.625, 0.3)$, and $(0.9375, 0.5)$, and unscaled ones $(37, 34000)$, $(48, 40000)$, $(52, 30000)$, $(40, 60000)$, and $(50, 80000)$.

So, Peter belongs to the red cluster. What is the proportion of house owners in the red cluster, not counting Peter? $2/4$, or $1/2$, of the people in the red cluster are house owners. Thus, the red cluster, to which Peter belongs, does not seem to have a high predictive power in determining whether Peter would be a house owner or not. We may try to cluster the data into more clusters, in the hope that we would gain a purer cluster that could be more reliable for the prediction of house ownership for Peter. Therefore, let's try to cluster the data into three clusters.

Output for three clusters:

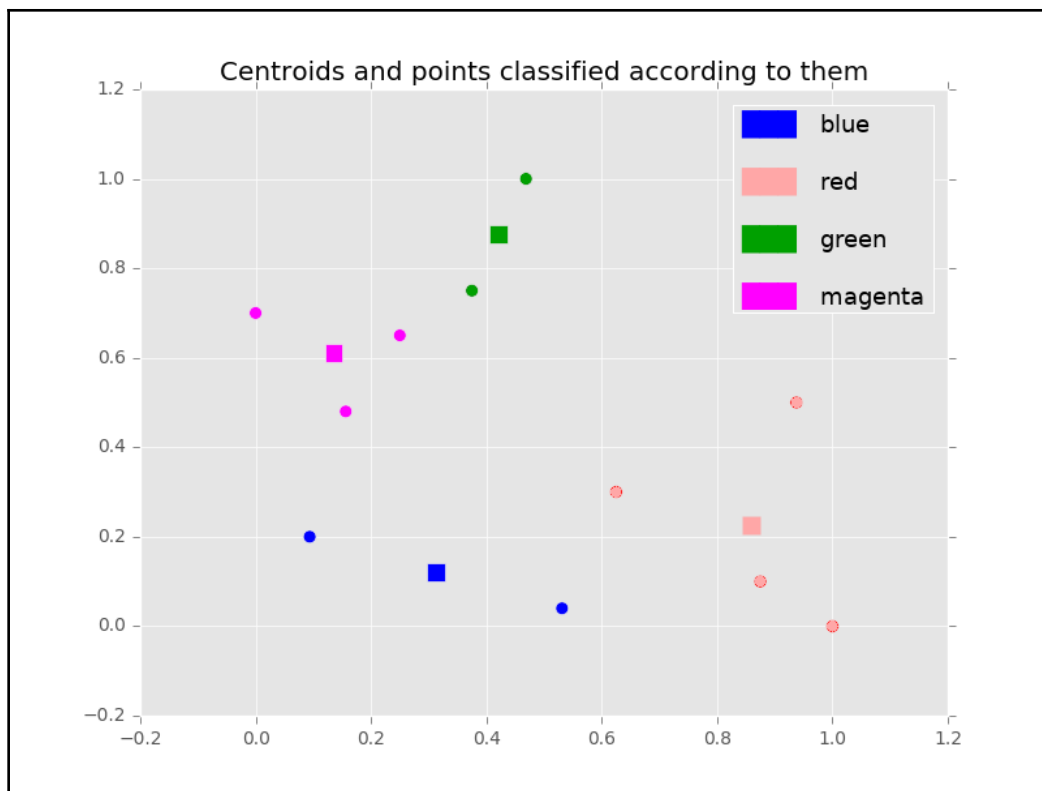
```
$ python k-means_clustering.py house_ownership2.csv 3 last
The total number of steps: 3
The history of the algorithm:
Step number 0: point_groups = [((0.09375, 0.2), 0), ((0.53125, 0.04), 0),
((0.875, 0.1), 1), ((1.0, 0.0), 1), ((0.25, 0.65), 2), ((0.15625, 0.48),
0), ((0.46875, 1.0), 2), ((0.375, 0.75), 2), ((0.0, 0.7), 0), ((0.625,
0.3), 1), ((0.9375, 0.5), 1)]
centroids = [(0.09375, 0.2), (1.0, 0.0), (0.46875, 1.0)]
Step number 1: point_groups = [((0.09375, 0.2), 0), ((0.53125, 0.04), 1),
((0.875, 0.1), 1), ((1.0, 0.0), 1), ((0.25, 0.65), 2), ((0.15625, 0.48),
0), ((0.46875, 1.0), 2), ((0.375, 0.75), 2), ((0.0, 0.7), 2), ((0.625,
0.3), 1), ((0.9375, 0.5), 1)]
centroids = [(0.1953125, 0.355), (0.859375, 0.225), (0.3645833333333333,
0.7999999999999999)]
Step number 2: point_groups = [((0.09375, 0.2), 0), ((0.53125, 0.04), 1),
((0.875, 0.1), 1), ((1.0, 0.0), 1), ((0.25, 0.65), 2), ((0.15625, 0.48),
0), ((0.46875, 1.0), 2), ((0.375, 0.75), 2), ((0.0, 0.7), 2), ((0.625,
0.3), 1), ((0.9375, 0.5), 1)]
centroids = [(0.125, 0.33999999999999997), (0.79375, 0.188), (0.2734375,
0.7749999999999999)]
```



The red cluster has stayed the same. So let's cluster the data into four clusters.

Output for four clusters:

```
$ python k-means_clustering.py house_ownership2.csv 4 last
The total number of steps: 2
The history of the algorithm:
Step number 0: point_groups = [(0.09375, 0.2), 0), ((0.53125, 0.04), 0),
((0.875, 0.1), 1), ((1.0, 0.0), 1), ((0.25, 0.65), 3), ((0.15625, 0.48),
3), ((0.46875, 1.0), 2), ((0.375, 0.75), 2), ((0.0, 0.7), 3), ((0.625,
0.3), 1), ((0.9375, 0.5), 1)]
centroids = [(0.09375, 0.2), (1.0, 0.0), (0.46875, 1.0), (0.0, 0.7)]
Step number 1: point_groups = [(0.09375, 0.2), 0), ((0.53125, 0.04), 0),
((0.875, 0.1), 1), ((1.0, 0.0), 1), ((0.25, 0.65), 3), ((0.15625, 0.48),
3), ((0.46875, 1.0), 2), ((0.375, 0.75), 2), ((0.0, 0.7), 3), ((0.625,
0.3), 1), ((0.9375, 0.5), 1)]
centroids = [(0.3125, 0.12000000000000001), (0.859375, 0.225), (0.421875,
0.875), (0.13541666666666666, 0.61)]
```

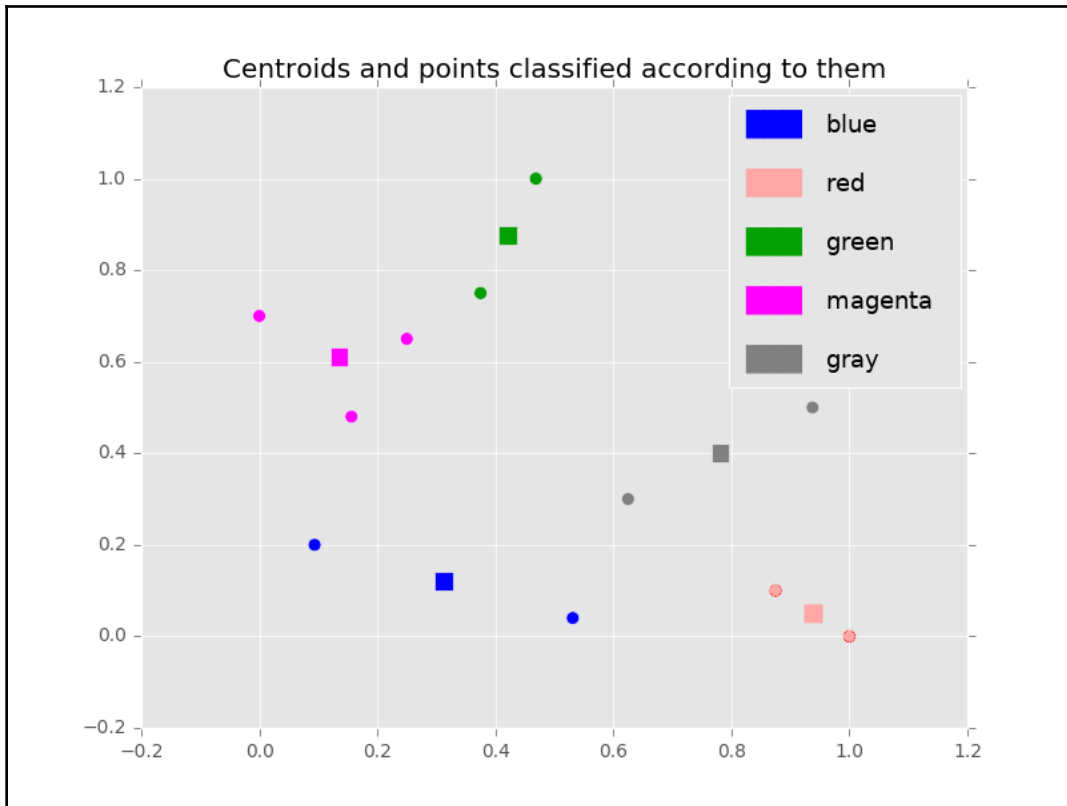


Now, the red cluster, where Peter belongs, has changed. What is the proportion of house owners in the red cluster now? If we do not count Peter, $2/3$ of people in the red cluster own a house. When we clustered into two or three clusters, the proportion was only $1/2$, which did not tell us whether Peter is a house owner or not. Now, there is a majority of house owners in the red cluster, not counting Peter, so we have a stronger belief that Peter is also a house owner. However, $2/3$ is still a relatively low confidence level for classifying Peter as a house owner. Let's cluster the data into five clusters to see what happens.

Output for five clusters:

```
$ python k-means_clustering.py house_ownership2.csv 5 last
The total number of steps: 2
The history of the algorithm:
Step number 0: point_groups = [((0.09375, 0.2), 0), ((0.53125, 0.04), 0),
((0.875, 0.1), 1), ((1.0, 0.0), 1), ((0.25, 0.65), 3), ((0.15625, 0.48),
3), ((0.46875, 1.0), 2), ((0.375, 0.75), 2), ((0.0, 0.7), 3), ((0.625,
0.3), 4), ((0.9375, 0.5), 4)]
centroids = [(0.09375, 0.2), (1.0, 0.0), (0.46875, 1.0), (0.0, 0.7),
```

```
(0.9375, 0.5)]
Step number 1: point_groups = [((0.09375, 0.2), 0), ((0.53125, 0.04), 0),
((0.875, 0.1), 1), ((1.0, 0.0), 1), ((0.25, 0.65), 3), ((0.15625, 0.48),
3), ((0.46875, 1.0), 2), ((0.375, 0.75), 2), ((0.0, 0.7), 3), ((0.625,
0.3), 4), ((0.9375, 0.5), 4)]
centroids = [(0.3125, 0.12000000000000001), (0.9375, 0.05), (0.421875,
0.875), (0.13541666666666666, 0.61), (0.78125, 0.4)]
```



Now, the red cluster contains only Peter and a non-owner. This clustering suggests that Peter is more likely a non-owner as well. However, according to the previous cluster, Peter is more likely be an owner of a house. Therefore, it may not be so clear whether Peter owns a house or not. Collecting more data would improve our analysis and should be carried out before making a definite classification of this problem.

From our analysis, we noticed that a different number of clusters can result in a different result for a classification, since the nature of members in an individual cluster can change. After collecting more data, we should perform cross-validation to determine the number of clusters that classifies the data with the highest accuracy.

Document clustering – understanding the number of k clusters in a semantic context

We are given the following information about the frequency counts for the words *money* and *god(s)* in the following 17 books from the Project Gutenberg library:

Book number	Book name	Money as a %	God(s) as a %
1	<i>The Vedanta-Sutras, with the commentary by Ramanuja</i> , by Trans. George Thibaut	0	0.07
2	<i>The Mahabharata of Krishna-Dwaipayana Vyasa -Adi Parva</i> , by Kisari Mohan Ganguli	0	0.17
3	<i>The Mahabharata of Krishna-Dwaipayana Vyasa, Pt. 2</i> , by Krishna-Dwaipayana Vyasa	0.01	0.10
4	<i>The Mahabharata of Krishna-Dwaipayana Vyasa Bk. 3 Pt. 1</i> , by Krishna-Dwaipayana Vyasa	0	0.32
5	<i>The Mahabharata of Krishna-Dwaipayana Vyasa Bk. 4</i> , by Kisari Mohan Ganguli	0	0.06
6	<i>The Mahabharata of Krishna-Dwaipayana Vyasa Bk. 3 Pt. 2</i> , translated by Kisari Mohan Ganguli	0	0.27
7	<i>The Vedanta-Sutras, with commentary by Sankaracarya</i>	0	0.06
8	<i>The King James Bible</i>	0.02	0.59
9	<i>Paradise Regained</i> , by John Milton	0.02	0.45
10	<i>Imitation of Christ</i> , by Thomas A Kempis	0.01	0.69
11	<i>The Koran</i> , as translated by Rodwell	0.01	1.72
12	<i>The Adventures of Tom Sawyer</i> , complete, by Mark Twain (Samuel Clemens)	0.05	0.01
13	<i>The Adventures of Huckleberry Finn</i> , complete, by Mark Twain (Samuel Clemens)	0.08	0
14	<i>Great Expectations</i> , by Charles Dickens	0.04	0.01

15	<i>The Picture of Dorian Gray</i> , by Oscar Wilde	0.03	0.03
16	<i>The Adventures of Sherlock Holmes</i> , by Arthur Conan Doyle	0.04	0.03
17	<i>Metamorphosis</i> , by Franz Kafka Translated by David Wyllie	0.06	0.03

We would like to cluster this dataset, based on the chosen frequency counts of the words, into groups according to their semantic context.

Analysis

First, we will perform rescaling, since the highest frequency count of the word *money* is 0.08 percent, whereas the highest frequency count of the word "god(s)" is 1.72%. So, we will divide the frequency count of money by 0.08, and the frequency count of god(s) by 1.72, as follows:

Book number	Money scaled	God(s) scaled
1	0	0.0406976744
2	0	0.0988372093
3	0.125	0.0581395349
4	0	0.1860465116
5	0	0.0348837209
6	0	0.1569767442
7	0	0.0348837209
8	0.25	0.3430232558
9	0.25	0.261627907
10	0.125	0.4011627907
11	0.125	1
12	0.625	0.0058139535
13	1	0
14	0.5	0.0058139535
15	0.375	0.0174418605
16	0.5	0.0174418605
17	0.75	0.0174418605

Now that we have rescaled the data, let's apply the k -means clustering algorithm by trying to divide the data into a different amount of clusters.

Input:

```
source_code/5/document_clustering/word_frequencies_money_god_scaled.csv
0,0.0406976744
0,0.0988372093
0.125,0.0581395349
0,0.1860465116
0,0.0348837209
0,0.1569767442
0,0.0348837209
0.25,0.3430232558
0.25,0.261627907
0.125,0.4011627907
0.125,1
0.625,0.0058139535
1,0
0.5,0.0058139535
0.375,0.0174418605
0.5,0.0174418605
0.75,0.0174418605
```

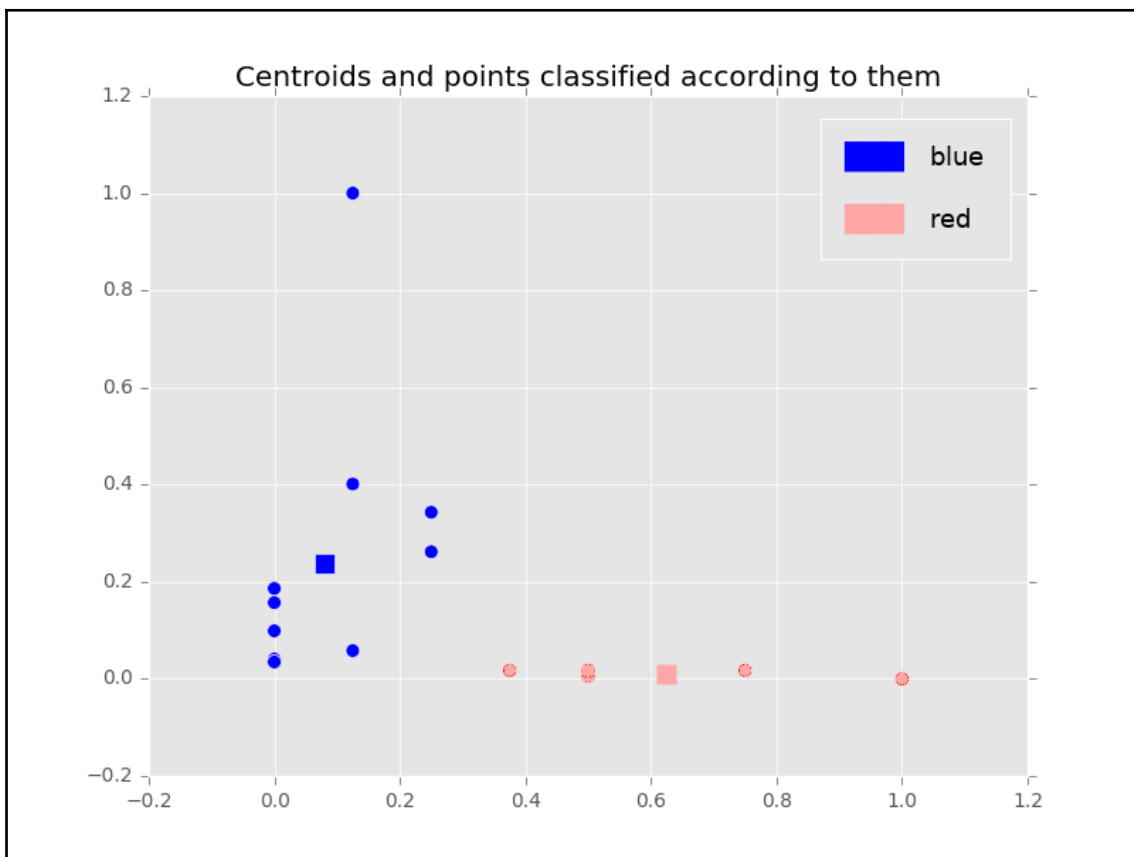
Output for two clusters:

```
$ python k-means_clustering.py
document_clustering/word_frequencies_money_god_scaled.csv 2 last
The total number of steps: 3
The history of the algorithm:
Step number 0: point_groups = [((0.0, 0.0406976744), 0), ((0.0,
0.0988372093), 0), ((0.125, 0.0581395349), 0), ((0.0, 0.1860465116), 0),
((0.0, 0.0348837209), 0), ((0.0, 0.1569767442), 0), ((0.0, 0.0348837209),
0), ((0.25, 0.3430232558), 0), ((0.25, 0.261627907), 0), ((0.125,
0.4011627907), 0), ((0.125, 1.0), 0), ((0.625, 0.0058139535), 1), ((1.0,
0.0), 1), ((0.5, 0.0058139535), 1), ((0.375, 0.0174418605), 0), ((0.5,
0.0174418605), 1), ((0.75, 0.0174418605), 1)]
centroids = [(0.0, 0.0406976744), (1.0, 0.0)]
Step number 1: point_groups = [((0.0, 0.0406976744), 0), ((0.0,
0.0988372093), 0), ((0.125, 0.0581395349), 0), ((0.0, 0.1860465116), 0),
((0.0, 0.0348837209), 0), ((0.0, 0.1569767442), 0), ((0.0, 0.0348837209),
0), ((0.25, 0.3430232558), 0), ((0.25, 0.261627907), 0), ((0.125,
0.4011627907), 0), ((0.125, 1.0), 0), ((0.625, 0.0058139535), 1), ((1.0,
0.0), 1), ((0.5, 0.0058139535), 1), ((0.375, 0.0174418605), 1), ((0.5,
0.0174418605), 1), ((0.75, 0.0174418605), 1)]
centroids = [(0.10416666666666667, 0.21947674418333332), (0.675,
0.0093023256)]
Step number 2: point_groups = [((0.0, 0.0406976744), 0), ((0.0,
```

```

0.0988372093), 0), ((0.125, 0.0581395349), 0), ((0.0, 0.1860465116), 0),
((0.0, 0.0348837209), 0), ((0.0, 0.1569767442), 0), ((0.0, 0.0348837209),
0), ((0.25, 0.3430232558), 0), ((0.25, 0.261627907), 0), ((0.125,
0.4011627907), 0), ((0.125, 1.0), 0), ((0.625, 0.0058139535), 1), ((1.0,
0.0), 1), ((0.5, 0.0058139535), 1), ((0.375, 0.0174418605), 1), ((0.5,
0.0174418605), 1), ((0.75, 0.0174418605), 1)]
centroids = [(0.07954545454545454, 0.2378435517909091), (0.625,
0.01065891475)]

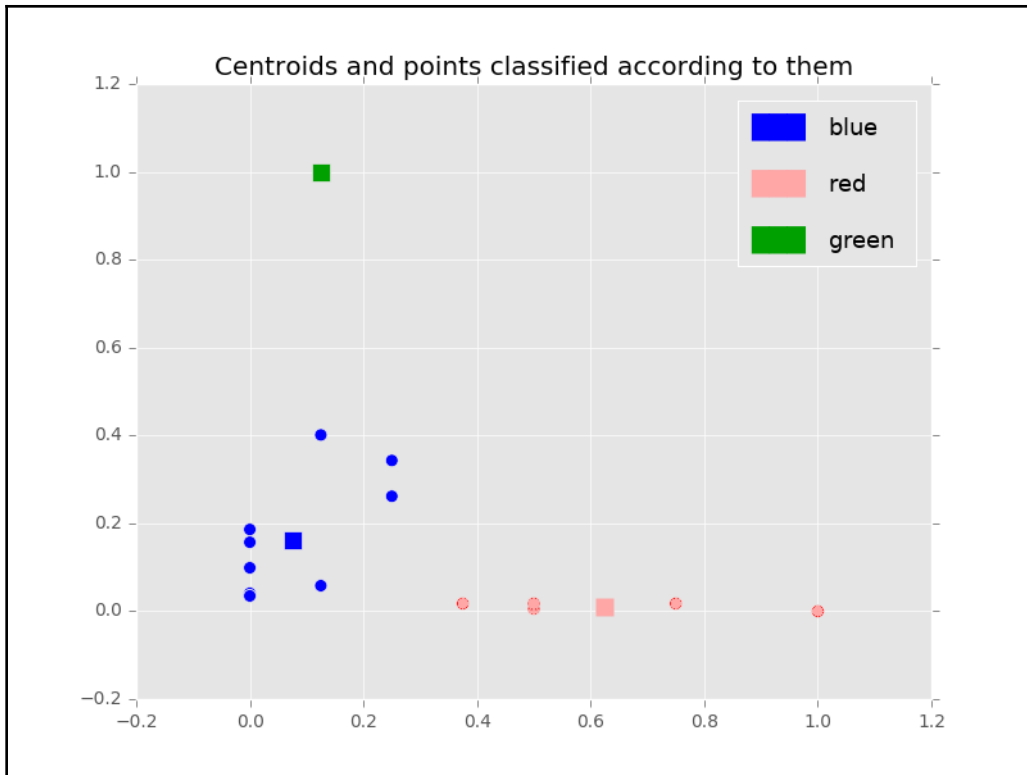
```



We can observe that clustering into two clusters divides books into religious ones, which are the ones in the blue cluster, and non-religious ones, which are the ones in the red cluster. Let's try to cluster the books into three clusters to observe how the algorithm will divide up the data.

Output for three clusters:

```
$ python k-means_clustering.py
document_clustering/word_frequencies_money_god_scaled.csv 3 last
The total number of steps: 3
The history of the algorithm:
Step number 0: point_groups = [((0.0, 0.0406976744), 0), ((0.0,
0.0988372093), 0), ((0.125, 0.0581395349), 0), ((0.0, 0.1860465116), 0),
((0.0, 0.0348837209), 0), ((0.0, 0.1569767442), 0), ((0.0, 0.0348837209),
0), ((0.25, 0.3430232558), 0), ((0.25, 0.261627907), 0), ((0.125,
0.4011627907), 0), ((0.125, 1.0), 2), ((0.625, 0.0058139535), 1), ((1.0,
0.0), 1), ((0.5, 0.0058139535), 1), ((0.375, 0.0174418605), 0), ((0.5,
0.0174418605), 1), ((0.75, 0.0174418605), 1)]
centroids = [(0.0, 0.0406976744), (1.0, 0.0), (0.125, 1.0)]
Step number 1: point_groups = [((0.0, 0.0406976744), 0), ((0.0,
0.0988372093), 0), ((0.125, 0.0581395349), 0), ((0.0, 0.1860465116), 0),
((0.0, 0.0348837209), 0), ((0.0, 0.1569767442), 0), ((0.0, 0.0348837209),
0), ((0.25, 0.3430232558), 0), ((0.25, 0.261627907), 0), ((0.125,
0.4011627907), 0), ((0.125, 1.0), 2), ((0.625, 0.0058139535), 1), ((1.0,
0.0), 1), ((0.5, 0.0058139535), 1), ((0.375, 0.0174418605), 1), ((0.5,
0.0174418605), 1), ((0.75, 0.0174418605), 1)]
centroids = [(0.10227272727272728, 0.14852008456363636), (0.675,
0.0093023256), (0.125, 1.0)]
Step number 2: point_groups = [((0.0, 0.0406976744), 0), ((0.0,
0.0988372093), 0), ((0.125, 0.0581395349), 0), ((0.0, 0.1860465116), 0),
((0.0, 0.0348837209), 0), ((0.0, 0.1569767442), 0), ((0.0, 0.0348837209),
0), ((0.25, 0.3430232558), 0), ((0.25, 0.261627907), 0), ((0.125,
0.4011627907), 0), ((0.125, 1.0), 2), ((0.625, 0.0058139535), 1), ((1.0,
0.0), 1), ((0.5, 0.0058139535), 1), ((0.375, 0.0174418605), 1), ((0.5,
0.0174418605), 1), ((0.75, 0.0174418605), 1)]
centroids = [(0.075, 0.16162790697), (0.625, 0.01065891475), (0.125, 1.0)]
```

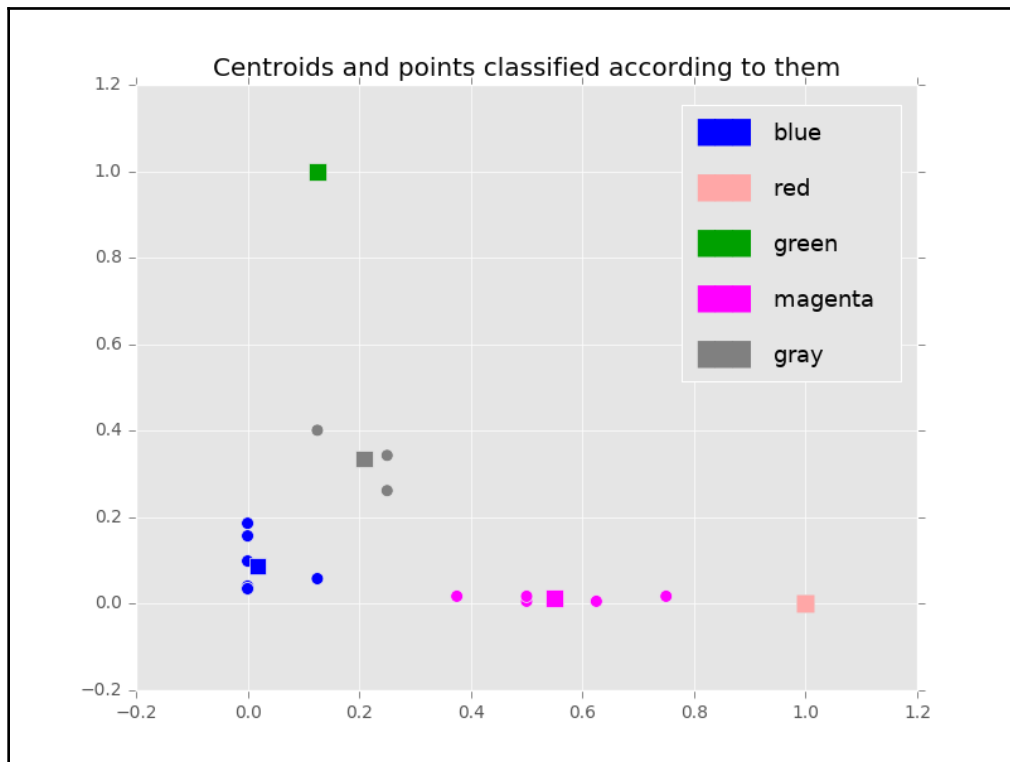


This time, the algorithm separated The Koran, from the religious books, into a green cluster. This is because, in fact, the word *god* is the fifth most commonly occurring word in The Koran. The clustering here happens to divide the books according to the writing style they were written in. Clustering into four clusters separates one book, which has a relatively high frequency of the word *money*, from the red cluster of non-religious books into a separate cluster. Now let's look at clustering into five clusters.

Output for five clusters:

```
$ python k-means_clustering.py word_frequencies_money_god_scaled.csv 5 last
The total number of steps: 2
The history of the algorithm:
Step number 0: point_groups = [((0.0, 0.0406976744), 0), ((0.0,
0.0988372093), 0), ((0.125, 0.0581395349), 0), ((0.0, 0.1860465116), 0),
((0.0, 0.0348837209), 0), ((0.0, 0.1569767442), 0), ((0.0, 0.0348837209),
0), ((0.25, 0.3430232558), 4), ((0.25, 0.261627907), 4), ((0.125,
0.4011627907), 4), ((0.125, 1.0), 2), ((0.625, 0.0058139535), 3), ((1.0,
0.0), 1), ((0.5, 0.0058139535), 3), ((0.375, 0.0174418605), 3), ((0.5,
0.0174418605), 3), ((0.75, 0.0174418605), 3)]
```

```
centroids = [(0.0, 0.0406976744), (1.0, 0.0), (0.125, 1.0), (0.5,
0.0174418605), (0.25, 0.3430232558)]
Step number 1: point_groups = [((0.0, 0.0406976744), 0), ((0.0,
0.0988372093), 0), ((0.125, 0.0581395349), 0), ((0.0, 0.1860465116), 0),
((0.0, 0.0348837209), 0), ((0.0, 0.1569767442), 0), ((0.0, 0.0348837209),
0), ((0.25, 0.3430232558), 4), ((0.25, 0.261627907), 4), ((0.125,
0.4011627907), 4), ((0.125, 1.0), 2), ((0.625, 0.0058139535), 3), ((1.0,
0.0), 1), ((0.5, 0.0058139535), 3), ((0.375, 0.0174418605), 3), ((0.5,
0.0174418605), 3), ((0.75, 0.0174418605), 3)]
centroids = [(0.017857142857142856, 0.08720930231428571), (1.0, 0.0),
(0.125, 1.0), (0.55, 0.0127906977), (0.20833333333333334,
0.3352713178333333)]
```



This clustering further divides the blue cluster of the remaining religious books into a blue cluster of Hindu books and a gray cluster of Christian books.

We can use clustering in this way to group items with similar properties and then find similar items quickly based on the given example. The granularity of the clustering under the parameter k determines how similar we can expect the items in a group to be. The higher the parameter, the more similar the items in the cluster are going to be, albeit in a smaller number.

Summary

In this chapter, we learned how the clustering of data is very efficient and can be used to facilitate the faster classification of new features by classifying a feature as belonging to the class that is represented in the cluster of that feature. An appropriate number of clusters can be determined through cross-validation, by choosing the one that results in the most accurate classification.

Clustering orders data according to its similarity. The more clusters there are, the greater the similarity between the features in a cluster, but the fewer features in a cluster there are.

We also learned that the k -means algorithm is a clustering algorithm that tries to cluster features in such a way that the mutual distance of the features in a cluster is minimized. To do this, the algorithm computes the centroid of each cluster and a feature belongs to the cluster whose centroid is closest to it. The algorithm finishes the computation of the clusters as soon as they or their centroids no longer change.

In the next chapter, we will analyze the relationship between dependent variables using mathematical regression. Unlike with the classification and clustering algorithms, regression analysis will be used to estimate the most probable value of a variable, such as weight, distance, or temperature.

Problems

Problem 1: Compute the centroid of the following clusters:

- a) 2, 3, 4
- b) USD 100, USD 400, USD 1,000
- c) (10,20), (40, 60), (0, 40)
- d) (USD 200, 40 km), (USD 300, 60 km), (USD 500, 100 km), (USD 250, 200 km)
- e) (1,2,4), (0,0,3), (10,20,5), (4,8,2), (5,0,1)

Problem 2: Cluster the following datasets into two, three, and four clusters using the k -means clustering algorithm:

a) 0, 2, 5, 4, 8, 10, 12, 11

b) (2,2), (2,5), (10,4), (3,5), (7,3), (5,9), (2,8), (4,10), (7,4), (4,4), (5,8), (9,3)

Problem 3: We are given the ages of the couples and the number of children they have:

Couple number	Wife's age	Husband's age	Number of children
1	48	49	5
2	40	43	2
3	24	28	1
4	49	42	3
5	32	34	0
6	24	27	0
7	29	32	2
8	35	35	2
9	33	36	1
10	42	47	3
11	22	27	2
12	41	45	4
13	39	43	4
14	36	38	2
15	30	32	1
16	36	38	0
17	36	39	3
18	37	38	?

We would like to guess, using clustering, how many children a couple has where the age of the husband is 37 and the age of the wife is 38.

Analysis

Problem 1: a) $(1/3) * (2 + 3 + 4) = 3$

b) $(1/3) * (100\$ + 400\$ + 1000\$) = 500\$$

c) $((10 + 40 + 0)/3, (20 + 60 + 40)/3) = (50/3, 120/3) = (50/3, 40)$

$$\begin{aligned} & ((200\$ + 300\$ + 500\$ + 250\$)/4, (40km + 60km + 100km + 200km)/4) \\ & = (1250\$/4, 400km/4) \end{aligned}$$

d) $= (312.5\$, 100km)$

e) $((1 + 0 + 10 + 4 + 5)/5, (2 + 0 + 20 + 8 + 0)/5, (4 + 3 + 5 + 2 + 1)/5) = (4, 6, 3)$

Problem 2: a) We add a second coordinate and set it to 0 for all the features. This way, the distance between the features does not change and we can use the clustering algorithm we implemented earlier in this chapter.

Input:

```
# source_code/5/problem5_2.csv
0,0
2,0
5,0
4,0
8,0
10,0
12,0
11,0
```

For two clusters:

```
$ python k-means_clustering.py problem5_2.csv 2 last
The total number of steps: 2
The history of the algorithm:
Step number 0: point_groups = [((0.0, 0.0), 0), ((2.0, 0.0), 0), ((5.0,
0.0), 0), ((4.0, 0.0), 0), ((8.0, 0.0), 1), ((10.0, 0.0), 1), ((12.0, 0.0),
1), ((11.0, 0.0), 1)]
centroids = [(0.0, 0.0), (12.0, 0.0)]
Step number 1: point_groups = [((0.0, 0.0), 0), ((2.0, 0.0), 0), ((5.0,
0.0), 0), ((4.0, 0.0), 0), ((8.0, 0.0), 1), ((10.0, 0.0), 1), ((12.0, 0.0),
1), ((11.0, 0.0), 1)]
centroids = [(2.75, 0.0), (10.25, 0.0)]
```

For three clusters:

```
$ python k-means_clustering.py problem5_2.csv 3 last
```

```
The total number of steps: 2
```

```
The history of the algorithm:
```

```
Step number 0: point_groups = [((0.0, 0.0), 0), ((2.0, 0.0), 0), ((5.0, 0.0), 2), ((4.0, 0.0), 2), ((8.0, 0.0), 2), ((10.0, 0.0), 1), ((12.0, 0.0), 1), ((11.0, 0.0), 1)]
```

```
centroids = [(0.0, 0.0), (12.0, 0.0), (5.0, 0.0)]
```

```
Step number 1: point_groups = [((0.0, 0.0), 0), ((2.0, 0.0), 0), ((5.0, 0.0), 2), ((4.0, 0.0), 2), ((8.0, 0.0), 2), ((10.0, 0.0), 1), ((12.0, 0.0), 1), ((11.0, 0.0), 1)]
```

```
centroids = [(1.0, 0.0), (11.0, 0.0), (5.666666666666667, 0.0)]
```

For four clusters:

```
$ python k-means_clustering.py problem5_2.csv 4 last
```

```
The total number of steps: 2
```

```
The history of the algorithm:
```

```
Step number 0: point_groups = [((0.0, 0.0), 0), ((2.0, 0.0), 0), ((5.0, 0.0), 2), ((4.0, 0.0), 2), ((8.0, 0.0), 3), ((10.0, 0.0), 1), ((12.0, 0.0), 1), ((11.0, 0.0), 1)]
```

```
centroids = [(0.0, 0.0), (12.0, 0.0), (5.0, 0.0), (8.0, 0.0)]
```

```
Step number 1: point_groups = [((0.0, 0.0), 0), ((2.0, 0.0), 0), ((5.0, 0.0), 2), ((4.0, 0.0), 2), ((8.0, 0.0), 3), ((10.0, 0.0), 1), ((12.0, 0.0), 1), ((11.0, 0.0), 1)]
```

```
centroids = [(1.0, 0.0), (11.0, 0.0), (4.5, 0.0), (8.0, 0.0)]
```

b) We use the implemented algorithm again.

Input:

```
# source_code/5/problem5_2b.csv
```

```
2,2
```

```
2,5
```

```
10,4
```

```
3,5
```

```
7,3
```

```
5,9
```

```
2,8
```

```
4,10
```

```
7,4
```

```
4,4
```

```
5,8
```

```
9,3
```

Output for two clusters:

```
$ python k-means_clustering.py problem5_2b.csv 2 last
The total number of steps: 3
The history of the algorithm:
Step number 0: point_groups = [((2.0, 2.0), 0), ((2.0, 5.0), 0), ((10.0,
4.0), 1), ((3.0, 5.0), 0), ((7.0, 3.0), 1), ((5.0, 9.0), 1), ((2.0, 8.0),
0), ((4.0, 10.0), 0), ((7.0, 4.0), 1), ((4.0, 4.0), 0), ((5.0, 8.0), 1),
((9.0, 3.0), 1)]
centroids = [(2.0, 2.0), (10.0, 4.0)]
Step number 1: point_groups = [((2.0, 2.0), 0), ((2.0, 5.0), 0), ((10.0,
4.0), 1), ((3.0, 5.0), 0), ((7.0, 3.0), 1), ((5.0, 9.0), 0), ((2.0, 8.0),
0), ((4.0, 10.0), 0), ((7.0, 4.0), 1), ((4.0, 4.0), 0), ((5.0, 8.0), 0),
((9.0, 3.0), 1)]
centroids = [(2.8333333333333335, 5.666666666666667), (7.166666666666667,
5.166666666666667)]
Step number 2: point_groups = [((2.0, 2.0), 0), ((2.0, 5.0), 0), ((10.0,
4.0), 1), ((3.0, 5.0), 0), ((7.0, 3.0), 1), ((5.0, 9.0), 0), ((2.0, 8.0),
0), ((4.0, 10.0), 0), ((7.0, 4.0), 1), ((4.0, 4.0), 0), ((5.0, 8.0), 0),
((9.0, 3.0), 1)]
centroids = [(3.375, 6.375), (8.25, 3.5)]
```

Output for three clusters:

```
$ python k-means_clustering.py problem5_2b.csv 3 last
The total number of steps: 2
The history of the algorithm:
Step number 0: point_groups = [((2.0, 2.0), 0), ((2.0, 5.0), 0), ((10.0,
4.0), 1), ((3.0, 5.0), 0), ((7.0, 3.0), 1), ((5.0, 9.0), 2), ((2.0, 8.0),
2), ((4.0, 10.0), 2), ((7.0, 4.0), 1), ((4.0, 4.0), 0), ((5.0, 8.0), 2),
((9.0, 3.0), 1)]
centroids = [(2.0, 2.0), (10.0, 4.0), (4.0, 10.0)]
Step number 1: point_groups = [((2.0, 2.0), 0), ((2.0, 5.0), 0), ((10.0,
4.0), 1), ((3.0, 5.0), 0), ((7.0, 3.0), 1), ((5.0, 9.0), 2), ((2.0, 8.0),
2), ((4.0, 10.0), 2), ((7.0, 4.0), 1), ((4.0, 4.0), 0), ((5.0, 8.0), 2),
((9.0, 3.0), 1)]
centroids = [(2.75, 4.0), (8.25, 3.5), (4.0, 8.75)]
```

Output for four clusters:

```
$ python k-means_clustering.py problem5_2b.csv 4 last
The total number of steps: 2
The history of the algorithm:
Step number 0: point_groups = [((2.0, 2.0), 0), ((2.0, 5.0), 3), ((10.0,
4.0), 1), ((3.0, 5.0), 3), ((7.0, 3.0), 1), ((5.0, 9.0), 2), ((2.0, 8.0),
2), ((4.0, 10.0), 2), ((7.0, 4.0), 1), ((4.0, 4.0), 3), ((5.0, 8.0), 2),
((9.0, 3.0), 1)]
centroids = [(2.0, 2.0), (10.0, 4.0), (4.0, 10.0), (3.0, 5.0)]
```

```

Step number 1: point_groups = [((2.0, 2.0), 0), ((2.0, 5.0), 3), ((10.0,
4.0), 1), ((3.0, 5.0), 3), ((7.0, 3.0), 1), ((5.0, 9.0), 2), ((2.0, 8.0),
2), ((4.0, 10.0), 2), ((7.0, 4.0), 1), ((4.0, 4.0), 3), ((5.0, 8.0), 2),
((9.0, 3.0), 1)]
centroids = [(2.0, 2.0), (8.25, 3.5), (4.0, 8.75), (3.0,
4.666666666666667)]

```

Problem 3: We are given the ages of 17 couples and the number of children they have, and would like to find out how many children the 18th couple has. We will use the first 14 couples as training data, and the next 3 couples for cross-validation to determine the number of clusters k that we will use to find out how many children the 18th couple is expected to have.

After clustering, we will say that a couple is likely to have roughly the number of children that is the average in that cluster. Using cross-validation, we will choose the number of clusters that will minimize the difference between the actual and predicted number of children. We will capture this difference for all the items in the cluster cumulatively as the square root of the squares of the differences between the number of children of each couple. This will minimize the variance of the random variable in relation to the predicted number of children for the 18th couple.

We will perform clustering into two, three, four, and five clusters.

Input:

```

# source_code/5/couples_children.csv
48,49
40,43
24,28
49,42
32,34
24,27
29,32
35,35
33,36
42,47
22,27
41,45
39,43
36,38
30,32
36,38
36,39
37,38

```

Output for two clusters:

A couple listed for a cluster is of the form `(couple_number, (wife_age, husband_age))`:

```
Cluster 0: [(1, (48.0, 49.0)), (2, (40.0, 43.0)), (4, (49.0, 42.0)), (10,
(42.0, 47.0)), (12, (41.0, 45.0)), (13, (39.0, 43.0)), (14, (36.0, 38.0)),
(16, (36.0, 38.0)), (17, (36.0, 39.0)), (18, (37.0, 38.0))]
Cluster 1: [(3, (24.0, 28.0)), (5, (32.0, 34.0)), (6, (24.0, 27.0)), (7,
(29.0, 32.0)), (8, (35.0, 35.0)), (9, (33.0, 36.0)), (11, (22.0, 27.0)),
(15, (30.0, 32.0))]
```

We would like to determine the expected number of children for the 15th couple (30,32), that is, where the wife is 30 years old and the husband is 32 years old. (30,32) is in cluster 1. The couples in cluster 1 are as follows: (24.0, 28.0), (32.0, 34.0), (24.0, 27.0), (29.0, 32.0), (35.0, 35.0), (33.0, 36.0), (22.0, 27.0), and (30.0, 32.0). Of these, and the first 14 couples used for data purposes, the remaining couples are: (24.0, 28.0), (32.0, 34.0), (24.0, 27.0), (29.0, 32.0), (35.0, 35.0), (33.0, 36.0), and (22.0, 27.0). The average number of children for these couples is $est15 = 8/7 \sim 1.14$. This is the estimated number of children for the 15th couple, based on the data from the first 14 couples.

The estimated number of children for the 16th couple is $est16 = 23/7 \sim 3.29$. The estimated number of children for the 17th couple is also $est17 = 23/7 \sim 3.29$ since both the 16th and 17th couples belong to the same cluster.

Now we will calculate the $E2$ error (two for two clusters) between the estimated number of children (for example, denoted $est15$ for the 15th couple) and the actual number of children (for example, denoted $act15$ for the 15th couple) as follows:

$$E2 = \text{sqr}t(\text{sqr}(est15 - act15) + \text{sqr}(est16 - act16) + \text{sqr}(est17 - act17)) \\ = \text{sqr}t(\text{sqr}(8/7 - 1) + \text{sqr}(23/7 - 0) + \text{sqr}(23/7 - 3)) \sim 3.3$$

Now that we have calculated the $E2$ error, we will calculate the errors in terms of the estimation with the other clusters. We will choose the number of clusters containing the fewest errors to estimate the number of children for the 18th couple.

Output for three clusters:

```
Cluster 0: [(1, (48.0, 49.0)), (2, (40.0, 43.0)), (4, (49.0, 42.0)), (10,
(42.0, 47.0)), (12, (41.0, 45.0)), (13, (39.0, 43.0))]
Cluster 1: [(3, (24.0, 28.0)), (6, (24.0, 27.0)), (7, (29.0, 32.0)), (11,
(22.0, 27.0)), (15, (30.0, 32.0))]
Cluster 2: [(5, (32.0, 34.0)), (8, (35.0, 35.0)), (9, (33.0, 36.0)), (14,
(36.0, 38.0)), (16, (36.0, 38.0)), (17, (36.0, 39.0)), (18, (37.0, 38.0))]
```

Now, the 15th couple is in cluster 1, the 16th couple is in cluster 2, and the 17th couple is in cluster 2. So the estimated number of children for each couple is $5/4=1.25$.

The $E3$ error of the estimation is as follows:

$$E3 = \text{sqrt}((1.25 - 1)^2 + (1.25 - 0)^2 + (1.25 - 3)^2) \sim 2.17$$

Output for four clusters:

```
Cluster 0: [(1, (48.0, 49.0)), (4, (49.0, 42.0)), (10, (42.0, 47.0)), (12,
(41.0, 45.0))]
Cluster 1: [(3, (24.0, 28.0)), (6, (24.0, 27.0)), (11, (22.0, 27.0))]
Cluster 2: [(2, (40.0, 43.0)), (13, (39.0, 43.0)), (14, (36.0, 38.0)), (16,
(36.0, 38.0)), (17, (36.0, 39.0)), (18, (37.0, 38.0))]
Cluster 3: [(5, (32.0, 34.0)), (7, (29.0, 32.0)), (8, (35.0, 35.0)), (9,
(33.0, 36.0)), (15, (30.0, 32.0))]
```

The 15th couple is in cluster 3, the 16th is in cluster 2, and the 17th is in cluster 2. So, the estimated number of children for the 15th couple is $5/4=1.25$. The estimated number of children for the 16th and 17th couples is $8/3=2.67$.

The $E4$ error of the estimation is as follows:

$$E4 = \text{sqrt}((1.25 - 1)^2 + (\frac{8}{3} - 0)^2 + (\frac{8}{3} - 3)^2) \sim 2.70$$

Output for five clusters:

```
Cluster 0: [(1, (48.0, 49.0)), (4, (49.0, 42.0))]
Cluster 1: [(3, (24.0, 28.0)), (6, (24.0, 27.0)), (11, (22.0, 27.0))]
Cluster 2: [(8, (35.0, 35.0)), (9, (33.0, 36.0)), (14, (36.0, 38.0)), (16,
(36.0, 38.0)), (17, (36.0, 39.0)), (18, (37.0, 38.0))]
Cluster 3: [(5, (32.0, 34.0)), (7, (29.0, 32.0)), (15, (30.0, 32.0))]
Cluster 4: [(2, (40.0, 43.0)), (10, (42.0, 47.0)), (12, (41.0, 45.0)), (13,
(39.0, 43.0))]
```

The 15th couple is in cluster 3, the 16th is in cluster 2, and the 17th is in cluster 2. So, the estimated number of children for the 15th couple is 1. The estimated number of children for the 16th and 17th couples is $5/3=1.67$.

The $E5$ error of the estimation is as follows:

$$E5 = \text{sqrt}((1 - 1)^2 + (\frac{5}{3} - 3)^2) \sim 2.13$$

Using cross-validation to determine the outcome

We used 14 couples as training data for the estimation and three other couples for cross-validation to find the best parameter of k clusters among the values 2, 3, 4, and 5. We could try to cluster into more clusters, but since we have such a relatively small amount of data, it should be sufficient to cluster into five clusters at most. Let's summarize the errors arising from the estimation:

Number of clusters	Error rate
2	3.3
3	2.17
4	2.7
5	2.13

The error rate is lowest for 3 and 5 clusters. The fact that the error rate goes up for 4 clusters and then down again for 5 clusters could indicate that we don't have enough data to make a good estimate. A natural expectation would be that there are not local maxima of errors for values of k greater than 2. Moreover, the difference between the error rate for clustering with 3 and 5 clusters is very small, and one cluster out of 5 is smaller than one cluster out of 3. For this reason, we choose 3 clusters over 5 to estimate the number of children for the 18th couple.

When clustering into three clusters, the 18th couple is in cluster 2. Therefore, the estimated number of children for the 18th couple is 1.25.

6 Regression

Regression analysis is the process of estimating the relationship between dependent variables. For example, if a variable y is linearly dependent on variable x , then regression analysis tries to estimate the constants a and b in the equation $y = ax + b$, which expresses the linear relationship between the variables y and x .

In this chapter, you will learn about the following topics:

- The core idea of regression by performing a simple linear regression on the perfect data from first principles in a Fahrenheit and Celsius conversion example
- Performing linear regression analysis by using the **least squares method** from the `scipy.linalg` Python library on perfect and real-world data in the examples of Fahrenheit and Celsius conversion, weight prediction based on height, and flight time duration prediction based on distance
- The *gradient descent algorithm* in order to find a regression model with the best fit (using the least mean squares rule) and how to implement it in Python
- The establishment of a non-linear regression model using the least squares method in a ballistic flight analysis example and *Problem 4, Bacteria population prediction*

Fahrenheit and Celsius conversion – linear regression on perfect data

Fahrenheit and Celsius degrees are related in a linear way. Given a table with pairs of both Fahrenheit and Celsius degrees, we can estimate the constants to devise a conversion formula from degrees Fahrenheit to degrees Celsius, or vice versa:

$^{\circ}\text{F}$	$^{\circ}\text{C}$
5	-15
14	-10
23	-5
32	0
41	5
50	10

Analysis from first principles

We would like to derive a formula for converting F (degrees Fahrenheit) to C (degrees Celsius), as follows:

$$C = a * F + b$$

Here, a and b are the constants to be found. A graph of the $C = a * F + b$ function is a straight line and, thus, is uniquely determined by two points. Therefore, we actually only need two points from the table, say, pairs (F_1, C_1) and (F_2, C_2) . Then, we will have the following:

$$C_1 = a * F_1 + b$$

$$C_2 = a * F_2 + b$$

Now, we have the following:

$$C_2 - C_1 = (a * F_2 + b) - (a * F_1 + b) = a * (F_2 - F_1).$$

Therefore, we have the following:

$$a = (C_2 - C_1) / (F_2 - F_1)$$

$$b = C_1 - a * F_1 = C_1 - [(C_2 - C_1) / (F_2 - F_1)] * F_1$$

Here, let's take the first two pairs $(F1, C1)=(5, -15)$ and $(F2, C2)=(14, -10)$. This will give us the following:

$$a = (-10 - (-15)) / (14 - 5) = \frac{5}{9}$$

$$b = -15 - \left(\frac{5}{9}\right) * 5 = \frac{160}{9}$$

Therefore, the formula to calculate degrees Celsius from degrees Fahrenheit is as follows:

$$C = \left(\frac{5}{9}\right) * F - \frac{160}{9} \sim 0.5556 * F - 17.7778$$

Let's verify this against the data in the table:

$^{\circ}\text{F}$	$^{\circ}\text{C}$	$(5/9)*F-160/9$
5	-15	-15
14	-10	-10
23	-5	-5
32	0	0
41	5	5
50	10	10

We can see that the formula fits 100% of our input data. The data we worked with was perfect. In later examples, we will see that the formula that we derive doesn't fit the data perfectly. The aim will be to derive a formula that fits the data best so that the difference between the prediction and the actual data is minimized.

Least squares method for linear regression

Given the input variables $x_1, \dots, x_n$ and the output variables $y_1, \dots, y_n$, we would like to find the variables a and b so that $y_i = a * x_i + b$ for every value of i from 1 to n . If the data is perfect, this is possible. In real-world data, we want y_i to be as close to $a * x_i + b$ as possible instead; that is, we want to minimize the absolute value of $y_i - a * x_i + b$. For one specific term, this is equivalent to minimizing $(y_i - a * x_i + b)^2$.

A square function instead of an absolute value is often used, as it has more desirable mathematical properties.

The least squares method finds a and b so that the following term, called **error**, is minimized:

$$E(a, b) = \sum_{i=1}^n (y_i - (a \cdot x_i + b))^2$$

Analysis using the least squares method in Python

We use the `lstsq` least squares method from the `scipy.linalg` Python library to calculate the linear dependence relation between the degrees Celsius and degrees Fahrenheit variables.

The `lstsq` function is used as `lstsq(M, y)`, where M is a matrix of $M = x[:, \text{np.newaxis}] ** [0, 1]$, constructed from the input vector x , and y is the output vector. In other words, y is the variable that's dependent on x .

Input:

```
# source_code/6/fahrenheit_celsius.py
import numpy as np
from scipy.linalg import lstsq

#temperature data
fahrenheit = np.array([5,14,23,32,41,50])
celsius = np.array([-15,-10,-5,0,5,10])

M = fahrenheit[:, np.newaxis]**[0, 1]
model, _, _, _ = lstsq(M,celsius)
print "Intercept =", model[0]
print "fahrenheit =", model[1]
```

Output:

```
$ python fahrenheit_celsius.py
Intercept = -17.77777777777782
fahrenheit = 0.5555555555555558
```

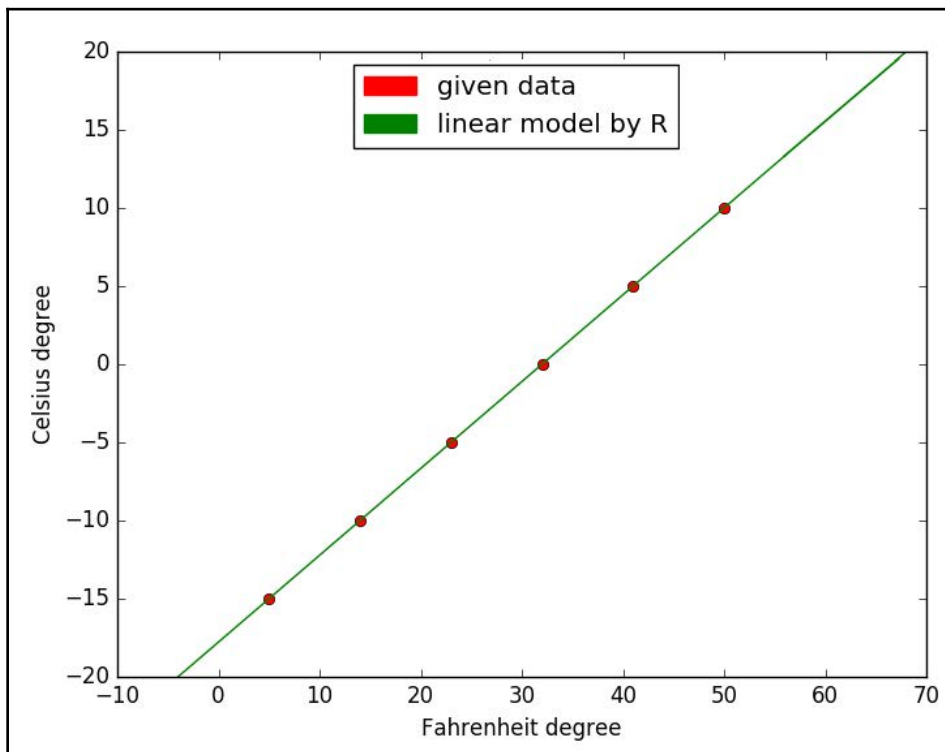
Therefore, we can see the following approximate linear dependence relation between C (degrees Celsius) and F (degrees Fahrenheit):

$$C = \text{fahrenheit} * F + \text{Intercept} = 0.5556 * F - 17.7778$$

Note that this corresponds to our previous calculation.

Visualization

Here is the linear model for predicting degrees Celsius from degrees Fahrenheit via a linear line. Its meaning is that point (F,C) is on the green line if, and only if, F converts to C and vice versa:



Weight prediction from height – linear regression on real-world data

Here, we are predicting the weight of a man from his height by using linear regression on the following data:

Height in cm	Weight in kg
180	75
174	71
184	83
168	63
178	70
172	?

We would like to estimate the weight of a man given that his height is 172 cm.

Analysis

In the previous example of Fahrenheit and Celsius conversion, the data fitted the linear model perfectly. Thus, we could perform even a simple mathematical analysis (solving basic equations) to gain the conversion formula. Most data in the real world does not fit a model perfectly. For such an analysis, it would be good to find a model that fits the given data with minimal errors. We can use the least squares method to find such a linear model.

Input:

We put the data from the preceding table into the vectors and try to fit the linear model:

```
# source_code/6/weight_prediction.py
import numpy as np
from scipy.linalg import lstsq

height = np.array([180,174,184,168,178])
weight = np.array([75,71,83,63,70])

M = height[:, np.newaxis]**[0, 1]
model, _, _, _ = lstsq(M,weight)
print "Intercept =", model[0]
print "height coefficient =", model[1]
```

Output:

```
$ python weight_prediction.py
Intercept = -127.68817204301082
height coefficient = 1.1317204301075274
```

Thus, the formula expressing the linear relationship between weight and height is as follows: $weight = 1.132 * height - 127.688$

Therefore, we estimate that a man with a height of 172 cm would have a weight of $1.132 * 172 - 127.688 = 67.016kg$.

Gradient descent algorithm and its implementation

To understand how we may be able to predict a value by using linear regression from first principles in an even better way, we need to study the gradient descent algorithm and then implement it in Python.

Gradient descent algorithm

A gradient descent algorithm is an iterative algorithm that updates the variables in the model to fit the data, making as few errors as possible. More generally, it finds the minimum of a function.

We would like to express the weight in terms of height by using a linear formula:

$$weight(height, p) = p_1 * height + p_0$$

We estimate the parameter, $p = (p_0, p_1)$, using n data samples $(height_i, weight_i)$ to minimize the following square error:

$$E(p) = \frac{1}{2} \sum_{i=1}^n [weight(height_i, p) - weight_i]^2$$

The gradient descent algorithm does this by updating the p_i parameter in the direction of $(\partial/\partial p_i) E(p)$, in particular:

$$p_j = p_j - \text{learning_rate} * \left(\frac{\partial}{\partial p_j} E(p) \right)$$

Here, *learning_rate* determines that the speed of the convergence of $E(p)$ is at the minimum. Updating the p parameter will result in the convergence of $E(p)$ to a certain value, providing that *learning_rate* is sufficiently small. In the Python program, we use a *learning_rate* of 0.000001. However, the drawback of this update rule is that the minimum of $E(p)$ may only be a local minimum.

To update the p parameter programmatically, we need to unfold the partial derivative on $E(p)$. Therefore, we update the p parameter as follows:

$$p_0 := p_0 + \text{learning_rate} * \sum_{i=1}^n [\text{weight}_i - \text{weight}(\text{height}_i, p)]$$

$$p_1 := p_1 + \text{learning_rate} * \sum_{i=1}^n [(\text{weight}_i - \text{weight}(\text{height}_i, p)) * \text{height}_i]$$

We will keep updating the p parameter until it changes only very slightly; that is, until the change in both p_0 and p_1 is less than some constant *acceptable_error*. Once the p parameter stabilizes, we can use it to estimate the weight from the height.

Implementation

```
# source_code/6/regression.py
# Linear regression program to learn a basic linear model.
import math
import sys
sys.path.append('../common')
import common # noqa

# Calculate the gradient by which the parameter should be updated.
def linear_gradient(data, old_parameter):
    gradient = [0.0, 0.0]
    for (x, y) in data:
        term = float(y) - old_parameter[0] - old_parameter[1] * float(x)
        gradient[0] += term
        gradient[1] += term * float(x)
```

```

    return gradient

# This function will apply gradient descent algorithm
# to learn the linear model.
def learn_linear_parameter(data, learning_rate,
                           acceptable_error, LIMIT):
    parameter = [1.0, 1.0]
    old_parameter = [1.0, 1.0]
    for i in range(0, LIMIT):
        gradient = linear_gradient(data, old_parameter)
        # Update the parameter with the Least Mean Squares rule.
        parameter[0] = old_parameter[0] + learning_rate * gradient[0]
        parameter[1] = old_parameter[1] + learning_rate * gradient[1]
        # Calculate the error between the two parameters to compare with
        # the permissible error in order to determine if the calculation
        # is sufficiently accurate.
        if abs(parameter[0] - old_parameter[0]) <= acceptable_error
        and abs(parameter[1] - old_parameter[1]) <= acceptable_error:
            return parameter
        old_parameter[0] = parameter[0]
        old_parameter[1] = parameter[1]
    return parameter

# Calculate the y coordinate based on the linear model predicted.
def predict_unknown(data, linear_parameter):
    for (x, y) in data:
        print(x, linear_parameter[0] + linear_parameter[1] * float(x))

# Program start
csv_file_name = sys.argv[1]
# The maximum number of the iterations in the batch learning algorithm.
LIMIT = 100
# Suitable parameters chosen for the problem given.
learning_rate = 0.0000001
acceptable_error = 0.001

(heading, complete_data, incomplete_data,
 enquired_column) = common.csv_file_to_ordered_data(csv_file_name)
linear_parameter = learn_linear_parameter(
    complete_data, learning_rate, acceptable_error, LIMIT)
print("Linear model:\n(p0,p1)=" + str(linear_parameter) + "\n")
print("Unknowns based on the linear model:")
predict_unknown(incomplete_data, linear_parameter)

```

Input:

We use the data from the table in the preceding example *Weight prediction from height* and save it in a CSV file:

```
# source_code/6/height_weight.csv
height,weight
180,75
174,71
184,83
168,63
178,70
172,?
```

Output:

```
$ python regression.py height_weight.csv
Linear model:
(p0,p1)=[0.9966468959362077, 0.4096393414704317]

Unknowns based on the linear model:
('172', 71.45461362885045)
```

The output for the linear model means that the weight can be expressed in terms of the height as follows:

$$weight = 0.4096393414704317 * height + 0.9966468959362077$$

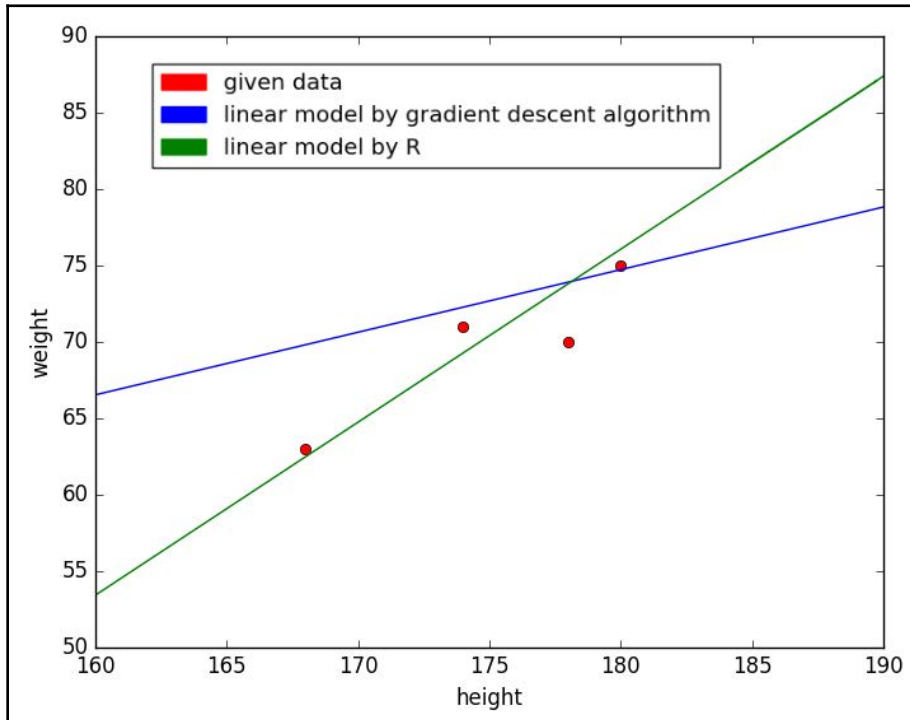
Therefore, a man with a height of 172 cm is predicted to weigh the following:

$$0.4096393414704317 * 172 + 0.9966468959362077 = 71.45461362885045 \text{ } 71.455kg$$

Note that this prediction of 71.455 kg is slightly different from the prediction of 67.016 kg achieved when using the least squares method. This may be due to the fact that the Python gradient descent algorithm found only a local minimum in the prediction.

Visualization – comparison of the least squares method and the gradient descent algorithm

By using weight prediction based on height, we can visualize the linear prediction models of the least squares method and of the gradient descent algorithm that are implemented in Python as follows:



Flight time duration prediction based on distance

Given a table of flights with their origin, destination, and flight time, we would like to estimate the length of a proposed flight from Bratislava, Slovakia, to Amsterdam, the Netherlands:

Origin	Destination	Distance in km	Flight duration	Flight duration in hours
London	Amsterdam	365	1 hr 10 m	1.167
London	Budapest	1462	2 hr 20 m	2.333
London	Bratislava	1285	2 hr 15 m	2.250
Bratislava	Paris	1096	2 hr 5 m	2.083
Bratislava	Berlin	517	1 hr 15 m	2.250
Vienna	Dublin	1686	2 hr 50 m	2.833
Vienna	Amsterdam	932	1 hr 55 m	1.917
Amsterdam	Budapest	1160	2 hr 10 m	2.167
Bratislava	Amsterdam	978	?	?

Analysis

We can reason that the flight duration time consists of two times—the first is the time to take off and the landing time; the second is the time that the airplane moves at a certain speed in the air. The first time is a constant. The second time depends linearly on the speed of the plane, which we assume is similar for all of the flights in the table. Therefore, the flight time can be expressed by using a linear formula in terms of the flight distance.

Input:

```
source_code/6/flight_time.py
import numpy as np
from scipy.linalg import lstsq

distance = np.array([365,1462,1285,1096,517,1686,932,1160])
time = np.array([1.167,2.333,2.250,2.083,2.250,2.833,1.917,2.167])

M = distance[:, np.newaxis]**[0, 1]
model, _, _, _ = lstsq(M,time)
print "Intercept =", model[0]
print "distance coefficient =", model[1]
```

Output:

```
$ python flight_time.py
Intercept = 1.2335890147536381
distance coefficient = 0.0008386790405704925
```

According to linear regression, the time between takeoff and landing for an average flight is about 1.2335890 hours. To travel 1 km with the plane takes 0.0008387 hours; in other words, the speed of an airplane is 1,192 km per hour. The actual usual speed of an airplane for short-distance flights, like the ones in the preceding table, is about 850 km per hour. This leaves room for improvement in our estimation (refer to *Problem 3*).

Therefore, we derived the following formula:

$$\text{flight_time} = 0.0008387 * \text{distance} + 1.2335890 \text{ hours}$$

Using this, we estimate that the flight from Bratislava to Amsterdam, with a distance of 978 km, would take about $0.0008387 * 978 + 1.2335890 = 2.0538376$ hours, or about 2 hours and 3 minutes, which is a little longer than from Vienna to Amsterdam (1 hr 55 m), and a little shorter than from Budapest to Amsterdam (2 hr 10 m).

Ballistic flight analysis – non-linear model

An interplanetary spaceship lands on a planet with a negligible atmosphere and fires three projectiles carrying exploratory bots at the planet, but at different initial velocities. After the bots land on the surface, their distances are measured and the data is recorded, as follows:

Velocity in m/s	Distance in m
400	38,098
600	85,692
800	152,220
?	300,000

At what speed should the projectile carrying the fourth bot be fired in order for it to land 300 km from the spacecraft?

Analysis

For this problem, we need to understand the trajectory of the projectile. Since the atmosphere on the planet is weak, the trajectory is almost equivalent to the ballistic curve without any air drag. The distance, d , traveled by an object fired from a point on the ground, neglecting the curvature of the planet's surface, is given approximately by the following equation:

$$d = v^2 * \sin(2 * \tau) / g$$

Where v is the initial velocity of the object, τ is the angle at which the object was fired, and g is the gravitational force exerted by the planet on the object. Note that the angle, τ , and the gravitational force, g , do not change. Therefore, you can define a constant as $c = \sin(2 * \tau) / g$. This means that the distance traveled on the explored planet can be explained in terms of velocity by the following equation:

$$d = v^2 * c$$

Although d and v are not in a linear relationship, d and the square of v are. Therefore, we can still apply linear regression to determine the relationship between d and v .

Analysis by using the least squares method in Python

Input:

```
source_code/6/speed_distance.py
import numpy as np
from scipy.linalg import lstsq

distance = np.array([38098, 85692, 152220])
squared_speed = np.array([160000, 360000, 640000])

M = distance[:, np.newaxis]**[0, 1]
model, _, _, _ = lstsq(M, squared_speed)
print "Intercept =", model[0]
print "distance coefficient =", model[1]
```

Output:

```
$ python speed_distance.py
Intercept = -317.7078806050511
distance coefficient = 4.206199498720391
```

Therefore, the relationship between the squared velocity and the distance is predicted by the regression as follows:

$$v_2 = 4.206 * d - 317.708$$

The presence of the intercept term may be caused by errors in the measurements, or by other forces in the equation. Since it is relatively small, the final velocity should be estimated reasonably well. Adding the distance of 300 km into the equation, we get the following:

$$v_2 = 4.206 * 300000 - 317.708 = 1261482.292$$

$$v = 1123.157$$

Therefore, for the projectile to reach 300 km from the source, we need to fire it at a speed of approximately 1123.157 m/s.

Summary

In this chapter, we learned about regression analysis. We can think of variables as being dependent on each other in a functional way. For example, the y variable is a function of x denoted by $y=f(x)$. The $f(x)$ function has constant parameters. If y depends on x linearly, then $f(x)=a*x+b$, where a and b are constant parameters in the $f(x)$ function.

We saw that regression is a method to estimate these constant parameters in such a way that the estimated $f(x)$ follows y as closely as possible. This is formally measured by the squared error between $f(x)$ and y for x data samples.

We also covered the gradient descent method, which minimizes this error by updating the constant parameters in the direction of the steepest descent (that is, the partial derivative of the error), ensuring that the parameters converge to the values resulting in minimal errors in the quickest possible way.

Finally, we learned about the `scipy.linalg` Python library which supports the estimation of linear regression using the `lstsq` function based on the least squares method.

In the next chapter, you will apply regression analysis to the analysis of time-dependent data.

Problems

Problem 1: Cloud storage prediction cost: Our software application generates data on a monthly basis and stores this data in cloud storage, together with the data from the previous months. We are given the following bills for cloud storage, and would like to estimate the running costs for the first year of using it:

Months using cloud storage	Monthly bill in euros
1	120.00
2	131.20
3	142.10
4	152.90
5	164.30
1 to 12	?

Problem 2: Fahrenheit and Celsius conversion: In the example that we looked at earlier in this chapter, we devised a formula for converting degrees Fahrenheit to degrees Celsius. Devise a formula converting degrees Celsius into degrees Fahrenheit.

Problem 3: Flight time duration prediction based on distance: Why do you think that a linear regression model resulted in an estimation of the speed of 1,192 km/h, as opposed to the real speed of about 850 km/h? Can you suggest a way to better model the estimation of the flight duration based on flight distances and times?

Problem 4: Bacteria population prediction: A bacteria, *Escherichia coli*, has been observed in the laboratory, and the size of its population was estimated by various measurements at 5-minute intervals, as follows:

Time	Size of population in millions
10:00	47.5
10:05	56.5
10:10	67.2
10:15	79.9
11:00	?

What is the expected number of bacteria at 11:00, assuming that the bacteria continues to grow at the same rate?

Analysis

Problem 1: Every month, we have to pay for the data we have stored in the cloud thus far, plus the new data that is added in that month. We will use linear regression to predict the cost for a general month, and then we will calculate the sum of the first 12 months to calculate the cost for the whole year.

Input:

```
source_code/6/cloud_storage.py
import numpy as np
from scipy.linalg import lstsq

month = np.array([1,2,3,4,5])
bill = np.array([120.0,131.2,142.1,152.9,164.3])

M = month[:, np.newaxis]**[0, 1]
model, _, _, _ = lstsq(M,bill)
print "Intercept =", model[0]
print "month_data =", model[1]
```

Output:

```
$ python cloud_storage.py
Intercept = 109.00999999999992
month_data = 11.030000000000008
```

This means that the base cost is $base_cost=109.01$ euros, and then to store the data added in 1 month costs an additional $month_data=11.03$ euros. Therefore, the formula for the n^{th} monthly bill is as follows:

$$bill_amount = month_data * month_number + base_cost = 11.03 * month_number + 109.01euro$$

Remember that the sum of the first n numbers is $(1/2) * n * (n + 1)$. Thus, the cost for the first n months will be as follows:

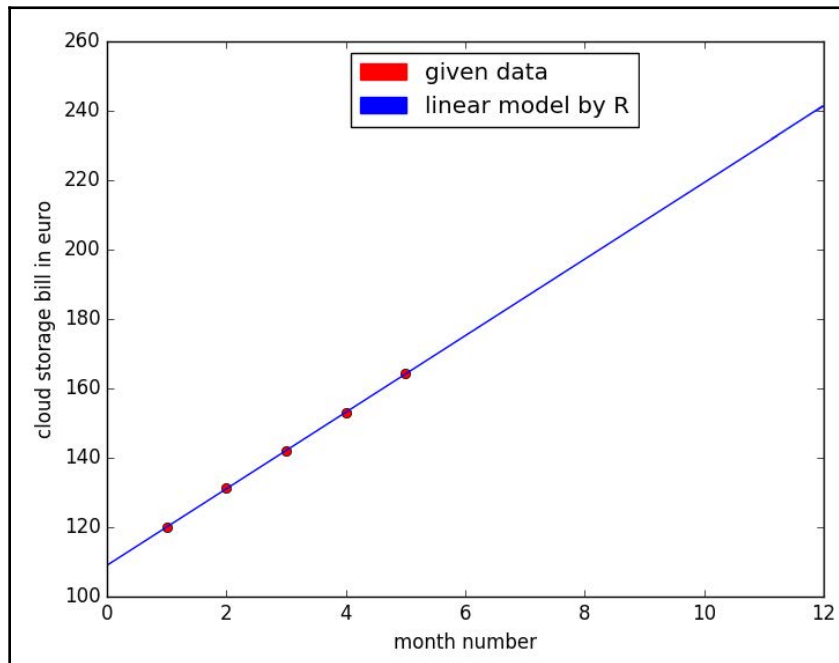
$$\begin{aligned} total_cost(nmonths) &= base_cost * n + month_data * [(1/2) * n * (n + 1)] \\ &= n * [base_cost + month_data * (1/2) * (n + 1)] \\ &= n * [109.01 + 11.03 * (1/2) * (n + 1)] \\ &= n * [114.565 + 5.515 * n] \end{aligned}$$

For the whole year, the cost will be as follows:

$$total_cost(12months) = 12 * [114.565 + 5.515 * 12] = 2168.94euros$$

Visualization

1. In the following graph, we can observe the linearity of the model, which is represented by the blue line. On the other hand, the sum of the points on the linear line is quadratic in nature and is represented by the area beneath the line:



Problem 2: There are many ways to obtain the formula for converting degrees Celsius into degrees Fahrenheit. We could use the least squares method and, from the initial Python file, take the following lines:

```
M = fahrenheit[:, np.newaxis]**[0, 1]
model, _, _ = lstsq(M, celsius)
```

We then change them to the following:

```
M = celsius[:, np.newaxis]**[0, 1]
model, _ , _ = lstsq(M, fahrenheit)
```

Then, we would obtain the desired reversed model:

```
Intercept = 32.000000000000014
celsius = 1.7999999999999998
```

Therefore, degrees Fahrenheit can be expressed from degrees Celsius as follows:

$$F = 1.8 * C + 32$$

Alternatively, we may obtain the preceding formula by modifying the following one:

$$C = (5/9) * F - 160/9$$

$$160/9 + C = (5/9) * F$$

$$160 + 9 * C = 5 * F$$

Problem 3: The estimated speed is so high because even flights over a short distance take quite a long time; for example, the flight from London to Amsterdam, where the distance between the two cities is only 365 km, takes about 1.167 hours. But, on the other hand, if the distance changes only slightly, then the flight time changes only slightly as well. This results in us estimating a very long initial setup time. Consequently, the speed has to be very fast because there is only a small amount of time left to travel a certain distance.

If we consider very long flights, where the initial setup time to flight time ratio is much smaller, we could predict the flight speed more accurately.

Problem 4: The number of bacteria at the 5-minute intervals is 47.5, 56.5, 67.2, and 79.9 million. The differences between these numbers are 9, 10.7, and 12.7. The sequence is increasing. Therefore, we need to look at the ratios of the neighbor terms to see how the sequence grows: $56.5/47.5=1.18947$, $67.2/56.5=1.18938$, and $79.9/67.2=1.18899$. The ratios of the successive terms are close to each other, so we have reason to believe that the number of bacteria in the growing population can be estimated by using the exponential distribution of the model:

$$n = 47.7 * b^m$$

Where n is the number of bacteria in millions, b is a constant (the base), the letter m is the exponent expressing the number of minutes since 10:00, which was the time of the first measurement, and 47.7 is the number of bacteria at that time in millions.

To estimate the constant, b , we use the ratios between the sequence terms. We know that b^5 is approximately $(56.5/47.5 + 67.2/56.5 + 79.9/67.2)/3 = 1.18928$. Therefore, the constant, b , is approximately $b = 1.18928^{1/5} = 1.03528$. Thus, the number of bacteria in millions is as follows:

$$n = 47.7 * 1.03528^m$$

At 11:00, which is 60 minutes later than 10:00, the estimated number of bacteria is as follows:

$$47.7 * 1.03528^{60} = 381.9 \quad 47.7 * 1.03528^{60} = 381.9 \text{ million}$$

7 Time Series Analysis

Time series analysis is the analysis of time-dependent data. Given data for a certain period, the aim is to predict data for a different period, usually in the future. For example, time series analysis is used to predict financial markets, earthquakes, and weather.

In this chapter, we are mostly concerned with predicting the numerical values of certain quantities, for example, the human population in 2030.

The main elements of time-based prediction are as follows:

- **Trends:** Does the variable tend to rise or fall as time passes? For example, does the human population grow or shrink?
- **Seasonality:** How is the data dependent on certain regular events in time? For example, are restaurant profits greater on Fridays than on Tuesdays?

Combining these two elements of time series analysis equips us with a powerful method to make time-dependent predictions.

In this chapter, you will learn about the following topics:

- Analyzing data trends using regression through a business profits example
- Observing and analyzing recurring patterns in data in the form of seasonality through an electronics shop's sales example
- Using the example of an electronics shop's sales, you will learn how to combine the analysis of trends and seasonality to predict time-dependent data
- Creating time-dependent models in Python using the examples of business profits and the electronics shop's sales

Business profits – analyzing trends

We are interested in predicting the profits of a business for the year 2018 given its profits for previous years:

Year	Profit in USD
2011	\$40,000
2012	\$43,000
2013	\$45,000
2014	\$50,000
2015	\$54,000
2016	\$57,000
2017	\$59,000
2018	?

Analysis

In this example, the profit is always increasing, so we can think of representing the profit as a growing function that's dependent on the time variable, which is represented by years. The variations in profit between the subsequent years are \$3,000, \$2,000, \$5,000, \$4,000, \$3,000, and \$2,000. These differences do not seem to be affected by time, and the variation between them is relatively low. Therefore, we may try to predict the profit for the coming years by performing linear regression. We express profit, p , in terms of the year, y , in a linear equation, also called a trend line:

$$profit = a * year + b$$

We can find the constants, a and b , using linear regression.

Analyzing trends using the least squares method in Python

Input:

We store the data from the preceding table in the `year` and `profit` vectors in the Python code:

```
# source_code/7/profit_year.py
```

```
#Predicting profit based on the year
import numpy as np
from scipy.linalg import lstsq

year = np.array([2011,2012,2013,2014,2015,2016,2017])
profit = np.array([40,43,45,50,54,57,59])

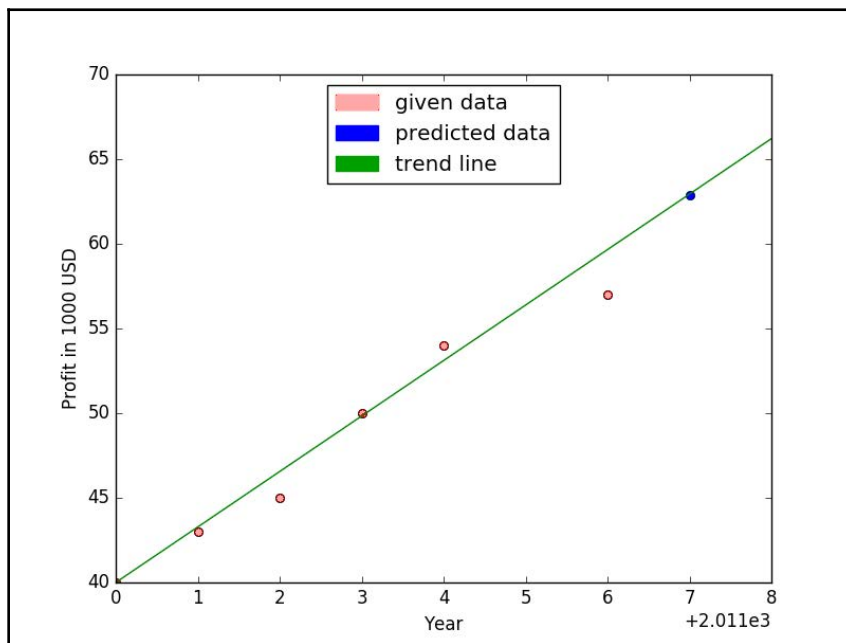
M = year[:, np.newaxis]**[0, 1]
model, _, _, _ = lstsq(M,profit)
print "Intercept =", model[0]
print "year coefficient =", model[1]
```

Output:

```
$ python profit_year.py
Intercept = -6711.571428570175
year coefficient = 3.3571428571422364
```

Visualization

Now, we add the trend line to the graph:



Business profits - data analysis

Conclusion

The trend line equation for the company profit is as follows:

$$profit = 3.357 * year - 6711.571$$

From this equation, we can predict the profit for the year 2018 to be as follows:

$$\begin{aligned} profit &= 3.357 * 2018 - 6711.571 \\ &= 62.855 \text{ k USD or } 62855 \text{ USD} \end{aligned}$$

This example was simple—we were able to make a prediction just by using linear regression on the trend line.

In the next example, we will look at data that is subject to both trends and seasonality.

Electronics shop's sales – analyzing seasonality

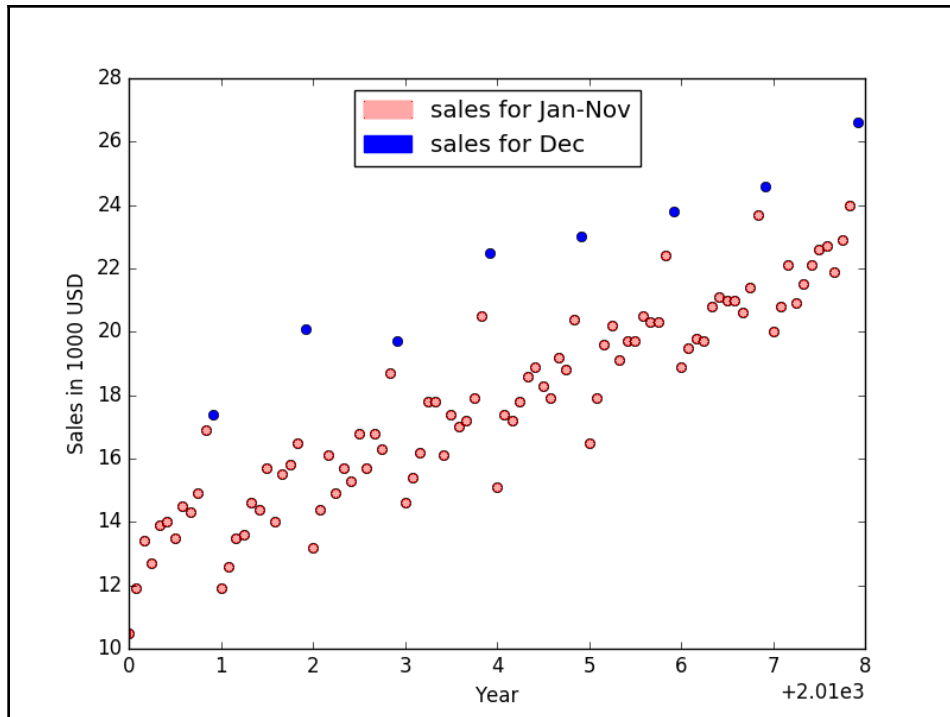
We have data for sales in thousands of USD for a small electronics shop, by month, for the years 2010 to 2017. We would like to predict sales for each month of 2018:

Month/Year	2010	2011	2012	2013	2014	2015	2016	2017	2018
January	10.5	11.9	13.2	14.6	15.1	16.5	18.9	20	
February	11.9	12.6	14.4	15.4	17.4	17.9	19.5	20.8	
March	13.4	13.5	16.1	16.2	17.2	19.6	19.8	22.1	
April	12.7	13.6	14.9	17.8	17.8	20.2	19.7	20.9	
May	13.9	14.6	15.7	17.8	18.6	19.1	20.8	21.5	
June	14	14.4	15.3	16.1	18.9	19.7	21.1	22.1	
July	13.5	15.7	16.8	17.4	18.3	19.7	21	22.6	
August	14.5	14	15.7	17	17.9	20.5	21	22.7	
September	14.3	15.5	16.8	17.2	19.2	20.3	20.6	21.9	
October	14.9	15.8	16.3	17.9	18.8	20.3	21.4	22.9	
November	16.9	16.5	18.7	20.5	20.4	22.4	23.7	24	
December	17.4	20.1	19.7	22.5	23	23.8	24.6	26.6	

The data in the table is sales in thousands of USD

Analysis

To be able to analyze this, we will first graph the data so that we can observe patterns and analyze them:



From the graph and table, we notice that, in the long term, sales increase linearly. This is the trend. However, we can also see that the sales for December tend to be higher than any other month. Thus, we have reason to believe that sales are also influenced by the month.

How can we predict the monthly sales for the following years? First, we determine the exact long-term trend of the data. Then, we analyze the change across the months.

Analyzing trends using the least squares method in Python

Input:

The year list contains the periods of the year represented as a decimal number $year+month/12$. The sales list contains the sales in thousands of USD for the corresponding periods in the year list. We will use linear regression to establish the trend line. From the initial graph, we observe that the trend is linear in nature:

```
# source_code/6/sales_year.py
#Predicting sales based on the period in the year
import numpy as np
from scipy.linalg import lstsq

year = np.array([2010.000000, 2010.083333, 2010.166667, 2010.250000,
 2010.333333, 2010.416667, 2010.500000, 2010.583333,
 2010.666667, 2010.750000, 2010.833333, 2010.916667,
 2011.000000, 2011.083333, 2011.166667, 2011.250000,
 2011.333333, 2011.416667, 2011.500000, 2011.583333,
 2011.666667, 2011.750000, 2011.833333, 2011.916667,
 2012.000000, 2012.083333, 2012.166667, 2012.250000,
 2012.333333, 2012.416667, 2012.500000, 2012.583333,
 2012.666667, 2012.750000, 2012.833333, 2012.916667,
 2013.000000, 2013.083333, 2013.166667, 2013.250000,
 2013.333333, 2013.416667, 2013.500000, 2013.583333,
 2013.666667, 2013.750000, 2013.833333, 2013.916667,
 2014.000000, 2014.083333, 2014.166667, 2014.250000,
 2014.333333, 2014.416667, 2014.500000, 2014.583333,
 2014.666667, 2014.750000, 2014.833333, 2014.916667,
 2015.000000, 2015.083333, 2015.166667, 2015.250000,
 2015.333333, 2015.416667, 2015.500000, 2015.583333,
 2015.666667, 2015.750000, 2015.833333, 2015.916667,
 2016.000000, 2016.083333, 2016.166667, 2016.250000,
 2016.333333, 2016.416667, 2016.500000, 2016.583333,
 2016.666667, 2016.750000, 2016.833333, 2016.916667,
 2017.000000, 2017.083333, 2017.166667, 2017.250000,
 2017.333333, 2017.416667, 2017.500000, 2017.583333,
 2017.666667, 2017.750000, 2017.833333, 2017.916667])

sale = np.array([10.500000, 11.900000, 13.400000, 12.700000, 13.900000,
 14.000000, 13.500000, 14.500000, 14.300000, 14.900000,
 16.900000, 17.400000, 11.900000, 12.600000, 13.500000,
 13.600000, 14.600000, 14.400000, 15.700000, 14.000000,
 15.500000, 15.800000, 16.500000, 20.100000, 13.200000,
 14.400000, 16.100000, 14.900000, 15.700000, 15.300000,
 16.800000, 15.700000, 16.800000, 16.300000, 18.700000,
 19.700000, 14.600000, 15.400000, 16.200000, 17.800000,
```

```

17.800000, 16.100000, 17.400000, 17.000000, 17.200000,
17.900000, 20.500000, 22.500000, 15.100000, 17.400000,
17.200000, 17.800000, 18.600000, 18.900000, 18.300000,
17.900000, 19.200000, 18.800000, 20.400000, 23.000000,
16.500000, 17.900000, 19.600000, 20.200000, 19.100000,
19.700000, 19.700000, 20.500000, 20.300000, 20.300000,
22.400000, 23.800000, 18.900000, 19.500000, 19.800000,
19.700000, 20.800000, 21.100000, 21.000000, 21.000000,
20.600000, 21.400000, 23.700000, 24.600000, 20.000000,
20.800000, 22.100000, 20.900000, 21.500000, 22.100000,
22.600000, 22.700000, 21.900000, 22.900000, 24.000000,
26.600000])

```

```

M = year[:, np.newaxis]**[0, 1]
model, _, _, _ = lstsq(M, sale)
print "Intercept =", model[0]
print "year coefficient =", model[1]

```

Output:

```

$ python sales_year.py
Intercept = -2557.777649371433
year coefficient = 1.2789880745818643

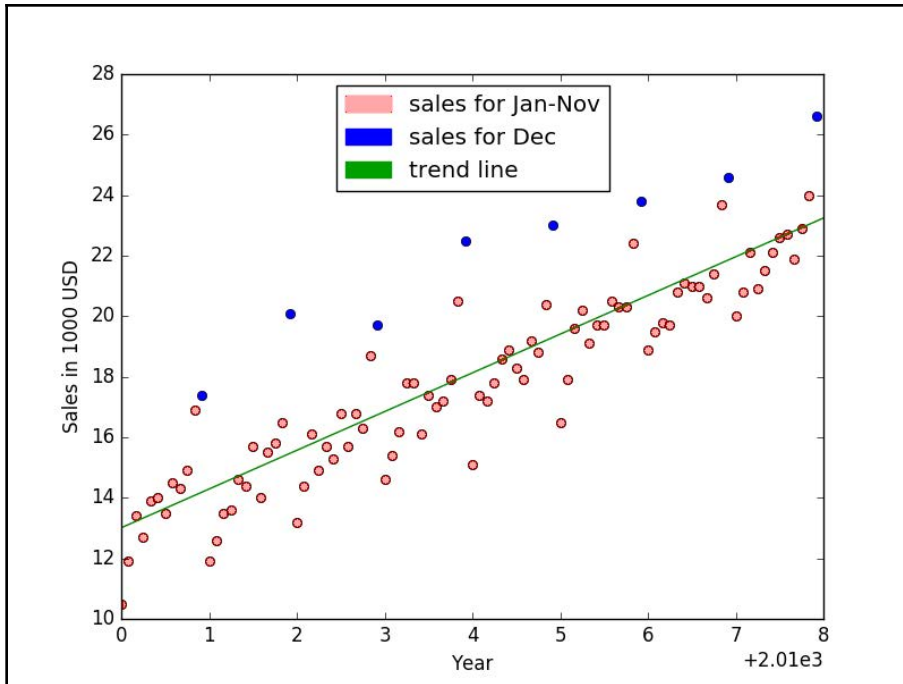
```

Therefore, the equation for the trend line is as follows:

$$sales = 1.279 * year - 2557.778$$

Visualization

Now, we add the trend line to the graph:



Analyzing seasonality

Now, we can analyze seasonality—that is, how data changes across months. From our observations, we know that, for some months, sales tend to be higher, whereas for other months, sales tend to be lower. We evaluate the differences between the linear trend and actual sales. Based on the pattern observed in these differences, we produce a model of seasonality to predict sales more accurately for each month:

Sales for January									
Year	2010	2011	2012	2013	2014	2015	2016	2017	Average
Actual sales	10.5	11.9	13.2	14.6	15.1	16.5	18.9	20	
Sales on the trend line	13.012	14.291	15.57	16.849	18.128	19.407	20.686	21.965	
Difference	-2.512	-2.391	-2.37	-2.249	-3.028	-2.907	-1.786	-1.965	-2.401

Sales for February									
Year	2010	2011	2012	2013	2014	2015	2016	2017	Average
Actual sales	11.9	12.6	14.4	15.4	17.4	17.9	19.5	20.8	
Sales on the trend line	13.1185833333	14.3975833333	15.6765833333	16.9555833333	18.2345833333	19.5135833333	20.7925833333	22.0715833333	
Difference	-1.2185833333	-1.7975833333	-1.2765833333	-1.5555833333	-0.8345833333	-1.6135833333	-1.2925833333	-1.2715833333	-1.3575833333
Sales for March									
Year	2010	2011	2012	2013	2014	2015	2016	2017	Average
Actual sales	13.4	13.5	16.1	16.2	17.2	19.6	19.8	22.1	
Sales on the trend line	13.2251666667	14.5041666667	15.7831666667	17.0621666667	18.3411666667	19.6201666667	20.8991666667	22.1781666667	
Difference	0.1748333333	-1.0041666667	0.3168333333	-0.8621666667	-1.1411666667	-0.0201666667	-1.0991666667	-0.0781666667	-0.4641666667
Sales for April									
Year	2010	2011	2012	2013	2014	2015	2016	2017	Average
Actual sales	12.7	13.6	14.9	17.8	17.8	20.2	19.7	20.9	
Sales on the trend line	13.33175	14.61075	15.88975	17.16875	18.44775	19.72675	21.00575	22.28475	
Difference	-0.63175	-1.01075	-0.98975	0.63125	-0.64775	0.47325	-1.30575	-1.38475	-0.60825
Sales for May									
Year	2010	2011	2012	2013	2014	2015	2016	2017	Average
Actual sales	13.9	14.6	15.7	17.8	18.6	19.1	20.8	21.5	
Sales on the trend line	13.4383333333	14.7173333333	15.9963333333	17.2753333333	18.5543333333	19.8333333333	21.1123333333	22.3913333333	
Difference	0.4616666667	-0.1173333333	-0.2963333333	0.5246666667	0.0456666667	-0.7333333333	-0.3123333333	-0.8913333333	-0.1648333333
Sales for June									
Year	2010	2011	2012	2013	2014	2015	2016	2017	Average
Actual sales	14	14.4	15.3	16.1	18.9	19.7	21.1	22.1	
Sales on the trend line	13.5449166667	14.8239166667	16.1029166667	17.3819166667	18.6609166667	19.9399166667	21.2189166667	22.4979166667	
Difference	0.4550833333	-0.4239166667	-0.8029166667	-1.2819166667	0.2390833333	-0.2399166667	-0.1189166667	-0.3979166667	-0.3214166667
Sales for July									
Year	2010	2011	2012	2013	2014	2015	2016	2017	Average
Actual sales	13.5	15.7	16.8	17.4	18.3	19.7	21	22.6	
Sales on the trend line	13.6515	14.9305	16.2095	17.4885	18.7675	20.0465	21.3255	22.6045	
Difference	-0.1515	0.7695	0.5905	-0.0885	-0.4675	-0.3465	-0.3255	-0.0045	-0.003
Sales for August									

Year	2010	2011	2012	2013	2014	2015	2016	2017	Average
Actual sales	14.5	14	15.7	17	17.9	20.5	21	22.7	
Sales on the trend line	13.7580833333	15.0370833333	16.3160833333	17.5950833333	18.8740833333	20.1530833333	21.4320833333	22.7110833333	
Difference	0.7419166667	-1.0370833333	-0.6160833333	-0.5950833333	-0.9740833333	0.3469166667	-0.4320833333	-0.0110833333	-0.3220833333
Sales for September									
Year	2010	2011	2012	2013	2014	2015	2016	2017	Average
Actual sales	14.3	15.5	16.8	17.2	19.2	20.3	20.6	21.9	
Sales on the trend line	13.8646666667	15.1436666667	16.4226666667	17.7016666667	18.9806666667	20.2596666667	21.5386666667	22.8176666667	
Difference	0.4353333333	0.3563333333	0.3773333333	-0.5016666667	0.2193333333	0.0403333333	-0.9386666667	-0.9176666667	-0.1161666667
Sales for October									
Year	2010	2011	2012	2013	2014	2015	2016	2017	Average
Actual sales	14.9	15.8	16.3	17.9	18.8	20.3	21.4	22.9	
Sales on the trend line	13.97125	15.25025	16.52925	17.80825	19.08725	20.36625	21.64525	22.92425	
Difference	0.92875	0.54975	-0.22925	0.09175	-0.28725	-0.06625	-0.24525	-0.02425	0.08975
Sales for November									
Year	2010	2011	2012	2013	2014	2015	2016	2017	Average
Actual sales	16.9	16.5	18.7	20.5	20.4	22.4	23.7	24	
Sales on the trend line	14.0778333333	15.3568333333	16.6358333333	17.9148333333	19.1938333333	20.4728333333	21.7518333333	23.0308333333	
Difference	2.8221666667	1.1431666667	2.0641666667	2.5851666667	1.2061666667	1.9271666667	1.9481666667	0.9691666667	1.8331666667
Sales for December									
Year	2010	2011	2012	2013	2014	2015	2016	2017	Average
Actual sales	17.4	20.1	19.7	22.5	23	23.8	24.6	26.6	
Sales on the trend line	14.1844166667	15.4634166667	16.7424166667	18.0214166667	19.3004166667	20.5794166667	21.8584166667	23.1374166667	
Difference	3.2155833333	4.6365833333	2.9575833333	4.4785833333	3.6995833333	3.2205833333	2.7415833333	3.4625833333	3.5515833333

We cannot observe any obvious trends in the differences between the actual sales and the sales on the trend line. Therefore, we just calculate the arithmetic mean of these differences for every month.

For example, we notice that the sales in December tend to be higher by about \$3,551.58 compared to the sales predicted on the trend line. Similarly, the sales for January tend to be lower, on average by \$2,401, compared to the sales predicted on the trend line.

Making the assumption that the month has an impact on the actual sales from our observations of the variation in sales across the months, we take our prediction rule:

$$sales = 1.279 * year - 2557.778$$

And we then update it to the new rule:

$$sales = 1.279 * year - 2557.778 + month_difference$$

Here, *sales* is the amount of sales for a chosen month and year in the prediction, and *month_difference* is the average difference in our given data between actual sales and sales on the trend line. More specifically, we get the following 12 equations and predictions for sales for the year 2018 in thousands of USD:

$$sales_january = 1.279 * (year + 0/12) - 2557.778 - 2.401$$

$$= 1.279 * (2018 + 0/12) - 2557.778 - 2.401 = 20.843$$

$$sales_february = 1.279 * (year + 1/12) - 2557.778 - 1.358$$

$$= 1.279 * (2018 + 1/12) - 2557.778 - 1.358 = 21.993$$

$$sales_march = 1.279 * (year + 2/12) - 2557.778 - 0.464$$

$$= 1.279 * (2018 + 2/12) - 2557.778 - 0.464 = 22.993$$

$$sales_april = 1.279 * (year + 3/12) - 2557.778 - 0.608$$

$$= 1.279 * (2018 + 3/12) - 2557.778 - 0.608 = 22.956$$

$$sales_may = 1.279 * (year + 4/12) - 2557.778 - 0.165$$

$$= 1.279 * (2018 + 4/12) - 2557.778 - 0.165 = 23.505$$

$$sales_june = 1.279 * (year + 5/12) - 2557.778 - 0.321$$

$$= 1.279 * (2018 + 5/12) - 2557.778 - 0.321 = 23.456$$

$$sales_july = 1.279 * (year + 6/12) - 2557.778 - 0.003$$

$$= 1.279 * (2018 + 6/12) - 2557.778 - 0.003 = 23.881$$

$$sales_august = 1.279 * (year + 7/12) - 2557.778 - 0.322$$

$$= 1.279 * (2018 + 7/12) - 2557.778 - 0.322 = 23.668$$

$$sales_september = 1.279 * (year + 8/12) - 2557.778 - 0.116$$

$$= 1.279 * (2018 + 8/12) - 2557.778 - 0.116 = 23.981$$

$$\text{sales_october} = 1.279 * (\text{year} + 9/12) - 2557.778 + 0.090$$

$$= 1.279 * (2018 + 9/12) - 2557.778 + 0.090 = 24.293$$

$$\text{sales_november} = 1.279 * (\text{year} + 10/12) - 2557.778 + 1.833$$

$$= 1.279 * (2018 + 10/12) - 2557.778 + 1.833 = 26.143$$

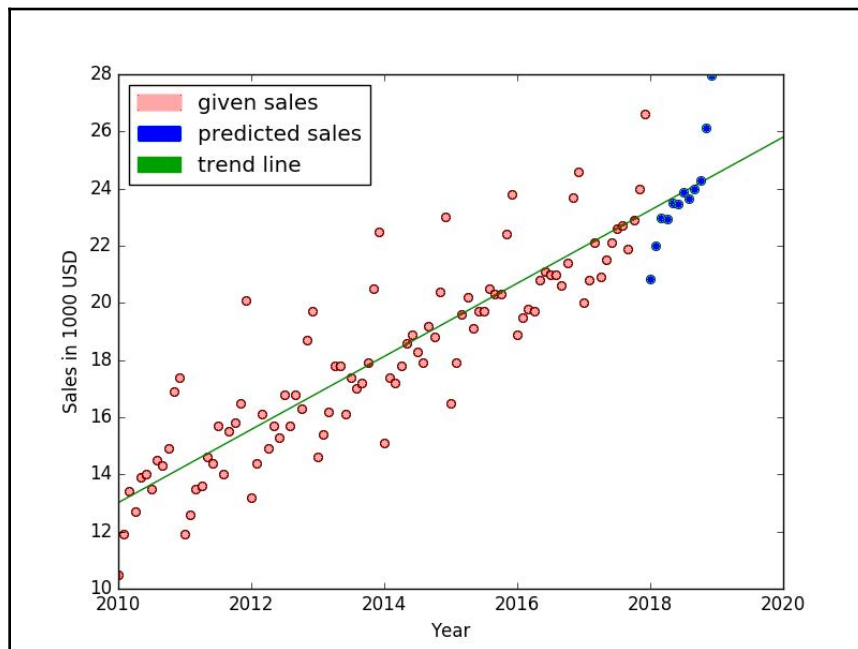
$$\text{sales_december} = 1.279 * (\text{year} + 11/12) - 2557.778 + 3.552$$

$$= 1.279 * (2018 + 11/12) - 2557.778 + 3.552 = 27.968$$

Conclusion

The table is complete with the sales for the year 2018, based on the preceding seasonal equations.

We can visualize the data predicted on a graph:



Summary

In this chapter, we studied the analysis of time-dependent data. The two most important factors in this analysis are trends and seasonality.

The analysis of trends can be considered as determining the function around which the data is distributed. Using the fact that data is dependent on time, this function can be determined using regression. Many phenomena have a linear trend line, whereas others may not follow a linear pattern.

We also learned that the analysis of seasonality tries to detect regular patterns occurring in time repeatedly, such as higher sales before Christmas. To detect a seasonal pattern, it is essential to divide data into the different seasons in such a way that a pattern reoccurs in the same season. This division can divide a year into months, a week into days or into workdays and the weekend, and so on. An appropriate division into seasons and analyzing patterns in those is the key to good seasonal analysis.

Once trend and seasonality have been analyzed in the data, the combined result is a prediction for the pattern that the time-dependent data will follow in the future.

This chapter represents the last method of data analysis that's presented in this book. After this chapter, you can find reference material on statistics, Python programming language concepts, and a glossary of algorithms and methods in data science.

Problems

Problem 1: Determining the trend for Bitcoin prices:

a) We are given a table containing Bitcoin prices in USD for the years 2010-2017. Determine a linear trend line for these prices. The monthly price is taken from the first day in each month:

Date: year-month-day	Bitcoin price in USD
2010-12-01	0.23
2011-06-01	9.57
2011-12-01	3.06
2012-06-01	5.27
2012-12-01	12.56
2013-06-01	129.3
2013-12-01	946.92
2014-06-01	629.02

2014-12-01	378.64
2015-06-01	223.31
2015-12-01	362.73
2016-06-01	536.42
2016-12-01	753.25
2017-06-01	2,452.18

b) As per the linear trend line from part a), what is the expected price of Bitcoin in 2020?

c) Discuss whether a linear line is a good indicator for the future price of Bitcoin.

Problem 2: Electronics shop's sales: Using the data in the electronics shop's sales example, predict the sales for every month of the year 2019.

Analysis

Problem 1:

Input:

```
source_code/7/year_bitcoin.py
#Determining a linear trend line for Bitcoin
import numpy as np
from scipy.linalg import lstsq

year = np.array([2010.9166666666666, 2011.4166666666666, 2011.9166666666666,
                 2012.4166666666666, 2012.9166666666666, 2013.4166666666666,
                 2013.9166666666666, 2014.4166666666666, 2014.9166666666666,
                 2015.4166666666666, 2015.9166666666666, 2016.4166666666666,
                 2016.9166666666666, 2017.4166666666666])
btc_price = np.array([0.23, 9.57, 3.06, 5.27, 12.56, 129.3, 946.92, 629.02,
                     378.64, 223.31, 362.73, 536.42, 753.25, 2452.18])

M = year[:, np.newaxis]**[0, 1]
model, _, _, _ = lstsq(M, btc_price)
print "Intercept =", model[0]
print "year coefficient =", model[1]
```

Output:

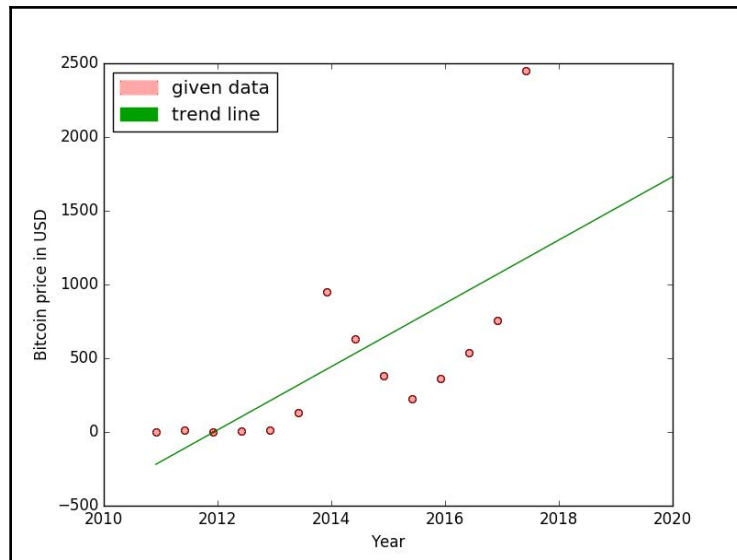
```
$ python year_bitcoin.py
Intercept = -431962.903846178
year coefficient = 214.69081318682586
```

Trend line:

From the output of the Python code, we find that the linear trend line for the price of Bitcoin in USD is as follows:

$$price = year * 214.7 - 431962.9$$

This gives us the following graph for the trend line:



As per the trend line, the expected price for Bitcoin for January 1, 2020 is \$1,731.10.

A linear trend line is probably not a good indicator or price predictor for Bitcoin. This is because of the many factors in play and because of the potential exponential nature often seen in trends in technology; for example, the number of active Facebook users and the number of transistors in the best consumer CPU under USD 1,000.

There are three important factors that could facilitate an exponential adoption of Bitcoin and thus drive its price upward:

- **Technological maturity (scalability):** The number of transactions per second can ensure an instant transfer, even though many people use Bitcoin to make and receive payments
- **Stability:** Once sellers are not afraid to lose their profits if they receive payments in Bitcoin, they are more open to accept it as currency
- **User-friendliness:** Once ordinary users can make and receive payments in Bitcoin in a natural way, there will not be a technical barrier to using Bitcoin as they would any other currency they are used to

To analyze the price of Bitcoin, we would have to take much more data into consideration and it is likely that its price will not follow a linear trend.

Problem 2: We use the 12 formulas from the example, one for each month, to predict the sales for each month in the year 2019:

$$\text{sales_january} = 1.279 * (\text{year} + 0/12) - 2557.778 - 2.401$$

$$= 1.279 * (2019 + 0/12) - 2557.778 - 2.401 = 22.122$$

$$\text{sales_february} = 1.279 * (2019 + 1/12) - 2557.778 - 1.358 = 23.272$$

$$\text{sales_march} = 1.279 * (2019 + 2/12) - 2557.778 - 0.464 = 24.272$$

$$\text{sales_april} = 1.279 * (2019 + 3/12) - 2557.778 - 0.608 = 24.234$$

$$\text{sales_may} = 1.279 * (2019 + 4/12) - 2557.778 - 0.165 = 24.784$$

$$\text{sales_june} = 1.279 * (2019 + 5/12) - 2557.778 - 0.321 = 24.735$$

$$\text{sales_july} = 1.279 * (2019 + 6/12) - 2557.778 - 0.003 = 25.160$$

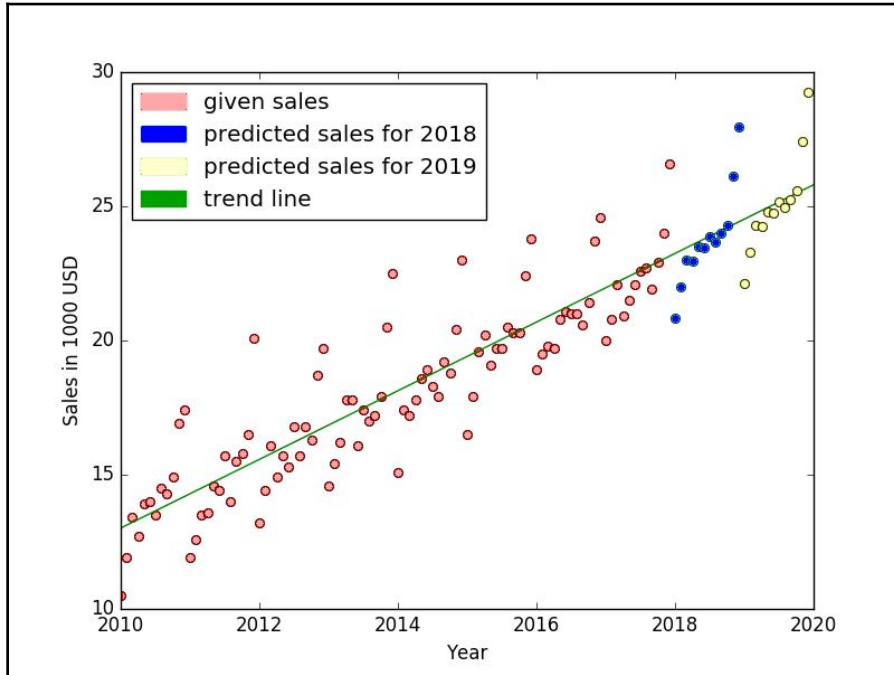
$$\text{sales_august} = 1.279 * (2019 + 7/12) - 2557.778 - 0.322 = 24.947$$

$$\text{sales_september} = 1.279 * (2019 + 8/12) - 2557.778 - 0.116 = 25.259$$

$$\text{sales_october} = 1.279 * (2019 + 9/12) - 2557.778 + 0.090 = 25.572$$

$$\text{sales_november} = 1.279 \cdot (2019 + 10/12) - 2557.778 + 1.833 = 27.422$$

$$\text{sales_december} = 1.279 \cdot (2019 + 11/12) - 2557.778 + 3.552 = 29.247$$



Electronics shop's sales - data analysis

Python Reference

Introduction

Python is a general-purpose programming and scripting language. Its simplicity and extensive libraries make it possible to develop an application that is compatible with the modern requirements of technology quickly.

Python code is written in files with the suffix `.py`, and can be executed with the `python` command.

Python Hello World example

The simplest program in Python prints a single line of text.

Input:

```
source_code/appendix_c_python/example00_helloworld.py
print "Hello World!"
```

Output:

```
$ python example00_helloworld.py
Hello World!
```

Comments

Comments are not executed in Python. They start with the `#` character, and end with the end of the line.

Input:

```
# source_code/appendix_c_python/example01_comments.py
print "This text will be printed because the print statement is executed."
#This is just a comment and will not be executed.
#print "Even commented statements are not executed."
print "But the comment finished with the end of the line."
print "So the 4th and 5th line of the code are executed again."
```

Output:

```
$ python example01_comments.py
```

```
This text will be printed because the print statement is executed  
But the comment finished with the end of the line.  
So the 4th and 5th line of the code are executed again.
```

Data types

Some of the data types available in Python are as follows:

- **Numeric data types:** int and float
- **Text data types:** str
- **Composite data types:** tuple, list, set, and dictionary

int

The int data type can hold only integer values.

Input:

```
# source_code/appendix_c_python/example02_int.py  
rectangle_side_a = 10  
rectangle_side_b = 5  
rectangle_area = rectangle_side_a * rectangle_side_b  
rectangle_perimeter = 2*(rectangle_side_a + rectangle_side_b)  
print "Let there be a rectangle with the sides of lengths:"  
print rectangle_side_a, "and", rectangle_side_b, "cm."  
print "Then the area of the rectangle is", rectangle_area, "cm squared."  
print "The perimeter of the rectangle is", rectangle_perimeter, "cm."
```

Output:

```
$ python example02_int.py
```

```
Let there be a rectangle with the sides of lengths: 10 and 5 cm.  
Then the area of the rectangle is 50 cm squared.  
The perimeter of the rectangle is 30 cm.
```

float

The `float` data type can also hold non-integer rational values.

Input:

```
# source_code/appendix_c_python/example03_float.py
pi = 3.14159
circle_radius = 10.2
circle_perimeter = 2 * pi * circle_radius
circle_area = pi * circle_radius * circle_radius
print "Let there be a circle with the radius", circle_radius, "cm."
print "Then the perimeter of the circle is", circle_perimeter, "cm."
print "The area of the circle is", circle_area, "cm squared."
```

Output:

```
$ python example03_float.py
Let there be a circle with the radius 10.2 cm.
Then the perimeter of the circle is 64.088436 cm.
The area of the circle is 326.8510236 cm squared.
```

String

A string variable can be used to store text.

Input:

```
# source_code/appendix_c_python/example04_string.py
first_name = "Satoshi"
last_name = "Nakamoto"
full_name = first_name + " " + last_name
print "The inventor of Bitcoin is", full_name, "."
```

Output:

```
$ python example04_string.py
The inventor of Bitcoin is Satoshi Nakamoto .
```

Tuple

A tuple data type is analogous to a vector in mathematics; for example, `tuple = (integer_number, float_number)`.

Input:

```
# source_code/appendix_c_python/example05_tuple.py
import math

point_a = (1.2, 2.5)
point_b = (5.7, 4.8)
#math.sqrt computes the square root of a float number.
#math.pow computes the power of a float number.
segment_length = math.sqrt(
    math.pow(point_a[0] - point_b[0], 2) +
    math.pow(point_a[1] - point_b[1], 2))
print "Let the point A have the coordinates", point_a, "cm."
print "Let the point B have the coordinates", point_b, "cm."
print "Then the length of the line segment AB is", segment_length, "cm."
```

Output:

```
$ python example05_tuple.py
Let the point A have the coordinates (1.2, 2.5) cm.
Let the point B have the coordinates (5.7, 4.8) cm.
Then the length of the line segment AB is 5.0537115074 cm.
```

List

A list in Python is an ordered set of values.

Input:

```
# source_code/appendix_c_python/example06_list.py
some_primes = [2, 3]
some_primes.append(5)
some_primes.append(7)
print "The primes less than 10 are:", some_primes
```

Output:

```
$ python example06_list.py
The primes less than 10 are: [2, 3, 5, 7]
```

Set

A Set in Python is a non-ordered mathematical set of values.

Input:

```
# source_code/appendix_c_python/example07_set.py
from sets import Set

boys = Set(['Adam', 'Samuel', 'Benjamin'])
girls = Set(['Eva', 'Mary'])
teenagers = Set(['Samuel', 'Benjamin', 'Mary'])
print 'Adam' in boys
print 'Jane' in girls
girls.add('Jane')
print 'Jane' in girls
teenage_girls = teenagers & girls #intersection
mixed = boys | girls #union
non_teenage_girls = girls - teenage_girls #difference
print teenage_girls
print mixed
print non_teenage_girls
```

Output:

```
$ python example07_set.py
True
False
True
Set(['Mary'])
Set(['Benjamin', 'Adam', 'Jane', 'Eva', 'Samuel', 'Mary'])
Set(['Jane', 'Eva'])
```

Dictionary

A dictionary is a data structure that can store values by their keys.

Input:

```
# source_code/appendix_c_python/example08_dictionary.py
dictionary_names_heights = {}
dictionary_names_heights['Adam'] = 180.
dictionary_names_heights['Benjamin'] = 187
dictionary_names_heights['Eva'] = 169
print 'The height of Eva is', dictionary_names_heights['Eva'], 'cm.'
```

Output:

```
$ python example08_dictionary.py
The height of Eva is 169 cm.
```

Flow control

The Python programming language controls the flow of program execution through the use of conditionals, for loops (including the statements `break` and `continue` within them), and functions.

Conditionals

If a particular condition is met, a certain amount of code can be executed using the `if` statement. If the condition is not met, then we can execute the code following the `else` statement. If the first condition is not met, we can set the next condition for the code to be executed using the `elif` statement.

Input:

```
# source_code/appendix_c_python/example09_if_else_elif.py
x = 10
if x == 10:
    print 'The variable x is equal to 10.'

if x > 20:
    print 'The variable x is greater than 20.'
else:
    print 'The variable x is not greater than 20.'

if x > 10:
    print 'The variable x is greater than 10.'
elif x > 5:
    print 'The variable x is not greater than 10, but greater ' +
        'than 5.'
else:
    print 'The variable x is not greater than 5 or 10.'
```

Output:

```
$ python example09_if_else_elif.py
The variable x is equal to 10.
The variable x is not greater than 20.
The variable x is not greater than 10, but greater than 5.
```

For loop

The for loop facilitates iteration through every element in a set of elements, for example, range, python set, and list.

For loop on range

Input:

```
source_code/appendix_c_python/example10_for_loop_range.py
print "The first 5 positive integers are:"
for i in range(1,6):
    print i
```

Output:

```
$ python example10_for_loop_range.py
The first 5 positive integers are:
1
2
3
4
5
```

For loop on list

Input:

```
source_code/appendix_c_python/example11_for_loop_list.py
primes = [2, 3, 5, 7, 11, 13]
print 'The first', len(primes), 'primes are:'
for prime in primes:
    print prime
```

Output:

```
$ python example11_for_loop_list.py
The first 6 primes are:
2
3
5
7
11
13
```

Break and continue

The `for` loops can be exited earlier using the `break` statement. The remainder of the cycle in the `for` loop can be skipped using the `continue` statement.

Input:

```
source_code/appendix_c_python/example12_break_continue.py
for i in range(0,10):
    if i % 2 == 1: #remainder from the division by 2
        continue
    print 'The number', i, 'is divisible by 2.'

for j in range(20,100):
    print j
    if j > 22:
        break;
```

Output:

```
$ python example12_break_continue.py
The number 0 is divisible by 2.
The number 2 is divisible by 2.
The number 4 is divisible by 2.
The number 6 is divisible by 2.
The number 8 is divisible by 2.
20
21
22
23
```

Functions

Python supports the definition of functions, which is a good way to define a piece of code that can be executed at multiple places in the program. A function is defined using the `def` keyword.

Input:

```
source_code/appendix_c_python/example13_function.py
def rectangle_perimeter(a, b):
    return 2 * (a + b)

print 'Let a rectangle have its sides 2 and 3 units long.'
print 'Then its perimeter is', rectangle_perimeter(2, 3), 'units.'
print 'Let a rectangle have its sides 4 and 5 units long.'
print 'Then its perimeter is', rectangle_perimeter(4, 5), 'units.'
```

Output:

```
$ python example13_function.py
Let a rectangle have its sides 2 and 3 units long.
Then its perimeter is 10 units.
Let a rectangle have its sides 4 and 5 units long.
Then its perimeter is 18 units.
```

Input and output

Let's look at how to pass arguments to a Python program and how to read and write to a file.

Program arguments

We can pass arguments to a program from the command line.

Input:

```
source_code/appendix_c_python/example14_arguments.py
#Import the system library in order to use the argument list.
import sys

print 'The number of the arguments given is', len(sys.argv), 'arguments.'
print 'The argument list is ', sys.argv, '.'
```

Output:

```
$ python example14_arguments.py arg1 110
The number of the arguments given is 3 arguments.
The argument list is ['example14_arguments.py', 'arg1', '110'] .
```

Reading and writing a file

The following program will write two lines to the `test.txt` file, then read them, and finally, print them to the output.

Input:

```
# source_code/appendix_c_python/example15_file.py
#write to the file with the name "test.txt"
file = open("test.txt", "w")
file.write("first line\n")
file.write("second line")
file.close()

#read the file
file = open("test.txt", "r")
print file.read()
```

Output:

```
$ python example15_file.py
first line
second line
```

Statistics

Basic concepts

Notation

A set intersection of two sets, A and B , denoted by $A \cap B$, is a subset of A or B that contains all elements that are in both A and B . In other words, $A \cap B := \{x : x \text{ in } A \text{ and } x \text{ in } B\}$.

A set union of two sets, A and B , denoted by $A \cup B$, is a set that contains precisely the elements that are in A or in B . In other words, $A \cup B := \{x : x \text{ in } A \text{ or } x \text{ in } B\}$.

A set difference of two sets, A and B , denoted by $A - B$ or $A \setminus B$, is a subset of A that contains all elements in A that are not in B . In other words, $A - B := \{x : x \text{ in } A \text{ and } x \text{ not in } B\}$.

The summation symbol, $\sum$, represents the sum of all members over the set, for example:

$$\sum_{i=1}^n a_i = a_1 + a_2 + \cdots + a_n$$

Definitions and terms

- **Population:** A set of similar data or items subject to analysis.
- **Sample:** A subset of the population.
- **Arithmetic mean (average) of a set:** The sum of all the values in the set divided by the size of the set.
- **Median:** The middle value in an ordered set, for example, the median of the set $\{x_1, \dots, x_{2k+1}\}$, where $x_1 < \dots < x_{2k+1}$ is the value x_{k+1} .
- **Random variable:** A function from a set of possible outcomes (for example, heads or tails) to a set of values (for example, 0 for heads and 1 for tails).
- **Expectation:** An expectation of a random variable is the limit of the average values of the increasing sets of the values given by the random variable.
- **Variance:** Measures the dispersion of the population from its mean. Mathematically, the variance of a random variable, X , is the expected value of the square of the difference between the random variable and the mean μ of X , that is, $Var(X) = E[(X - \mu)^2]$.

- **Standard deviation:** The deviation in the random variable, X , is the square root of the variation in the X variable, that is, $SD(X) = \sqrt{Var(X)}$.
- **Correlation:** The measure of the dependency between the random variables. Mathematically, for the random variables X and Y , the correlation is defined as $corr(X, Y) = E[(X - \mu_X) * (Y - \mu_Y)] / (SD(X) * SD(Y))$.
- **Causation:** A dependence relation explaining the occurrence of one phenomenon through the occurrence of another phenomenon. Causation implies correlation, but not vice versa!
- **Slope:** The a variable in the linear equation $y = a * x + b$.
- **Intercept:** The b variable in the linear equation $y = a * x + b$.

Bayesian inference

Let $P(A)$, and $P(B)$ be the probabilities of A and B respectively. Let $P(A|B)$ be the conditional probability of A given B , and let $P(B|A)$ be the probability of B given A .

Then, Bayes' theorem states the following:

$$P(A|B) = (P(B|A) * P(A)) / P(B)$$

Distributions

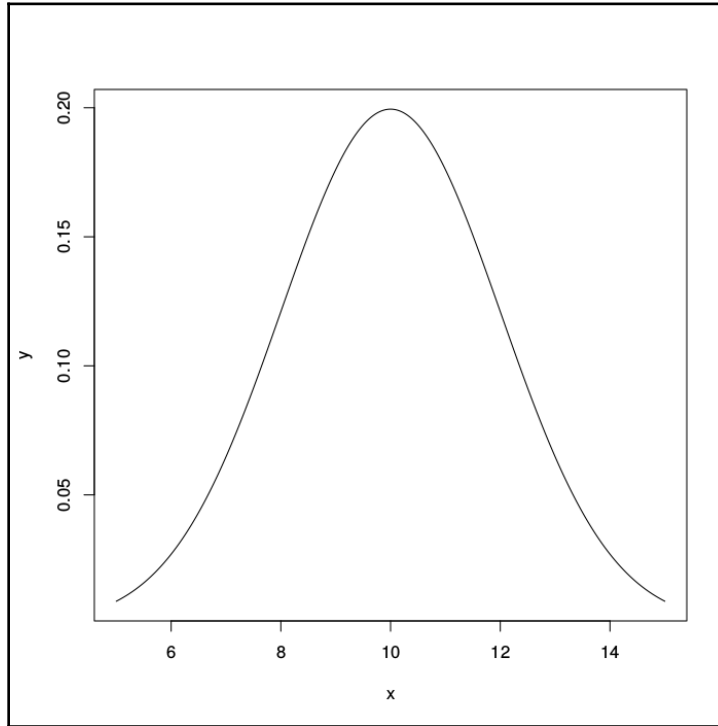
The probability distribution is a function from a set of possible outcomes (for example, heads and tails) to a set of probabilities of those outcomes (that is, 50% for both heads and tails).

Normal distribution

Random variables of many natural phenomena are modeled by a normal distribution. A normal distribution has the following probability density:

$$f(x|\mu, \sigma^2) = \frac{e^{-\frac{(x-\mu)^2}{2\sigma^2}}}{\sqrt{2\sigma^2\pi}}$$

Here, μ is the mean of the distribution and σ^2 is the variation of the distribution. A normal distribution graph has the shape of a bell curve; for example, refer to the following graph of a normal distribution with a mean of 10 and a standard deviation of 2:



Normal distribution with a mean of 10 and a standard deviation of 2

Cross-validation

Cross-validation is a method of validating a hypothesis about data. At the beginning of the analysis process, the data is split into learning data and testing data. A hypothesis is fit to the learning data, and then its actual error is measured on the testing data. This way, we can estimate how well a hypothesis may perform on future data. Reducing the amount of learning data can also be beneficial as it reduces the chance of hypothesis over-fitting. This is where a hypothesis is trained to a particularly narrow subset of the data.

K-fold cross-validation

The original data is partitioned randomly into k folds. One fold is used for validation, while $k-1$ folds of data are used for hypothesis training.

A/B testing

A/B testing is the validation of two hypotheses about data—usually on real data. Then, the hypothesis with the superior outcome (lower estimation error) is chosen as an estimator for future data.

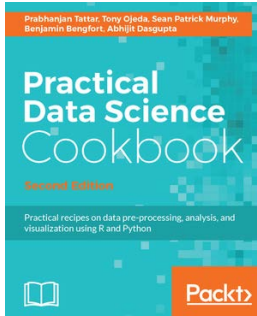
Glossary of Algorithms and Methods in Data Science

- ***k*-nearest neighbors algorithm:** An algorithm that estimates an unknown data item as being like the majority of the *k*-closest neighbors to that item.
- **Naive Bayes classifier:** A way to classify a data item using Bayes' theorem concerning the conditional probabilities $P(A|B) = (P(B|A) * P(A)) / P(B)$. It also assumes that variables in the data are independent, which means that no variable affects the probability of the remaining variables attaining a certain value.
- **Decision tree:** A model classifying a data item into one of the classes at the leaf node, based on matching properties between the branches on the tree and the actual data item.
- **Random decision tree:** A decision tree in which every branch is formed using only a random subset of the available variables during its construction.
- **Random forest:** An ensemble of random decision trees constructed on a random subset of the data with replacement, where a data item is classified to the class with the majority vote from its trees.
- **K-means algorithm:** The clustering algorithm that divides a dataset into *k* groups such that the members in each group are as similar as possible, that is, closest to one another.
- **Regression analysis:** A method for estimating the unknown parameters in a functional model that predicts the output variable from the input variables, for example, to estimate *a* and *b* in the linear model $y = a * x + b$.
- **Time series analysis:** The analysis of data dependent on time; it mainly includes the analysis of trends and seasonality.
- **Support vector machines:** A classification algorithm that finds the hyperplane dividing the training data into given classes. This division by the hyperplane is then used to classify the data further.
- **Principal component analysis:** The preprocessing of individual components of given data in order to achieve better accuracy, for example, rescaling of the variables in the input vector depending on how much impact they have on the end result.

- **Text mining:** The search and extraction of text, and its possible conversion to numerical data that is used for data analysis.
- **Neural networks:** A machine learning algorithm consisting of a network of simple classifiers that make decisions based on the input or the results of the other classifiers in the network.
- **Deep learning:** The ability of a neural network to improve its learning process.
- **A priori association rules:** The rules that can be observed in training data and, on the basis of which, a classification of the future data can be made.
- **PageRank:** A search algorithm that assigns the greatest relevance to the search result that has the greatest number of incoming web links from the most relevant search results for a given search term. In mathematical terms, PageRank calculates a certain eigenvector representing these measures of relevance.
- **Ensemble learning:** A method of learning where different learning algorithms are used to reach a final conclusion.
- **Bagging:** A method of classifying a data item by the majority vote of the classifiers trained on random subsets of the training data.
- **Genetic algorithms:** Machine learning algorithms inspired by genetic processes, for example, an evolution where classifiers with the greatest accuracy are trained further.
- **Inductive inference:** A machine learning method for learning the rules that produced the actual data.
- **Bayesian networks:** A graph model representing random variables with their conditional dependencies.
- **Singular value decomposition:** The factorization of a matrix, a generalization of eigendecomposition, used in the least squares methods.
- **Boosting:** A machine learning meta-algorithm that decreases the variance in an estimation by making a prediction based on the ensembles of the classifiers.
- **Expectation maximization:** An iterative method for searching the parameters in the model that maximize the accuracy of the prediction of the model.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

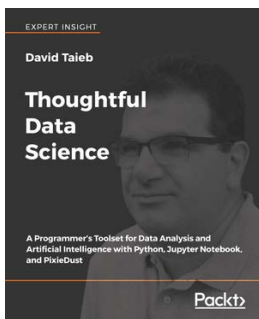


Practical Data Science Cookbook - Second Edition

Prabhanjan Tattar et al.

ISBN: 978-1-78712-962-7

- Learn and understand the installation procedure and environment required for R and Python on various platforms
- Prepare data for analysis by implement various data science concepts such as acquisition, cleaning and munging through R and Python
- Build a predictive model and an exploratory model
- Analyze the results of your model and create reports on the acquired data
- Build various tree-based methods and Build random forest



Thoughtful Data Science

David Taieb

ISBN: 978-1-78883-996-9

- Bridge the gap between developer and data scientist with a Python-based toolset
- Get the most out of Jupyter Notebooks with new productivity-enhancing tools
- Explore and visualize data using Jupyter Notebooks and PixieDust
- Work with and assess the impact of artificial intelligence in data science
- Work with TensorFlow, graphs, natural language processing, and time series
- Deep dive into multiple industry data science use cases
- Look into the future of data analysis and where to develop your skills

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

B

- ballistic flight analysis
 - about 152, 153
 - least squares method, using 154
- Bayes' theorem
 - about 34
 - analysis 33
 - basic application, building 33
 - extension 34
 - proof 34
 - using, for continuous random variables 44
- bootstrap aggregating 80
- business profits
 - prediction, by analyzing trends 161
 - trend line equation 163
 - trend, analyzing with least squares method 161
 - visualizing 162

C

- chess game
 - analysis 37, 42
 - analyzing, with decision tree 70, 72, 73
 - dependent events 41
- clusters k
 - counts, in semantic context 124, 125, 129, 131
- clusters
 - selecting 117, 118, 121, 122

D

- data inconsistency
 - confidence level, measuring 99
 - dealing with 73, 74
 - overcoming, with randomness 99
- data
 - rescaling 20, 21
 - using, as decision tree 56, 57

E

- electronics shop's sales
 - seasonality, analyzing 163, 164, 167, 170, 171
 - trend, analyzing with least squares method 165
 - visualizing 167
- extended Bayes' theorem
 - about 35
 - proof 35

F

- Fahrenheit and Celsius conversion
 - analysis 141
 - linear regression 141
 - visualization 144
- flight time duration prediction
 - analysis 151
 - based, on distance 151

G

- gender classification example
 - about 110
 - analysis 111, 112
 - input data 116
 - program output 116
- gradient descent algorithm
 - about 146
 - implementation 149
 - visualization 150

H

- height – linear regression
 - weight prediction 145

I

- ID3 algorithm

- about 61
- used, for classification 68
- used, for constructing decision tree 61

information entropy

- defining 58

information theory

- about 57
- gaining process 59

Italy map example

- analysis 16, 19
- k value, selecting 15

K

k clusters

- clustering into 106

k-means clustering algorithm

- about 107
- applying, on household income example 108, 109
- centroid, computing 108
- implementing 113
- initial k-centroids, selecting 108

k-nearest neighbors algorithm

- implementing 10, 13
- used, for text classification 25

L

least squares method, Python

- used, for analysis 143

linear regression

- about 141
- least squares method 142

N

Naive Bayes classifier

- implementation 38, 41

non-Euclidean distances

- used, for text classification 22

R

random forest algorithm

- about 80
- analysis 81
- chess example, playing 91
- classifying 98
- constructed random forest 87
- constructing 82, 92
- implementation 88, 91
- overview 80
- random decision tree number 0, constructing 82
- random decision tree number 1, constructing 84
- used, for performing analysis 100
- used, for swim preference analysis 81

S

swimming preference decision tree

- about 62
- analysis, with random forest algorithm 81
- implementation 63, 68
- information gain calculation 60
- used, for data sample classification 69

T

temperature preferences 6, 10

text classification

- k-NN, using 25, 26
- non-Euclidean distances, using 22, 24

time-based prediction

- seasonality 160
- trends 160

tree bagging 80