Brian Steele · John Chandler
Swarna Reddy

Algorithms for Data Science

Springer

Algorithms for Data Science

Brian Steele • John Chandler • Swarna Reddy

Algorithms for Data Science

Springer

Brian Steele
University of Montana
Missoula, MT, USA

Swarna Reddy
SoftMath Consultants, LLC
Missoula, MT, USA

John Chandler
School of Business Administration
University of Montana
Missoula, MT, USA

ISBN 978-3-319-45795-6 ISBN 978-3-319-45797-0 (eBook)
DOI 10.1007/978-3-319-45797-0

Library of Congress Control Number: 2016952812

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

Data science has been recognized as a science since 2001, roughly. Its origin lies in technological advances that are generating nearly inconceivable volumes of data. The rate at which new data are being produced is not likely to slow for some time. As a society, we have realized that these data provide opportunities to learn about the systems and processes generating the data. But data in its original form is of relatively little value. Paradoxically, the more of it that there is, the less the value. It has to be reduced to extract value from it. Extracting information from data is the subject of data science.

Becoming a successful practitioner of data science is a real challenge. The knowledge base incorporates demanding topics from statistics, computer science, and mathematics. On top of that, domain-specific knowledge, if not critical, is very helpful. Preparing students in these three or four areas is necessary. But at some point, the subject areas need to be brought together as a coherent package in what we consider to be a course in *data science*. A student that lacks a course that actually teaches data science is not well prepared to practice data science. This book serves as a backbone for a course that brings together the main subject areas.

We've paid attention to the needs of employers with respect to entry-level data scientists—and what they say is lacking from the skills of these new data scientists. What is most lacking are programming abilities. From the educators' point of view, we want to teach principles and theory—the stuff that's needed by students to learn on their own. We're not going to be able to teach them everything they need in their careers, or even in the short term. But teaching principles and foundations is the best preparation for independent learning. Fortuitously, there is a subject that encompasses both principles and programming—algorithms. Therefore, this book has been written about the algorithms of data science.

Algorithms for Data Science focuses on the principles of data reduction and core algorithms for analyzing the data of data science. Understanding the fundamentals is crucial to be able to adapt existing algorithms and create new algorithms. The text provides many opportunities for the reader to develop and improve their programming skills. Every algorithm discussed at length is accompanied by a tutorial that guides the reader through implementation of the algorithm in either `Python` or `R`. The algorithm is then applied to a real-world data set. Using real data allows us to talk about domain-specific problems. Regrettably, our self-imposed coding edict eliminates some important predictive analytic algorithms because of their complexity.

We have two audiences in mind. One audience is practitioners of data science and the allied areas of statistics, mathematics, and computer science. This audience would read the book if they have an interest in improving their analytical skills, perhaps with the objective of working as a data scientist. The second audience are upper-division undergraduate and graduate students in data science, business analytics, mathematics, statistics, and computer science. This audience would be engaged in a course on data analytics or self-study.

Depending on the sophistication of the audience, the book may be used for a one- or two-semester course on data analytics. If used for a one-semester course, the instructor has several options regarding the course content. All options begin with Chaps. 1 and 2 so that the concepts of data reduction and data dictionaries are firmly established.

1. If the instructional emphasis is on computation, then Chaps. 3 and 4 on methods for massively large data and distributed computing would be covered. Chapter 12 works with streaming data, and so this chapter is a nice choice to close the course. Chapter 7 on healthcare analytics is optional and might be covered as time allows. The tutorials of Chap. 7 involve relatively large and challenging data sets. These data sets provide the student and instructor with many opportunities for interesting projects.
2. A course oriented toward general analytical methods might pass over Chaps. 3 and 4 in favor of data visualization (Chap. 5) and linear regression (Chap. 6). The course could close with Chap. 9 on k-nearest neighbor prediction functions and Chap. 11 on forecasting.
3. A course oriented toward predictive analytics would focus on Chaps. 9 and 10 on k-nearest neighbor and naïve Bayes prediction functions. The course would close with Chaps. 11 and 12 on forecasting and streaming data.

Acknowledgments

We thank Brett Kassner, Jason Kolberg, and Greg St. George for reviewing chapters and Guy Shepard for help solving hardware problems and unraveling network mysteries. Many thanks to Alex Philp for anticipating the future and breaking trail. We thank Leonid Kalachev and Peter Golubstov for many interesting conversations and insights.

Missoula, MT, USA Brian Steele
Missoula, MT, USA John Chandler
Missoula, MT, USA Swarna Reddy

Contents

Part I Data Reduction

Part II Extracting Information from Data

Part III Predictive Analytics

List of Figures

List of Tables

Biographical Sketches

Brian Steele

Brian Steele is a full professor of Mathematics at the University of Montana and a Senior Data Scientist for SoftMath Consultants, LLC. Dr. Steele has published on the EM algorithm, exact bagging, the bootstrap, and numerous statistical applications. He teaches data analytics and statistics and consults on a wide variety of subjects related to data science and statistics.

John Chandler

John has worked at the forefront of marketing and data analysis since 1999. He has worked with Fortune 100 advertisers and scores of agencies, measuring the effectiveness of advertising and improving performance. Dr. Chandler joined the faculty at the University of Montana School of Business Administration as a Clinical Professor of Marketing in 2015 and teaches classes in advanced marketing analytics and data science. He is one of the founders and Chief Data Scientist for Ars Quanta, a Seattle-based data science consultancy.

Swarna Reddy

Dr. Swarna Reddy is the founder, CEO, and a Senior Data Scientist for SoftMath Consultants, LLC and serves as a faculty affiliate with the Department of Mathematical Sciences at the University of Montana. Her area of expertise is computational mathematics and operations research. She is a published researcher and has developed computational solutions across a wide variety of areas—spanning bioinformatics, cybersecurity, and business analytics.

Chapter 1
Introduction

Abstract The beginning of the twenty-first century will be remembered for dramatic and rapid technological advances in automation, instrumentation, and the internet. One consequence of these technological developments is the appearance of massively large data sets and data streams. The potential exists for extracting new information and insights from these data. But new ideas and methods are needed to meet the substantial challenges posed by the data. In response, data science has formed from statistics and computer science. Algorithms play a vastly important and uniting role in data analytics and in the narrative of this book. In this chapter, we expand on these topics and provide examples from healthcare, history, and business analytics. We conclude with a short discussion of algorithms, remarks on programming languages, and a brief review of matrix algebra.

1.1 What Is Data Science?

Data science is a new field of study, just sufficiently coalesced to be called a science. But the field is real and data science is truly a hybrid discipline, sitting at the intersection of statistics and computer science. Often, a third field, a domain in the language of data science, is intimately connected.

What then is data science? Data science is an amalgam of analytic methods aimed at extracting information from data. This description also fits statistics and data mining, but data science is neither. To better understand what data science is, we start with the genesis. Technological advances are driving the formation of massively large data sets and streaming data. Two broad technologies are most responsible—the internet and automated data collection devices. Devices that collect data have become nearly ubiquitous and often surreptitious in the human environment. For example, smart phones, internet websites, and automatic license plate readers are collecting data nearly all

B. Steele et al., *Algorithms for Data Science*,
DOI 10.1007/978-3-319-45797-0_1

of the time, and often for obscure reasons. The smart phone in your pocket is measuring the ambient temperature and your latitude and longitude. The phone's gravitometer is measuring the local gravitational field. Observations on these three variables are collected at the approximate rate of 100 times per second. Not all of these data collection processes are surreptitious in nature though. State, local, and national governments are releasing volumes of data to the public in the interest of openness and improving the well-being of the citizenry. Non-profit organizations such as DataKind (http://datakind.org) and Data4America (https://data4america.org/) are, theoretically, working towards the betterment of society through the analyses of these data.

In the past century, data collection often was carried out by statistical sampling since data was manually collected and therefore expensive. The prudent researcher would design the process to maximize the information that could be extracted from the data. In this century, data are arriving by firehose and without design. The data exist for tangential reasons relative to the analyst's objectives. Because the data are generated without benefit of design, the information content of each datum is often extremely small with respect to answering a sensible question. Sometimes it's difficult to even ask a sensible question (what would you do with trillion observations on gravity?) The challenges posed by what is often called *big data* stem from the prevailing diluteness of information content, volume, and the dearth of design and control. Statistical science is unsuited for the challenges posed by these data. Data science has developed in response to the opportunity and need for creating value from these data.

A science is an organized body of knowledge united by a common subject. Organization within the science arises from foundations and principles. Organization reveals linkages and similarities and turns a miscellany of facts into understanding. And so here is the challenge: if there are foundations and principles of data science, what are they? It's not entirely obvious. But, there is a preponderant theme, a raison d'être for data science from which the foundations are forming. The theme is the extraction of information from data.

Statistical science overlaps with data science with respect to the objective of extracting information from data but data science spans situations that are beyond the scope of statistics. In these situations, the statistical aspects of the problem pale in comparison with the importance of algorithmic and computational considerations. Since statistical science revolves around the analysis of relatively small and information-rich data, statistical philosophy and principles are at times marginally relevant to the task at hand. Furthermore, the validity of many statistical methods is nullified if the data are not collected by a probabilistic sampling design such as random sampling. Hypothesis testing is nearly irrelevant in data science because opportunistically collected data lack the necessary design. When the data are appropriate and hypotheses exist, massively large data sets tend to yield highly significant results (p-values) no matter how small the practical differences are. If hypothesis tests produce the same outcome with near certainty, there's no point in carrying out the tests. One of the skills of an accomplished data scientist is the ability to judge what statistical techniques are useful and figure out how apply them at large scale. Both programming techniques and statistical methods are necessary and ubiquitous in data science. Data science is not statistics and this is not a book about statistics.

The data of data science may be loosely described as being of three varieties: massive in volume but static, arriving in a stream and needing immediate analysis, and high dimensional. The problem of high dimensionality differs in several respects from the first two varieties. Most importantly, data reduction techniques require complex mathematical and computational methods that are incompatible with the do-it-yourself philosophy of this book. The philosophy will be discussed in some detail soon, but for now, the philosophy dictates that every algorithm and method is to be programmed by the reader. That dictum excludes many dimension reduction techniques, for example, principal components analysis and its parent, the singular value decomposition, and the lasso method of variable selection in regression. We focus instead on massively large static data and streaming data. Let's look at a few examples.

1.2 Diabetes in America

The U.S. Centers for Disease Control and Prevention (CDC) is dedicated to learning about and understanding the factors that affect the health and well-being of the U.S. population. To this end, the CDC has been conducting the largest annual sample survey in the world. The Behavioral Risk Factor Surveillance System (BRFSS) survey [10] asks the participants a number of questions regarding health and health-related behaviors. Of the many questions asked of the participants, one asks whether the respondent has diabetes, a chronic, incurable, and often preventable disease. Diabetes is an enormous public health problem as it probably affects more than 9% of U.S. adults. The cost of treating diabetes over a lifetime is estimated to be as much as $130,800 per patient [71] and the consequences of not treating the disease are dire. Physicians, the Centers for Medicare & Medicaid Services, and private insurers would like to know who is most likely to develop the disease so that those at risk may be encouraged to participate in prevention programs. But who is most at risk? More to the point, is it possible to build an algorithm that estimates risk based on a few simple variables?

To investigate the question, we used $n = 5{,}195{,}986$ responses from the BRFSS surveys and reduced these individual records according to the respondents' age, level of education, annual household income, and body mass index. Mathematically, we mapped each individual to one of 14,270 profiles, each of which is a specific combination of the four variables. In plain language, we dumped individuals into bins and computed sample proportions. There's a little more to it than just dumping observations into bins, but that's the essence of the algorithm. The tutorials of Chap. 7 guide the reader through the complete process.

Table 1.1 shows the profiles with the largest estimated diabetes prevalence.[1] All of the estimates exceed .625. The profiles shown in Table 1.1 are easy to describe. The individuals are old, very poor, and with body weights much exceeding ideal weight. The individuals tend to be poorly educated though there is more variation in this variable than the others. An individual with a profile listed in Table 1.1 and not yet having the disease is a prime candidate for a prevention program.

Table 1.1 A few profiles and estimated diabetes prevalence. Data from the Centers for Disease Control and Prevention, Behavioral Risk Factor Surveillance System surveys years 2001 through 2014, inclusive

Category				Number of	Estimated
Income[a]	Education[b]	Age[c]	Body mass index[d]	respondents	prevalence
1	3	8	50	110	.627
1	2	8	50	202	.629
2	2	12	50	116	.629
2	6	10	46	159	.635
1	4	10	52	129	.636
3	2	10	44	177	.638
2	3	11	50	123	.642

[a] Income category 1 corresponds to an annual household income of less than $10,000, category 2 is $10,000–15,000, and category 3 is $15,000–20,000

[b] There are six education categories, labeled 1 through 6 and ordered from no education (1) to 4 years or more of college (6)

[c] Age category 8 corresponds to 55–59 years, 9 corresponds to 60–64 years, and so on. Category 12 is 75–79 years

[d] A body mass index of 30 kg/m^2 or more corresponds to clinically obese and a body mass index of 40 roughly equates to a weight approximately 100 pounds greater than ideal weight

Conceptually, the analysis was a simple exercise in data reduction. Without the massive data sets, this approach would not have worked. The sample sizes for individual profiles would be small and the estimates of prevalence too imprecise. The prudent analyst would have undertaken an effort to find a statistical model from which to compute prevalence estimates. But, with the CDC data, the average number of observations per profile is 382.8, a large average by most standards. We don't need a model. The argument will be made in Chap. 7 that the profile solution is better than a modeling solution. It's not necessarily easier though. The gain in statistical simplicity and accuracy requires a considerable amount of data processing, a process sometime referred to as data munging.

[1] Diabetes prevalence is the rate of diabetes.

1.3 Authors of the Federalist Papers

The Federalist Papers are a collection of 85 essays written by James Madison, Alexander Hamilton, and John Jay arguing for the ratification of the United States Constitution. The Federalist Papers have been an irreplaceable asset for historians and the judiciary as they expose the intent of the Constitution and its authors. At the time of publication (1787 and 1788), all were signed with a pseudonym. The authors of most of the papers were revealed in the writings of Hamilton after his death at the hand of Aaron Burr. The authorship of 12 papers claimed by Hamilton were disputed for almost 200 years. Some 50 years ago, analyses by historians and statisticians [1, 41] attributed authorship of all 12 to James Madison.

Mosteller and Wallace [41] conducted an assortment of statistical analyses. In particular, they contrasted the relative frequency of occurrence of certain words among the authors' papers. The number of words that they used was small and they omitted the great majority of words used by all three authors. They argued that for many words, differences in frequency of use would be attributable to differences in subject matter. In Chap. 10, we attempt to attribute authorship to the disputed papers. The idea motivating our analysis is that the similarity of a disputed paper to the papers of Hamilton, Madison, and Jay may be determined by comparing the word use distributions, paper to author. Pervasively occurring prepositions and conjunctions such as *the* and *in* have been excluded from the analysis, but any other word used by all three is included. We use a much larger set of words than Mosteller and Wallace, 1102 in number. Figure 1.1 shows the relative frequency of use for the twenty most commonly used words of Hamilton. A few words are used almost exclusively by one author, for instance, *upon*. A few words—*national*, for example—are used at nearly the same rate by the three authors.

The prediction function predicts the author of a disputed paper to be the author with the word distribution most similar to the paper. Mathematically, we built a function $f(\cdot|D)$ from $D = \{P_1, \ldots, P_n\}$, the set of undisputed, sole-author papers. The function is perhaps best described as an algorithm as it consists a composition of functions. When passed the digital text of a disputed paper, the paper is transformed to a vector of word counts $\mathbf{x}_0$. A second function compares $\mathbf{x}_0$ to each authors' word use distribution. The comparison produces an estimate of the relative likelihood that Hamilton wrote the paper, and similarly for Madison and Jay. Lastly, the largest relative likelihood determines the predicted author. For example, our prediction of the disputed Paper 53 is $f(P_{53}|D) =$ Hamilton.

The accuracy of the prediction function is good. All of the undisputed papers are correctly assigned to their author. Of the disputed authors, we find 6 assigned to Madison and 6 to Hamilton. Adair, a historian, and Mosteller and Wallace assigned all 12 to Madison. It should be noted that the effort put into research and analysis by Adair and Mosteller and Wallace was remarkably thorough even by the standards of the day. We've done little more

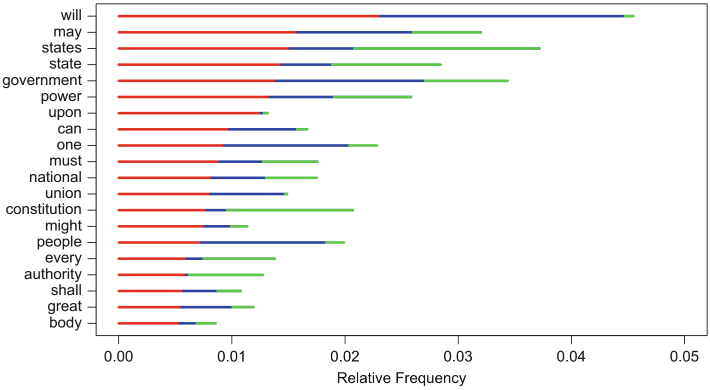


Fig. 1.1 Relative frequency of occurrence of the most common 20 words in Hamilton's undisputed papers. Hamilton's use is shown in *red*. The *blue* and *green lines* correspond to Madison and Jay, respectively

than reduce the textual data to word use distributions and implement a fairly simple multinomial naïve Bayes prediction function in `Python`.

1.4 Forecasting NASDAQ Stock Prices

The two examples discussed so far involve large data sets of two varieties: quantitative and ordinal measurements from the BRFSS database and textual data from the Federalist Papers. Another variety of data are streaming data. A data stream may be created by sending requests to the Yahoo Financial Application Programming Interface (API) for the asking price of one or more stocks listed on the NASDAQ stock exchange. Prices posted to the Yahoo API are delayed by some 15 min, so we can't carry out forecasting in real-time, but it's pretty close. We send requests once a second and receive perhaps five updates on the asking price per minute depending on the stock's trading activity. The objective is to forecast the price of a stock $\tau > 0$ time steps in the future. A time step is the interval between updates.

The NASDAQ self-generated stream serves as a model for higher-velocity streams. For instance, a commercial website's internet traffic needs to be defended against malicious activity. Packet analysis is one defense. Packets are the transmission units of the internet. A packet is structured like a letter—there are origin and destination addresses in the header and a payload (the data) contained within. The content is small in volume, typically less than 1000 bytes, and so vast numbers of packets are transferred in normal internet activity. One defense against malicious activity inspects the headers for anomalies in origin, packet size, and so on. Real-time analysis of packets necessitates an inspect-and-dismiss strategy as it is impossible and pointless to store the data for later analysis. Whatever is to be done must be done in real-time.

Returning to the stock price forecasting problem, there are practical constraints on a real-time forecasting algorithm. Upon the arrival of a datum, the algorithm should update the forecasting function to incorporate the information contained within the datum. The updating formula must execute rapidly and so must be simple. The amount of past data that is stored must be small in volume. Above all, the forecast should be accurate.

To illustrate, consider a linear regression forecasting function that uses time step as the predictor variable. Suppose that the current time step is n and τ is a positive integer. The target value to be forecasted is $y_{n+\tau}$ and the forecast is

$$\widehat{y}_{n+\tau} = \widehat{\beta}_{n,0} + (n+\tau)\widehat{\beta}_{n,1}$$

where $\widehat{\beta}_{n,0}$ is the intercept and the estimated price level at the start $(n = 0)$. The coefficient $\widehat{\beta}_{n,1}$ is the slope at time n and the estimated change in future price from one time step to the next. Unlike the usual linear regression situation, the coefficients are not unknown constants. They're unknown, but they vary in time to reflect changes in the price stream.

Whenever y_n arrives, the coefficient vector $\widehat{\boldsymbol{\beta}}_n = [\widehat{\beta}_{n,0} \;\; \widehat{\beta}_{n,1}]^T$ is computed. Then, a forecast $\widehat{y}_{n+\tau}$ is computed. The mathematical details are delayed until Chap. 11, but in brief, a solution to the real-time forecasting problem exploits the fact that $\widehat{\boldsymbol{\beta}}_n$ can be computed according to $\widehat{\boldsymbol{\beta}}_n = \mathbf{A}_n^{-1}\mathbf{z}_n$ where $\mathbf{A}_n$ is a matrix and $\mathbf{z}_n$ is a vector. The terms $\mathbf{A}_n$ and $\mathbf{z}_n$ encapsulate all of the past information. Upon the arrival of y_{n+1}, $\mathbf{A}_n$ and $\mathbf{z}_n$ are updated by incorporating y_{n+1}. Updating $\widehat{\boldsymbol{\beta}}_n$ and computing the forecast $\widehat{y}_{n+\tau}$ are carried out extremely rapidly.

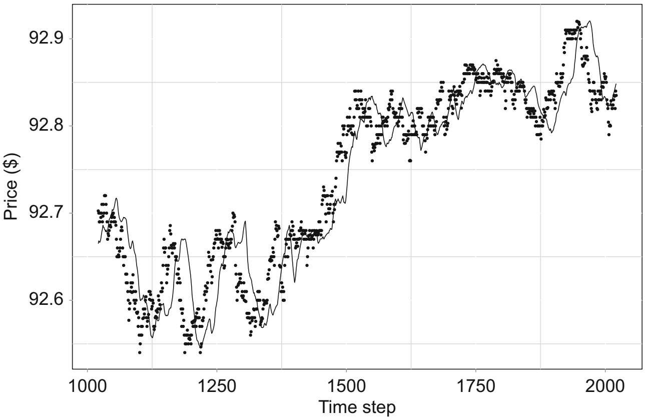


Fig. 1.2 Observed prices (points) and time-varying linear regression forecasts of Apple, Inc. (line) for $\tau = 20$

Figure 1.2 shows a short portion of the data stream and the forecasts for $\tau = 20$ time steps ahead. Two features are most notable. First, the function captures the trend. Secondly, the forecasts $\widehat{y}_{n+\tau}$ reflect what has happened at time steps $n, n-1, n-2, \ldots$. To compare actual and forecasted values, choose a time step and compare the actual and forecasted value *at that time step.* The forecasts reflect the recent past. Forecasting functions are not crystal-balls—the data cannot predict a change of direction in stock prices before it happens. They only perform well if the data that is used to build them resembles the future. The best we can do is give greater weight to the most recent observations at the expense of older observations.

1.5 Remarks

The three examples are very different with respect to the origins and form of the data and objectives of the analyses. However, the data analytics applied to the data are alike in purpose: the reduction of data to information relevant to the objectives. The profile analysis example reduced approximately 5.2 million observations to about 14,000 profiles. Each profile describes the demographic characteristics of a cohort of the U.S. adult population in terms of age, education, income, body mass index, and diabetes risk. The Federalist papers example reduced the undisputed papers to three word use distributions, one for each author. The thoughts and ideas expressed in each paper are lost. Our aim was not to encapsulate that information but rather to develop a function for predicting the authorship of a disputed paper. For this purpose, the distributions are very well-suited. The NASDAQ example illustrates the reduction of a data stream to a matrix $\mathbf{A}_n$ and a vector $\mathbf{z}_n$. Regardless of how many time steps have passed, the information relevant for forecasting is encapsulated in the two statistics.

1.6 The Book

This textbook is about practical data analytics written with the objective of uniting the principles, algorithms, and data of data science. Algorithms are the machinery behind the analytics and the focal point of the book. To be good at data analytics, one must be proficient at programming. This is new. Thirty years ago it was possible to be a practicing, applied statistician without the ability to write code. (And, in fact, we've bemusedly listened to colleagues brag about not writing code as recently as 2001.) To be good, one also must have some experience with the data and the algorithms of data science. To be expert at data analytics requires an understanding of the foundations and principles behind the algorithms. Why? Applying the algorithms to real problems often requires adapting existing algorithms and creating new ones. Almost nothing works as expected.

To get really good results, you'll have to tinker with the algorithms, try different things—in other words, innovate. But without an understanding of the foundations, there will be lots of frustration, dead-ends, and wasted time and precious little success. To get the reader to the point where innovation is not intimidating but instead an opportunity for creativity, the text presents a set of prototypical algorithms that are representative of data analytics. For most of us, reading about the algorithms is not enough. To learn how to innovate in a relatively short time, the reader must be actively engaged in turning the algorithms into code and using them with real data. By doing so, misconceptions and misunderstandings are confronted and remedied. Every one of these prototypical algorithms is the subject of a tutorial, usually in `Python`, but sometimes `R` in which the reader renders the algorithm as code and applies it to data.

There's a debate among the data science cognoscenti about the importance of domain expertise. Domain expertise is knowledge of the field in which the analytics are being applied. Examples of these fields are marketing, real estate, financial trading, credit-risk analysis, healthcare analytics, and social networks. Some argue that domain expertise is most important among the three areas of statistics, computer science, and domain expertise. This is probably an unproductive argument. The key point is that data scientists *must* collaborate with partners who collect the data and those who care about the answer to their questions. Regardless, our opinion is that domain expertise can be learned on the job or, for small projects, in conversations with experts. On the other hand, foundations cannot be learned in *ad hoc* fashion and learning algorithms on the job is slow and arduous. To expose the reader to domain-specific applications, Chap. 7 looks into healthcare analytics. Our tutorials work with real data and provide background information but do not go far into domain-specific details.

The book is intended for two audiences. One audience is practitioners of data science and the allied areas of statistics, mathematics, and computer science. This audience would read the book if they have an interest in improving their analytical skills, perhaps with the objective of transitioning from their narrower domain to data science. The second audience are upper-division undergraduate and graduate students in data science, business analytics, mathematics, statistics, and computer science. This audience would be engaged in a course on data analytics or self-study. There are aspects of this book that will be a technical challenge for this audience. We urge patience on the part of this audience—all skills are learned on a continuum and the ability to derive certain results from first principles is not a requirement that applies to these algorithms.

The prerequisites necessary to work comfortably with the book are kept low because of the widely differing backgrounds of the audience. The reader with one or two courses in probability or statistics, an exposure to vectors and matrices, and a programming course should succeed at mastering most of the content. The core material of every chapter is accessible to all with these prerequisites. The chapters sometimes expand at the close with innovations of interest to practitioners of data science.

The orientation of this book is what might be called *close to the metal* in the sense that there is very little reliance on existing data analytic algorithms. This approach is adopted for two reasons: to immerse the student in `Python` so that she may become a proficient programmer, and to develop a deep understanding of the fundamental, work-horse algorithms and methods of data science. The trade-off is that some of the more sophisticated algorithms in predictive analytics (e.g., neural nets and support vector machines) are beyond the scope of the course. There's lots about predictive analytics, data mining, programming, and statistics that are not touched on in this book. We offer no apology.

The book is divided into three parts:

I. **Data Reduction**: Herein, the dual foundations of data reduction and scalability are developed. Chapter 2 focuses on data reduction via data maps and the use of data dictionaries. The data for the tutorials come from the Federal Election Commission's compilation of monetary contributions to candidates and political action committees. Chapter 3 introduces associative statistics, the mathematical foundation of scalable algorithms and distributed computing. Another open government data source is used for the tutorials of Chap. 3, the Centers for Disease and Control's Behavioral Risk Surveillance System surveys. Practical aspects of distributed computing are the subject of Chap. 4 on Hadoop and MapReduce.

II. **Extracting Information from Data**: The focus shifts now to methods and the algorithms that support the methods. The topics are linear regression, data visualization, and cluster analysis. The chapters on linear regression and data visualization focus on methodological aspects of the analyses, and so depart from the algorithmic theme of the book. However, these methods are used routinely in practical data analytics, and a practicing data scientist must have a solid knowledge of the capabilities and constraints associated with linear regression and a working knowledge of data visualization. Chapter 7 dives into a growing domain—Healthcare Analytics—for an extended example of developing algorithms for data analysis.

III. **Predictive Analytics**: Part III introduces the reader to predictive analytics by developing two foundational and widely used prediction functions, k-nearest neighbors and naïve Bayes. Chapter 10, on the multinomial naïve Bayes prediction function, provides the reader with the opportunity to work with textual data. Forecasting is the subject of Chap. 11. Streaming data and real-time analytics are presented in Chap. 12. The tutorials use publicly accessible data streams originating from the Twitter API and the NASDAQ stock market.

This book does not cover data science *methods* exhaustively. An algorithmic treatment of methods is too much for a single text. Data science is too broad an area. For the reader who wants a deeper understanding in the methods of data science, we suggest further study in several areas: predictive analytics (or statistical learning), data mining, and traditional statistics. On the subjects of data mining and predictive analytics we recommend the texts by James et al., *An Introduction to Statistical Learning* [29], Aggarwal's *Data Mining* [2] and Witten et al.'s *Data Mining: Practical Machine Learning Tools and Techniques* [70]. Harrell's *Regression Modeling Strategies* [25] and Ramsey and Schafer's *The Statistical Sleuth* [48] are excellent statistics books. One should consider Wickham's [65] treatment of data visualization and the `R` package `ggplot2`. Provost and Fawcett's *Data Science for Business* [45] and Grus's *Data Science from Scratch* [23] provide complementary overviews of data science without the burden of algorithmic details. Janssens's *Data Science at the Command Line* [30] pursues a unique treatment of data science that will appeal to computer scientists.

1.7 Algorithms

An algorithm is a function. More to the point, an algorithm is a series of functions that progressively transform an input before yielding the output. Our focus is on algorithms that process data for the purpose of extracting information from data. We said earlier that algorithms are the connective tissue of data science, a metaphorical statement that deserves explanation. What is meant by that statement is that the principles are applied to the data through the algorithms.

The most important attributes of algorithms in general are correctness, efficiency, and simplicity [56]. When the algorithm's purpose is data reduction, then another criterion supersedes the second and third attributes—minimal information loss. Information loss is an anathema since the overriding objective is to extract information from data. As a result, the concepts of associative statistics and scalable algorithms are invaluable when working with massively large data sets. Building data structures that organize and preserve information also are very important, and we promote the use of dictionaries, or hash tables for data reduction. These topics are developed in detail in Chaps. 2 and 3.

The process of data reduction often involves aggregating observations according to one or more attributes with organizational meaning. In the Federalist Paper example, the organizational unit was the author, and so we ended up with three word distributions (one each for Hamilton, Madison, and Jay). In the diabetes example, there were 14,270 profiles, each consisting of a unique combination of age, education, income and body mass index. In the NASDAQ example, the data were aggregated in the form of a matrix $\mathbf{A}_n$ and a vector $\mathbf{z}_n$.

Though the diabetes and Federalist papers examples are very different, the data structure for collating and storing the reduced data is the same: a *dictionary*. A dictionary consists of a set of items. Each item has two components.

The *key* is the unit of organization. In a conventional dictionary, the keys are words. In the Federalist papers example, the keys are authors, and in the diabetes example, profiles are the keys. The second component is the *value*. In a conventional dictionary, the value is the definition of the word. In the Federalist papers example, the word distribution is the value and in the diabetes example, the estimated risk of diabetes is the value. Dictionaries are invaluable for working with data because of the ease with which data may be organized, stored, and retrieved. What of the algorithm? The algorithm builds the dictionary and so the design of the algorithm is dictated by the dictionary. The dictionary structure is dictated by the principles of data reduction and the objectives of the analysis.

The NASDAQ data stream is an example without a dictionary. The algorithm again is designed around the desired result: a price forecast that is updated upon the arrival of a new datum. The forecasted price is to be the output of a function driven by recently observed data. The influence of early observations diminishes as time advances. We use a simple variant on the usual sample mean to accomplish this objective—the exponentially weighted average. Since there's no need to store the data, there's no need for a dictionary.

1.8 Python

Python is the language of data science. While many languages are used in data science, for instance, C++, Java, Julia, R, and MATLAB, Python is dominant. It's easy-to-use, powerful, and fast. Python is open-source and free. The official Python home https://www.python.org/ has instructions for installation and a very nice beginners guide. It's best for most people to use Python in a development environment. We use jupyter (http://jupyter.org/) and Sypder3 (https://pythonhosted.org/spyder/). The shockingly comprehensive Anaconda distribution (https://www.continuum.io/why-anaconda) contains both. This book assumes that the reader is using version 3.3 of Python. We caution the reader that some 3.3 instructions used in the tutorials will not work with Python 2.7. It's rarely difficult to find a solution to version problems by searching the web.

A virtue of Python is that there is a large community of experienced programmers that have posted solutions on the internet to common Python coding problems. If you don't know how to code an operation (e.g., reading a file), then a internet search usually will produce useful instructions and code. Answers posted to http://stackoverflow.com/ are most likely to be helpful.

If you are not already familiar with a computer language, you should immerse yourself in a self-study Python program for a week or two. If you have not used Python before, then you will also benefit from some time spent in self-study. Three free online courses are

1. CodeAcademy (https://www.codecademy.com/learn/python) offers an interactive tutorial for beginners.
2. LearnPython (http://www.learnpython.org/) allows the student to work with `Python code` directly from a web browser. This is a good way to start learning immediately since installing `Python` is not necessary.
3. Google offers a course (https://developers.google.com/edu/python/) that assumes some familiarity with computer programming.

Once you have completed such a course, though, we recommend you dive in. Most of the data scientists we know learned at least one of their primary programming languages by working through real-world problems (often, sadly, with real-world deadlines!) There's a large number of relatively inexpensive books written on `Python` for those that have more time and an interest in becoming skilled rather than just competent. Our favorite book is Ramalho's *Fluent Python* [47]. Slatkin's *Effective Python* [57] is very helpful for developing good programming style and habits.

1.9 R

Data scientists often find themselves carrying out analyses that are statistical in nature. The ability to conduct statistical analyses and function in the statistical world is tremendously valuable for the practicing data scientist. The statistical package `R` [46] is the environment of choice among data scientist and for good reason. `R` is an object-oriented programming language with a huge number of excellent statistically-oriented functions and third-party packages. It's also free. You can work directly in the `R` environment but there are several front-ends that improve the experience. We use `RStudio` (https://www.rstudio.com/). Using `R` is not the only way forward though. There are `Python` packages, namely, `Numpy`, `pandas`, `statsModels`, and `matplotlib` that can do some of the same things as `R`. The `Python` packages, however, are not as mature and seamless as `R`. At the present time, knowing how to use `R` for modeling and statistics, and being able to build graphics with the `R` graphics package `ggplot2` [65] offer a clear advantage to the data scientist.

As with `Python`, there's an large collection of books written about `R` and doing statistics with `R`. Maindonald and Braun's *Data Analysis and Graphics Using R* [38] and Albert and Rizzo's *R by Example* [3] are excellent choices.

A few online tutorials for learning `R` are

1. O'Reilly's interactive tutorial at http://tryr.codeschool.com/ is appropriate if you have never used `R`.

2. The Institute for Digital Research and Education offers a self-directed tutorial: http://www.ats.ucla.edu/stat/r/. This tutorial is appropriate if you have had a little exposure to `R` or are proficient with a scripting language.
3. The site https://cran.r-project.org/doc/manuals/r-release/R-intro.html has an ancient manual for those that like do things the hard way.

1.10 Terminology and Notation

Data consists of measurements on observational units. An observational unit renders an observation consisting measurements one or more different attributes or variables.[2] An observational unit in the diabetes example is a U.S. adult resident that responded to the survey, and the attributes that we extracted from the data set are age, education, income, body mass index, and diabetes. We use the traditional statistical term *variable* interchangeably with attribute. In the Federalist papers example, an observational unit is one of the 85 papers. There are 1103 attributes associated with each observation, and the observation rendered from a particular paper consists of the number of uses of each word. We usually use n to denote the number of observations in a data set and p to denote the number of attributes used in the analysis.

1.10.1 Matrices and Vectors

A vector consisting of p elements is denoted as

$$\underset{p\times 1}{\mathbf{y}} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_p \end{bmatrix}.$$

A vector may be thought of as a $p \times 1$ matrix. The transpose of $\mathbf{y}$ is a row-vector or, equivalently, $1 \times p$ matrix, and so $\mathbf{y} = [y_1 \;\; y_2 \;\; \cdots \;\; y_p]^T$. A matrix is two-dimensional array of real numbers. For example,

$$\underset{n\times p}{\mathbf{Y}} = \begin{bmatrix} y_{1,1} & y_{1,2} & \cdots & y_{1,p} \\ y_{2,1} & y_{2,2} & \cdots & y_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ y_{n,1} & y_{n,2} & \cdots & y_{n,p} \end{bmatrix}. \tag{1.1}$$

[2] Multiple observations may originate from a single unit. For example, studies on growth often involve remeasuring individuals at different points in time.

The subscripting system uses the left subscript to identify the row position and the right subscript to identify the column position of the scalar $y_{i,j}$. Thus, $y_{i,j}$ occupies row i and column j. If the matrix is neither a column nor a row vector, then the symbol representing the matrix is written in upper case and in bold.

We often stack a set of row vectors to form a matrix. For example, $\mathbf{Y}$ (formula (1.1)) may be expressed as

$$\underset{n\times p}{\mathbf{Y}} = \begin{bmatrix} \mathbf{y}_1^T \\ \mathbf{y}_2^T \\ \vdots \\ \mathbf{y}_n^T \end{bmatrix},$$

where $\mathbf{y}_i^T$ is the ith row of $\mathbf{Y}$.

The product of a scalar $a \in \mathbb{R}$ and a matrix $\mathbf{B}$ is computed by multiplying every element in the matrix $\mathbf{B}$ by a. For example, $a\underset{1\times q}{\mathbf{b}^T} = [ab_1 \;\; ab_2 \;\; \cdots \;\; ab_q]$. Two products involving vectors are used frequently in this book, the inner product and the outer product. The inner product of $\mathbf{x}$ and $\mathbf{y}$ is

$$\underset{1\times p}{\mathbf{x}^T}\underset{p\times 1}{\mathbf{y}} = \sum_{i=1}^{p} x_i y_i = \mathbf{y}^T\mathbf{x}.$$

The inner product is defined only if the lengths of the two vectors are the same, in which case, the vectors are said to be *conformable.*

The product of two conformable matrices $\mathbf{A}$ and $\mathbf{B}$ is

$$\underset{p\times n}{\mathbf{A}}\underset{n\times q}{\mathbf{B}} = \underset{p\times q}{\begin{bmatrix} \mathbf{a}_1^T\mathbf{b}_1 & \cdots & \mathbf{a}_1^T\mathbf{b}_q \\ \mathbf{a}_2^T\mathbf{b}_1 & \cdots & \mathbf{a}_2^T\mathbf{b}_q \\ \vdots & \ddots & \vdots \\ \mathbf{a}_p^T\mathbf{b}_1 & \cdots & \mathbf{a}_p^T\mathbf{b}_q \end{bmatrix}}, \tag{1.2}$$

where $\mathbf{a}_i^T$ is the ith row of $\mathbf{A}$ and $\mathbf{b}_j$ is the jth column of $\mathbf{B}$.

The outer product of a vector $\mathbf{x}$ with another vector $\mathbf{w}$ is a matrix given by

$$\underset{p\times 1}{\mathbf{x}}\underset{1\times q}{\mathbf{w}}^T = \begin{bmatrix} x_1w_1 & \cdots & x_1w_q \\ x_2w_1 & \cdots & x_2w_q \\ \vdots & \ddots & \vdots \\ x_pw_1 & \cdots & x_pw_q \end{bmatrix}. \tag{1.3}$$

Generally, $\mathbf{x}\mathbf{w}^T \neq \mathbf{w}\mathbf{x}^T$.

If a matrix $\mathbf{A}$ is square and full rank, meaning that the columns of $\mathbf{A}$ are linearly independent, then there exists an *inverse* $\mathbf{A}^{-1}$ of $\mathbf{A}$. The product of $\mathbf{A}$ and its inverse is the identity matrix

$$\underset{n\times n}{\mathbf{I}} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}. \tag{1.4}$$

Hence, $\mathbf{I} = \mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A}$ since the inverse of $\mathbf{A}^{-1}$ is $\mathbf{A}$. If a solution $\mathbf{x}$ is needed to solve the equation

$$\underset{p\times p}{\mathbf{A}}\underset{p\times 1}{\mathbf{x}} = \underset{p\times 1}{\mathbf{y}}, \tag{1.5}$$

and $\mathbf{A}$ is invertible (that is, $\mathbf{A}$ has an inverse), then the solution of the equation is $\mathbf{x} = \mathbf{A}^{-1}\mathbf{y}$.

1.11 Book Website

The website for the book is http://www.springer.com/us/book/9783319457956.

Part I
Data Reduction

Chapter 2
Data Mapping and Data Dictionaries

Abstract This chapter delves into the key mathematical and computational components of data analytic algorithms. The purpose of these algorithms is to reduce massively large data sets to much smaller data sets with a minimal loss of relevant information. From the mathematical perspective, a data reduction algorithm is a sequence of *data mappings*, that is, functions that consume data in the form of sets and output data in a reduced form. The mathematical perspective is important because it imposes certain desirable attributes on the mappings. However, most of our attention is paid to the practical aspects of turning the mappings into code. The mathematical and computational aspects of data mappings are applied through the use of *data dictionaries.* The tutorials of this chapter help the reader develop familiarity with data mappings and `Python` dictionaries.

2.1 Data Reduction

One of principal reasons that data science has risen in prominence is the accelerating growth of massively large data sets in government and commerce. The potential exists for getting new information and knowledge from the data. Extracting the information is not easy though. The problem is due to the origins of the data. Most of the time, the data are not collected according to a design with the questions of interest in mind. Instead, the data are collected without planning and for purposes tangential to those of the analyst. For example, the variables recorded in retail transaction logs are collected for logistical and accounting purposes. There's information in the data about consumer habits and behaviors, but understanding consumer behavior is not the purpose for collecting the data. The information content is meager with respect to the objective. The analyst will have to work hard

B. Steele et al., *Algorithms for Data Science*,
DOI 10.1007/978-3-319-45797-0_2

to get the information and the strategy must be developed intelligently and executed carefully. Tackling the problem through the use of data mappings will help develop the strategy.

The second major hurdle in extracting information from massively large data sets materializes after the data analyst has developed a strategy. Translating the strategy to action will likely require a substantial programming effort. To limit the effort, we need a language with objects and functions compatible with data mapping. `Python` is the right language, and the `Python` dictionary is the right structure for building data reduction algorithms.

The next section discusses a fairly typical data source and database to provide some context for the discussion of data reduction and data mappings.

2.2 Political Contributions

In April of 2014, the Supreme Court struck down a 40 year-old limit on campaign contributions made by individuals within a single 2-year election cycle.[1] Many people believe that the ruling allows the very wealthy to have undue influence on election outcomes. Anyone that attempts to analyze the relationship between contributions and candidates must recognize that popular candidates attract contributions. A simple analysis of total contributions and election winners cannot produce evidence of causation. However, there is a rich, publicly available data source maintained by the Federal Election Commission that may be mined to learn about the contributors and recipients of money spent in the electoral process.

Let's suppose that you are running an electoral campaign and you have a limited budget with which to raise more money. The money should be spent when potential contributors are most likely to contribute to your campaign. But when? The Federal Election Commission data sets can be used to determine when people contribute. That's not exactly the answer to the question, but it is close. To answer that question, we computed the dollar value of contributions reported to the Federal Election Commission for each day of a 2-year period roughly corresponding to the 2012–2014 election cycle. Figure 2.1 shows that daily total contributions were fairly static for most of the cycle except for temporary increases soon before the end of the year. Closer inspection revealed that contributions increase briefly before December 31, March 31, June 30, and September 30, dates that mark the end of the fiscal quarters. A much greater and lasting increase began in September of 2014, often recognized as the beginning of the political campaign season. There are also substantial differences between weekday and weekend contributions perhaps because individuals and not corporations make donations on weekends.

[1] Election cycles correspond to the 2-year terms of the Representatives to the U.S. Congress.

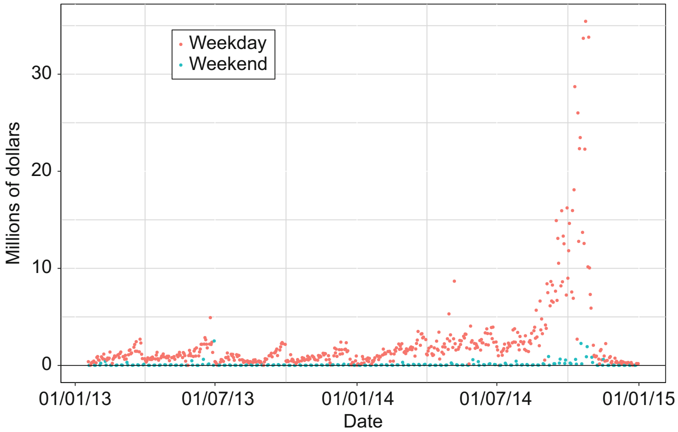


Fig. 2.1 Donation totals reported to the Federal Election Commission by Congressional candidates and Political Action Committees plotted against reporting date

The Federal Election Campaign Act requires candidate committees and political action committees (PACs) to report contributions in excess of $200 that have been received from individuals and committees. Millions of large individual contributions (that is, larger than $200) are reported in a 2-year election cycle. The 2014–2016 data set contained more than 12 million records as of July 1, 2016. Data from 2003 to the most recent election cycle are publicly available from the Federal Election Commission webpage http://www.fec.gov/disclosure.shtml. Three data file types of particular interest are: (a) contributions by individuals; (b) committee master files containing information about Political Action Committees, campaign committees, and other committees raising money to be spent on elections; and (c) candidate master files containing information about the candidates. The contributions by individuals files contain records of large contributions from individuals. The files list the contributor's name and residence, occupation, and employer. They also list the transaction amount and a code identifying the recipient committee. Some data entries are missing or non-informative. A contribution record looks like this:

```
C00110478|N|M3||15|IND|HARRIS, ZACHARY|BUTTE|MT|59701|PEABODY COAL|225
```

We see that Zachary Harris, an employee of Peabody Coal, made a contribution of $225 to the committee identified by `C00110478`.

Another application of the FEC data is motivated by the widely-held belief that big contributions buy political influence [21]. If this is true, then we ought to find out who contributed the largest sums of money. The principal effort in identifying the biggest contributors is the reduction of two million or more contribution records to a short list of contributors and sums. In what follows, reduction is accomplished by creating a `Python` dictionary containing the names and contribution sums of everyone that appears in the contributions by individuals data file.

2.3 Dictionaries

A Python dictionary is much like a conventional dictionary. A conventional dictionary is set of pairs. Each pair consists of a word (the Python analog is a *key*) and a definition (the Python analog is a *value*). Both types of dictionaries are organized around the key in the sense that the key is used to find the value. Keys are unique. That is, a key will only appear once in the dictionary. An example of three key-value pairs from a dictionary of contributors and contributions sums from the 2012–2014 election cycle is

$$\begin{aligned} \text{'CHARLES G. KOCH 1997 TRUST'} &: 5000000,\\ \text{'STEYER, THOMAS'} &: 5057267,\\ \text{'ADELSON, SHELDON'} &: 5141782. \end{aligned}$$

The keys are contributor names and the values are the totals made by the contributor during the election cycle. If there was more than one contribution made with the same name, then we store the sum of the contributions with the name. Python places some constraints on the types of objects to be used as keys but the values are nearly unconstrained; for example, values may be integers as above, strings, sets, or even dictionaries. In the example above, the number of entries in the contributor dictionary will not be dramatically less than the number of records in the data file. However, if the objective were to identify geographic entities from which the most money originated, then United States zip codes could be used as keys and the number of dictionary entries would be about 43,000.

2.4 Tutorial: Big Contributors

The objective of this tutorial is to construct a Python dictionary in which individual contributors are the keys and the values are the sum of all contributions made by the contributor in an election cycle. Some care must be taken in how the dictionary is constructed because big contributors make multiple contributions within an election cycle. Therefore, the value that is stored with the key must be a total, and we have to increment the total whenever second and subsequent records from a particular contributor are encountered in the data file.

After constructing the dictionary, the entries are to be sorted by the contribution totals (the dictionary values). The result will be a list of the names and contribution totals ordered from largest to smallest total contribution.

Proceed as follows:

1. Navigate to http://www.fec.gov/finance/disclosure/ftpdet.shtml, the Federal Election Commission website. Select an election cycle by clicking on one of the election cycle links. The individual contributions file appear as

indivxx.zip where xx is the last two digits of the last year of the election cycle, e.g., indiv14.zip contains data from the 2012–2014 election cycle. Download a file by clicking on the name of the zip file.

2. Before leaving the website, examine the file structure described under Format Description. In particular, note the column positions of the name of the contributor (8 in the 2012–2014 file) and the transaction amount (15 in the 2012–2014 file).

   Beware: the FEC labels the first column in their data sets as one, but the first element in a list in Python is indexed by zero. When the Python script is written, you will have to subtract one from the column position of a variable to obtain the Python list position of the variable.
3. Unzip the file and look at the contents. The file name will be itcont.txt. Opening large files in an editor may take a long time, and so it's useful to have a way to quickly look at the first records of the file. You may do the following:

   a. If your operating system is Linux, then open a terminal, navigate to the folder containing file and write the first few records of the file to the terminal using the instruction

```
cat itcont.txt | more
```

   Pressing the enter key will print the next row. Pressing the Ctrl+C key combination will terminate the cat (concatenate) command. You'll see that attributes are separated by the pipe character: |.

   b. If your operating system is Windows 7, then open a command prompt, navigate to the folder containing the file, and write first 20 records of the file to the window using the instruction

```
head 20 itcont.txt
```

   c. If your operating system is Windows 10, then open a PowerShell, navigate to the folder containing the file, and write first 20 records of the file to the window using the instruction

```
gc itcont.txt | select -first 20
```

4. Create a Python script—a text file with the py extension. Instruct the Python interpreter to import the sys and operator modules by entering the following instructions at the top of the file. A module is a collection of functions that extend the core of the Python language. The Python language has a relatively small number of commands—this is a virtue since it makes it relatively easy to master a substantial portion of the language.

```
import sys
import operator
```

5. Import a function for creating `defaultdict` dictionaries from the module `collections` and initialize a dictionary to store the individual contributor totals:

```
from collections import defaultdict
indivDict = defaultdict(int)
```

   The `int` argument passed to the `defaultdict` function specifies that the dictionary values in `indivDict` will be integers. Specifically, the values will be contribution totals.
6. Specify the path to the data file. For example,

```
path = ’home/Data/itcont.txt’
```

   If you need to know the full path name of a directory, submit the Linux command `pwd` in a terminal. In Windows, right-click on the file name in Windows Explorer and select the Properties option.
7. Create a file object using the `open` function so that the data file may be processed. Process the file one record at a time by reading each record as a single string of characters named `string`. Then, split each string into substrings wherever the pipe symbol appears in the string. The code for opening and processing the file is

```
with open(path) as f:     # The file object is named f.
    for string in f:      # Process each record in the file.
        data = string.split("|")  # Split the character string
                                  # and save as a list named data.
        print(data)
        sys.exit()
```

   The statement `data = string.split("|")` splits the character string at the pipe symbol. The result is a 21-element list named `data`. The instruction `print(len(data))` will print the length of the list named `data`. You can run the print statement from the console or put it in the program.

   The `sys.exit()` instruction will terminate the program. The data file will close when the program execution completes or when execution is terminated. Execute the script and examine the output for errors. You should see a list of 21 elements.

The Python language uses indentation for program flow control. For example, the for string in f: instruction is nested below the with open(path) as f: statement. Therefore, the with open(path) as f: statement executes as long as the file object f is open. Likewise, every statement that is indented below the for string in f: statement will execute before the flow control returns to the for statement. Consequently, for string in f reads strings until there are no more strings in f to be read. The object f will close automatically when the end of the file is reached. At that point, the program flow breaks out of the with open(path) as f loop.

The convention in this book is to show indentation in a single code segment, but not carry the indentation down to the following code segments. Therefore, it is up to the reader to understand the program flow and properly indent the code.

Let's return to the script.

8. Remove the termination instruction (sys.exit()) and the print instruction. Initialize a record counter using the instruction n = 0. Below the import and path declaration instructions, the program should appear as so:

```
n = 0
with open(path) as f:     # The file object is named f.
    for string in f:      # Process each record in the file.
        data = string.split("|")
```

9. In the Python lexicon, an *item* is a key-value pair. Each item in the dictionary indivDict consists of the name of a contributor (the key) and the total amount that the contributor has contributed to all election committees (the value).

   Items will be added to the dictionary by first testing whether the contributor name is a dictionary key. If the contributor's name is not a key, then the name and contribution amount are added as a key-value pair to the dictionary. On the other hand, if the candidate's name is already a key, then the contribution amount is added to the existing total. These operations are carried out using a *factory* function, a function contained in the module collections. This operation is done automatically—we only need one line of code:

```
indivDict[data[7]] += int(data[14])
```

   Add this instruction so that it operates on every list. Therefore, the indentation must be the same as for the instruction data = string.split("|"). Set it up like this:

```
with open(path) as f:
    for string in f:
        data = string.split("|")
        indivDict[data[7]] += int(data[14])
```

10. To trace the execution of the script, add the instructions

```
n += 1
if n % 5000 == 0:
    print(n)
```

within the `for` loop. The instruction `n % 5000` computes the modulus, or integer remainder, of $5000/n$. If n is printed more often than every 5000 lines, then the execution of the program will noticeably slow.

11. After processing the data set, determine the number of contributors by adding the line

```
print(len(indivDict))
```

This instruction should not be indented.

12. Place the instruction `import operator` at the top of the script. Importing the module allows us to use the functions contained in the module.
13. Sort the dictionary using the instruction

```
sortedSums = sorted(indivDict.items(), key=operator.itemgetter(1))
```

The statement `indivDict.items()` creates a list (not a dictionary) consisting of the key-value pairs composing `indivDict`. The `key` argument of the function `sorted()` points to the position within the pairs that are to be used for sorting. Setting `key = operator.itemgetter(1)` specifies that the elements in position 1 of the tuples are to be used for determining the ordering. Since zero-indexing is used, `itemgetter(1)` instructs the interpreter to use the values for sorting. Setting `key = operator.itemgetter(0)` will instruct the interpreter to use the keys for sorting.

14. Print the names of the contributors that contributed at least $25,000:

```
for item in sortedSums :
    if item[1] >= 25000 : print(item[1], item[0])
```

15. The list of largest contributors likely will show some individuals.

Transforming the individual contributors data set to the dictionary of contributors did not dramatically reduce the data volume. If we are to draw inferences about groups and behaviors, more reduction is needed. Before proceeding with further analysis, we will develop the principles of data reduction and data mapping in detail.

2.5 Data Reduction

Data reduction algorithms reduce data through a sequence of mappings. Thinking about a data reduction algorithm as a mapping or sequence of mappings is a key step towards insuring that the final algorithm is transparent and computationally efficient. Looking ahead to the next tutorial, consider an algorithm that is to consume an individual contribution data set A. The objective is to produce a set of pairs E in which each pair identifies a major employer, say Microsoft, and the amounts contributed by company employees to Republican, Democratic, and other party candidates. The output of the algorithm is a list of pairs with the same general form as r:

$$r = \big(\text{Microsoft} : \big[(D, 20030), (R, 4150), (\text{other}, 0)\big]\big). \tag{2.1}$$

We say that the algorithm maps A to E. It may not be immediately obvious how to carry out the mapping, but if we break down the mapping as a sequence of simple mappings, then the algorithm will become apparent. One sequence (of several possible sequences) begins by mapping individual contribution records to a dictionary in which each key is an employer and the value is a list of pairs. The pairs consist of the recipient committee code and the amount of the contribution. For instance, a dictionary entry may appear so:

$$\big(\text{Microsoft} : \big[(C20102, 200), (C84088, 1000), \ldots\big]\big),$$

where $C20102$ and $C84088$ are recipient committee identifiers. Be aware that the lengths of the lists associated with different employers will vary widely and the lists associated with some of the larger employers may number in the hundreds. Building the dictionary using `Python` is straightforward despite the complexity of the dictionary values.

We're not done of course. We need the political parties instead of the recipient committee identifiers. The second mapping consumes the just-built dictionary and replaces the recipient committee code with a political party affiliation if an affiliation can be identified. Otherwise, the pair is deleted. The final mapping maps each of the long lists to a shorter list. The shorter list consists of three pairs. The three-pair list is a little complicated: the first element of a pair identifies the political party (e.g., Republican) and the second element is the sum of all employee contributions received by committees with the corresponding party affiliation. Line 2.1 shows a hypothetical entry in the final reduced data dictionary.

It may be argued that using three maps to accomplish data reduction is computationally expensive, and that computational efficiency would be improved by using fewer maps. The argument is often misguided since the effort needed to program and test an algorithm consisting of fewer but more sophisticated and complicated mappings shifts the workload from the computer to the programmer.

2.5.1 Notation and Terminology

A *data mapping* is a function $f : A \mapsto B$ where A and B are data sets. The set B is called the *image* of A under f and A is the called *preimage* of B. Suppose that $\mathbf{y}$ is a vector of length p (Chap. 1, Sect. 1.10.1 provides a review of matrices and vectors). For instance, $\mathbf{y}$ may be constructed from a record or row in a data file. To show the mapping of an element $\mathbf{y} \in A$ to an element $\mathbf{z} \in B$ by f, we write

$$f : \mathbf{y} \mapsto \mathbf{z} \text{ or } f(\mathbf{y}) = \mathbf{z}.$$

We also write $B = f(A)$ to indicate that B is the image of A. The statement $B = f(A)$ implies that for every $\mathbf{b} \in B$ there exists $\mathbf{a} \in A$ such that $\mathbf{a} \mapsto \mathbf{b}$.

For the example above, it's possible to construct a single map f from A to E though it's less complicated to define two mappings that when applied in sequence, map A to E. If the first mapping is g and the second is h, then data reduction is carried out according to $g : A \mapsto B$ and then $h : B \mapsto E$, or more simply by realizing that f is the composite of g and h, i.e., $f = h \circ g$.

As a point of clarification, a *list* of length p in the `Python` lexicon is equivalent to a tuple of length p in mathematics. The elements of a list may be changed by replacement or a mathematical operation. If we may change the values of an object, then the object is said to be *mutable*. For instance, a `Python` list is *mutable*. The term *tuple* in the `Python` lexicon refers to an immutable list. The elements of a tuple cannot be changed. You may think of a tuple as a universal constant while the program executes.

The order or position of elements within a tuple is important when working with tuples and `Python` lists; hence, $\mathbf{x} \neq \mathbf{y}$ if $\mathbf{x} = (1, 2)$ and $\mathbf{y} = (2, 1)$. In contrast, the elements of sets may be rearranged without altering the set, as illustrated by $\{1, 2\} = \{2, 1\}$. The `Python` notation for a list uses brackets, so that mathematical tuples are expressed as `Python` lists by writing `x = [1,2]` and `y = [2,1]`. The beginning and end of a `Python` tuple (an immutable list) is marked by parentheses. For example, `z = (1,2)` is a `Python` two-tuple. You may verify that `x` $\neq$ `z` (submit `[1,2]==(1,2)` in a `Python` console). You can create a tuple named `a` from a list using the `tuple` function; e.g., `a = tuple([1,2])`; likewise, a list may be formed from a tuple using the statement `a = list[(1,2)]`. However, if you submit `[(1,2)]`, the result is not a list consisting of the elements 1 and 2. Instead the result is a list containing the tuple $(1, 2)$.

2.5.2 The Political Contributions Example

To make these ideas concrete we'll continue with the political contributions example and define three data mappings that reduce an input data file on the order of a million records to an output file with about 1000 records. The first mapping $g : A \mapsto B$ is defined to take a contribution record $\mathbf{y} \in A$ and map it to a three-tuple $\mathbf{b} = (b_1, b_2, b_3) \in B$ where b_1 is the employer of the contributor, b_2 is the political party of the recipient of the contribution, and b_3 is the contribution amount.

The set B will be only slightly smaller than A since the mapping does little more than discard entries in A with missing values for the attributes of interest. The second mapping h takes the set B as an input and computes the total contribution for each employer by political party. Supposing that the political parties are identified as Republican, Democratic, and Other, then the image of B under $h : B \mapsto C$ may appear as $C = \{\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_n\}$ where, for example,

$$\begin{aligned}
\mathbf{c}_1 &= (\text{Microsoft}, \text{Republican}, 1000)\\
\mathbf{c}_2 &= (\text{Microsoft}, \text{Democratic}, 70000)\\
\mathbf{c}_3 &= (\text{Microsoft}, \text{Other}, 350)\\
&\ \vdots\\
\mathbf{c}_n &= (\text{Google}, \text{Other}, 5010).
\end{aligned}$$

The data set C is convenient for producing summary tables or figures. However, C may be further reduced by a third mapping k that combines the records for a particular employer as a single pair with a somewhat elaborate structure. For example, the key-value pair $\mathbf{d}_1 \in D = k(C)$, may appear as

$$\mathbf{d}_1 = (d_{11}, \mathbf{d}_{12}) = \Big(\text{Microsoft}, \big((D, 20030), (R, 4150), (\text{other}, 0)\big)\Big) \tag{2.2}$$

where d_{11} =Microsoft. The second element of the pair $\mathbf{d}_1$ is a three-tuple $\mathbf{d}_{12} = (\mathbf{d}_{121}, \mathbf{d}_{122}, \mathbf{d}_{123})$ where the elements of the three-tuple are pairs. Hence, the first element of the three-tuple, $\mathbf{d}_{121} = (D, 20030)$, is a pair in which the first element identifies the Democratic party and the second element is the total contribution made by employees to candidates affiliated with the Democratic party.

The distribution of contributions to Democratic and Republican candidates made by employees of 20 companies are shown in Fig. 2.2 as an illustration of the results of the data reduction process described above. There is a substantial degree of variation between companies with respect to the distribution; for instance, the split between Democratic and Republican parties is nearly equal for Morgan Stanley, but very lopsided for Google (which appears twice) and Harvard. Those companies that reveal the least amount of diversity in party contributions contribute predominantly to Democratic party candidates.

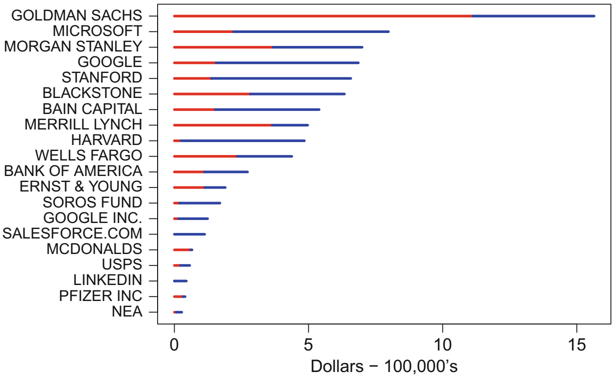


Fig. 2.2 Contributions to committees by individual contributors aggregated by employer. Length of the *red lines* represents the total amount to Republican party candidates and the length of the *blue lines* represents total contributions to Democratic party candidates

2.5.3 Mappings

Let us take a mathematical view of data mappings to help identify some properties, that if lacking, imply that an algorithm is at risk of not achieving its intended purpose.

The first of these mathematical properties is that a mapping must be *well-defined*. A mapping $f : D \to E$ is well-defined if, for every $\mathbf{x} \in D$ there is one and only one output $\mathbf{y} = f(\mathbf{x})$.

If an algorithm is treated as a well-defined mapping, then there can only be one and only one output of the algorithm for every possible input. A second essential property of a data mapping is that every output $\mathbf{y}$ must belong to the image set E. This implies that all possible outputs of the function must be anticipated and that the algorithm does not produce an unexpected or unusable output, say, an empty set instead of the expected four-element tuple. This condition avoids the possibility of an subsequent error later in the program. Computer programs that are intended for general use typically contain a significant amount of code dedicated to checking and eliminating unexpected algorithm output.

We define a *dictionary mapping* to be a mapping that produces a key-value pair. The key is a label or index that identifies the key-value pair. The key serves as a focal point about which the mapping and algorithm is constructed. In the example above, a natural choice for keys are the employer names because the objective is to summarize contributions to Democratic and Republican political parties by employers. A collection of key-value pairs will be called a *dictionary* in accordance with `Python` terminology.[2] A dictionary item is a key-value pair, say $\mathbf{e}_i = (\text{key}_i, \text{value}_i)$, and if the mapping is $f : D \mapsto E$, then we will refer to $f(D) = \{\mathbf{e}_1, \ldots, \mathbf{e}_n\}$ as the dictionary produced by the application of f to the data set D. Keep in mind that a `Python` dictionary is not a set. It has it's own type–`dict`. However, since the items in a dictionary are unique, we tend to think of dictionaries as sets. You may determine the type of an object `u` with the instruction `type(u)`.

2.6 Tutorial: Election Cycle Contributions

The aim of this tutorial is translate the logical and mathematical descriptions of the data reduction mappings above into `Python` code for the problem of summarizing the political orientation of company employees. It has been conjectured that the existence a corporate culture within a company fosters insularity, uniformity of thought, and suppression of originality, all of which are presumed to have negative effects on creativity and innovation. But, it's difficult to uncover objective evidence that supports this conjecture. We may, however, investigate the political complexion of a corporation, more specifically, the political preferences of the employees of a company by studying the recipients of their contributions. If the results of the investigation reveal that one political party is the dominant recipient of contributions, then we have some evidence of a tendency for employees to align politically.

The first task of this investigation is to build a dictionary that identifies a political party associated with an individual's contribution. The keys, however, are not individuals but instead are the employers of the individuals. To accomplish the task, a Federal Elections Commission Individual Contributions data file will be mapped to a dictionary in which a key is an employer and the associated value is a list of n pairs, where each pair consists of a recipient political party and contribution amount. The second task is to build another dictionary to store contribution totals by employer and political party. This task is accomplished by a mapping the n pairs associated with an employer to three pairs (corresponding to Republican, Democratic, and other parties)

As with Tutorial 2.4, you'll process an Individual Contributions data file for some election cycle. From each record, extract the employer, the

[2] `Python` dictionaries are equivalent to `Java` hashmaps.

contribution amount and the recipient code. If there is an employer listed, then determine whether there is a political party associated with the recipient code.

We'll need another dictionary that links political party to recipient. If there is a political party associated with the recipient, it will be recorded in one of two FEC files since there are two types of recipients: candidate committees and other committees. If the recipient is a candidate committee, then the committee information is in the *Candidate Master* file and there's a good chance that a political party will be identified in the file. The other recipients, or other committees, encompass political action committees, party committees, campaign committees, or other organizations spending money related to an election. Information on other committees is contained in the *Committee Master* file and a political party sometimes is identified with the committee in this file. Therefore, once the recipient code is extracted from the Individual Contributions data file, you'll have to check for the recipient code in the Candidate Master file; if it's not there, then check the Committee Master file.

The program is somewhat complicated because two dictionaries must be searched to determine the party affiliation of a receiving candidate. We wind up processing three files and building three dictionaries. Table 2.1 is shown for reference.

Table 2.1 Files and dictionaries used in Tutorial 2.6. The file `cn.txt` is a Candidate Master file and `cm.txt` is a Committee Master file. The file `itcont.txt` is a Contributions by Individual file. Field positions are identified using zero-indexing

		Attributes and field positions			
File	Dictionary	Key	Column	Value	Column
`cn.txt`	`canDict`	Committee code	9	Political party[a]	2
`cm.txt`	`comDict`	Committee code	0	Political party	10
`itcont.txt`	`employerDict`	Employer	11	Amount	14

[a] The recipient's political party is determined by searching the `canDict` and `comDict` dictionaries for the committee code associated with the individual's contribution

1. Decide on an election cycle and, for that cycle, download and unzip the (a) Candidate Master File; (b) Committee Master File; and (c) Contributions by Individuals Files from the Federal Elections Commission webpage http://www.fec.gov/finance/disclosure/ftpdet.shtml.
2. Build a candidate committee dictionary from the Candidate Master file using the principal campaign committee codes in field position 9 (the zero-indexed column) as keys and party affiliation in position 2 as values. We will refer to this dictionary by the name `canDict`.

```
canDict = {}
path = '../cn.txt'
with open(path) as f:
    for line in f:
        data = line.split("|")
        canDict[data[9]] = data[2]
```

Candidate identifiers appear only once in the Candidate Master file so it's not necessary to test whether `data[9]` is a dictionary key before creating the key-value pair.

3. Build an *other* committees dictionary from the Committee Master file. We'll use the `Python` indexing convention of indexing the first element of a list with 0. The keys are to be the committee identification codes located in field position 0 of Committee Master file and the values are the committee parties located in field position 10. We will refer to this dictionary by the name `otherDict`.

```
otherDict = {}
path = '../cm.txt'
with open(path) as f:
    for line in f:
        data = line.split("|")
        otherDict[data[0]] = data[10]
```

4. Build the dictionary `employerDict`. Keys will be employers and values will be a list of pairs. Each pair in a list will be a political party and a contribution amount. To build the dictionary, process the Contributions by Individuals (`itcont.txt`) data file one record at a time.[3]

   The first task to perform with each record is to determine what, if any political party affiliation is associated with the recipient of the contribution. The search begins with the Filer Identification number—`data[0]`. The filer is the recipient committee and the entity that filed the report to the Federal Elections Commission (the individual does not file the report). Determine if there is a party listed for the recipient committee. First, check the candidate committee dictionary in case the recipient committee is a candidate committee. Look for a party name entry in the dictionary named `canDict`. If the filer identification number is not a key in this dictionary, then look for an entry in the other committee dictionary named `otherDict`. Then create a two-tuple, `x`, with the party and the amount. Use the `int` function to convert the contribution amount (stored as a string) to an integer. The code follows:

[3] You may be able to reuse your code from the tutorial of Sect. 2.4.

```
path = ’../itcont14.txt’
n = 0
employerDict = {}
with open(path) as f:
    for line in f:
        data = line.split("|")
        party = canDict.get(data[0])
        if data[0] is None:
            party = otherDict[data[0]]
        x = (party, int(data[14]))
```

5. The next step is to save `x` in the dictionary named `employerDict`. Each key is to be the employer of a contributor and the value will be a list of contribution pairs (the `x`’s) built in step 4. If there is an entry in `data` for employer, it will be in position 11. So, extract `data[11]` and if it is not empty string, then assign the string to `employer`. If `employer` is an empty string, then ignore the record and process the next record. Assuming that there is an entry for `employer`, test whether it is a dictionary key. If `employer` is not a dictionary key, then create the dictionary entry by assigning a list containing `x` using the instruction `employerDict[employer] = [x]`. On the other hand, if `employer` is a key, then append `x` to the dictionary value associated with `employer` using the instruction `employerDict[employer].append(x)`:

```
employer = data[11]
if employer != ’’:
    value = employerDict.get(employer)
    if value is None:
        employerDict[employer] = [x]
    else:
        employerDict[employer].append(x)
```

Don’t forget to indent the code segment. It must be aligned with the `x = (party, int(data[14]))` statement because it is to execute every time that a new record is processed. The pair `x` is appended in the last statement of the code segment.

By construction, the value associated with the key will be a list. For instance, a value may appear as so:

$$\texttt{value} = \texttt{[('DEM', 1000), (", 500),('REP', 500)]}. \tag{2.3}$$

Note that one of the entries in `value` is an empty string which implies that the contribution was received by a committee without a political party listed in either the Candidate Master or Committee Master file.

6. It’s helpful to trace the progress as the script executes. Count the number of records as the file is processed and print the count at regular intervals. Process the entire file.
7. The last substantive set of code will reduce `employerDict` to the dictionary illustrated by Eq. (2.2). Having become familiar with dictionaries, we may describe the reduced dictionary, `reducedDict`, as a dictionary of dictionaries. The keys of `reducedDict` are to be employers and the values are to be dictionaries. The keys of these internal or sub-dictionaries are the party labels `’Other’`, `’DEM’`, and `’REP’` and the values are the total of contributions to each party made by employees of a company. Suppose that `GMC` is a key of `reducedDict`. Then, the value associated with `GMC` may appear as so:

   $$\texttt{reducedDict[’GMC’]}=\{\texttt{’Other’} : 63000, \texttt{’DEM’} : 73040, \texttt{’REP’}: 103750\}.$$

   The reduced dictionary will be formed by building a dictionary named `totals` for each employer. The dictionary `totals` will then be stored as the value associated with the employer key in `reducedDict`. Begin by initializing the reduced dictionary and iterating over each key-value pair in `employerDict`:

```
reducedDict = {}
for key in employerDict:                # Iterate over employerDict.
    totals = {’REP’:0,’DEM’:0,’Other’:0} # Initialize the dictionary.
    for value in employerDict[key]:
        try :
            totals[value[0]] += value[1]
        except KeyError:
            totals[’Other’] += value[1]
    reducedDict[key] = totals
```

   The value associated with `employerDict[key]` is a list of pairs (political party and contribution amount). The `for` loop that iterates over `employerDict[key]` extracts each pair as `value`. The first element `value[0]` is a party and the second element `value[1]` is a dollar amount. The first two lines of this code segment are not indented. The code

```
try :
    totals[value[0]] += value[1]
```

   attempts to add the contribution amount stored in `value[1]` with the party name stored in `value[0]`. If the party name is not `’REP’`, `’DEM’` or `’Other’`, then the `Python` interpreter will produce a `KeyError` exception (an error) and program flow will be directed to the instruction `totals[’Other’] += value[1]`. The result is that the contribution amount is added to the other party total. The `try` and `except` construct is called an *exception handler*.

8. We will want to sort the entries in the employer dictionary according to the total of all contributions made by employees of an employer. Immediately after the statement `reducedDict = {}`, initialize a dictionary named `sumDict` to contain the total of all contributions made by employees of each employer. Add an instruction that computes the sum of the three dictionary values and stores it with the employer key. The instruction is

```
sumDict[key] = totals['REP'] + totals['DEM'] + totals['Other']
```

   The instruction executes immediately after the assignment instruction `reducedDict[key] = totals` from step 7 and it should be aligned with it.

9. Add an instruction so that the execution of the script may be monitored, say,

```
if sumDict[key] > 10000 : print(key, totals)
```

   Indent this instruction so that it executes on every iteration of the `for` loop. This print statement is the last instruction in the `for` loop.

10. Now that `sumDict` has been built, we will create a list from the dictionary in which the largest contribution sums are the first elements. Specifically, sorting `sumDict` with respect to the sums will create the sorted list. The resulting list consists of key-value pairs in the form $[(\mathtt{k}_1, \mathtt{v}_1), \ldots, (\mathtt{k}_n, \mathtt{v}_n)]$ where $\mathtt{k}_i$ is the ith key and $\mathtt{v}_i$ is the ith value. Because the list has been sorted, $\mathtt{v}_1 \geq \mathtt{v}_2 \geq \cdots \geq \mathtt{v}_n$. Since `sortedList` is a list, we can print the employers with the 100 largest sums using the code

```
sortedList = sorted(sumDict.items(), key=operator.itemgetter(1))
n = len(sortedList)
print(sortedList[n-100:])
```

   If `a` is a list, then the expression `a[:10]` extracts the first ten elements and `a[len(a)-10:]` extracts the last 10. These operations are called *slicing.*

11. Write a short list of the largest 200 employers to a text file. We'll use `R` to construct a plot similar to Fig. 2.2. The code follows.

```
path = ’../employerMoney.txt’
with open(path,’w’) as f:
   for i in range(n-200, n):
        employerName = sortedList[i][0].replace("’", "")
        totals = reducedDict[employerName]
        outputRecord = [employerName] + [str(x) for x in totals.
                       values()] + [str(sortedSums[i][1])]
        string = ’;’.join(outputRecord) +’\n’
        f.write(string)
```

The ’w’ argument must be passed in the call to open to be able to write to the file. Some employer names contain apostrophes and will create errors when R reads the file so we must remove the apostrophes from the employer name before the data is written to the output file. Applying the replace operator to the string sortedList[i][0] removes the apostrophes wherever they are found in the string.

The list outputRecord is created by concatenating three lists (each is enclosed by brackets) using the + operator. The middle list is created by list comprehension. List comprehension is discussed in detail in the Sect. 2.7.1. For now, it’s a method of creating a list using a for loop.

In the instruction

```
string = ’;’.join(outputRecord)+’\n’
```

outputRecord is converted to a string using the .join operator. A semicolon joins each list element in the creation of the string. The semicolon serves as a delimiter so that the output file is easy to process in R. A delimiter is a symbol that is used to separate variables. For example, commas are used to separate variables in a comma-delimited file (often known as a csv file.) Lastly, the end-of-line marker \n is concatenated to the list.

12. Use the following R code to generate a figure showing the largest contribution totals to Republican and Democratic parties from company employees. The instruction s = 160:190 effectively ignores the largest ten employers (you can see that most of the largest 10 are not true employers but instead identify contributors that are self-employed, retired, and so on). The instruction order(v) returns a vector of indices that orders the vector v from smallest to largest.

```
Data = read.table(’../Data/employerMoney.txt’ ,sep=’;’, as.is = TRUE)
colnames(Data) = c(’Company’, ’Rep’, ’Dem’, ’Other’, ’Total’)
head(Data)
print(Data[,1])
s = 160:190  # Select specific rows to plot.
D = Data[s,]                # Take the subset.
D = D[order(D$Rep+D$Dem),]  # Re-order the data according to the total.
rep = D$Rep/10^5            # Scale the values.
dem = D$Dem/10^5
mx = max(rep+dem)
names = D[,1]
n = length(rep)
# Fix the plot window for long names.

plot(x = c(0,mx),y=c(1,n),yaxt = ’n’,xlab
  = "Dollars - 100,000’s",cex.axis = .65,typ = ’n’,ylab=’’,cex.lab=.8)
axis(side = 2, at = seq(1,n),labels = names, las = 2, cex.axis = .65)
for (i in 1:n) {
  lines(y=c(i,i),x=c(0,rep[i]),col=’red’,lwd=3)
  lines(y=c(i,i),x=c(rep[i],rep[i]+dem[i]),col=’blue’,lwd=3)
}
par(oma=c(0,0,0,0))  # Reset the plotting window to default values.
```

2.7 Similarity Measures

Similarity measurements are routinely used to compare individuals and objects. They’re used extensively by recommendation engines—algorithms that recommend products and services to potential customers. The underlying premise is that if viewer **A** is similar to viewer **B** with respect to the movies that they have watched, then movies watched by **A** and not by **B** may be recommended to **B**. In the case of political fund-raising, knowing that contributors to candidate **A** are likely to contribute to candidate **B** allows a campaign committee to more effectively direct their fund raising efforts.

Similarity measurements may also provide insight to a new or little-understood entity, say, a campaign committee that is secretive about its objectives. Suppose that a campaign committee **A** has received contributions from committees collected in the set $A = \{a_1, \ldots, a_n\}$. An analyst can gain an understanding of a secretive committee by determining its similarity to known committees. To proceed, a function is needed that will measure similarity between entities **A** and **B** based on sets A and B. More specifically, we need at least one similarity measure. We will examine two measures of similarity: Jaccard similarity and conditional probabilities.

Let $|A|$ denote the cardinality of the set A. If S is finite, then $|S|$ is the number of elements in S. The Jaccard similarity between sets A and B is the number of elements in both A and B relative to the number of elements in either A or B. Mathematically, the Jaccard similarity is

$$J(A,B)=\frac{|A\cap B|}{|A\cup B|}. \tag{2.4}$$

Jaccard similarity possesses several desirable attributes:

1. If the sets are the same then the Jaccard similarity is 1. Mathematically, if $A=B$, then $A\cap B=A\cup B$ and $J(A,B)=1$.
2. If the sets have no elements in common, then $A\cap B=\emptyset$ and $J(A,B)=0$.
3. $J(A,B)$ is bounded by 0 and 1 because $0\le|A\cap B|\le|A\cup B|$.

Jaccard similarity is particularly useful if all possible elements of $A\cup B$ are difficult or expensive to determine. For example, suppose that individuals are to be grouped according to the similarity of their gut microbiota.[4] The possible number of resident species of microorganisms may number in the hundreds of thousands but the number that is expected be found in a stool sample will tend to be many times less. In this case, similarity should be measured on the basis of the species that are present and not on the basis of species that are absent since the vast majority of possible resident species will not be present. Jaccard similarity depends only on the distribution of species in $A\cup B$ and those species absent from the union have no bearing on the value of $J(A,B)$.

Jaccard similarity is an imperfect similarity measure since if the numbers of elements $|A|$ and $|B|$ are much different, say $|A|<<|B|$,[5] then

$$|A\cap B|\le|A|<<|B|\le|A\cup B|. \tag{2.5}$$

Inequality (2.5) implies that the denominator of the Jaccard similarity (formula (2.4)) will be much larger than the numerator and hence, $J(A,B)\approx 0$. This situation will occur if a new customer, $\mathbf{A}$, is very much like $\mathbf{B}$ in purchasing habits and has made only a few purchases (recorded in A). Suppose that all of these purchases have been made by $\mathbf{B}$ and so whatever $\mathbf{B}$ has purchased ought to be recommended to $\mathbf{A}$. We recognize that $\mathbf{A}$ is similar $\mathbf{B}$, given the information contained in A. But, $J(A,B)$ is necessarily small because the combined set of purchases $A\cup B$ will be much larger in number than the set of common purchases $A\cap B$. There's no way to distinguish this situation between that of two individuals with dissimilar buying habits. Thus, it's beneficial to have an alternative similarity measure that will reveal the relationship.

An alternate measure of similarity that will meaningfully reflect substantial differences in the cardinalities of the sets A and B is the *conditional*

[4] The gut microbiota consists of the microorganism species populating the digestive tract of an organism.

[5] This notation conveys that $|A|$ is much smaller than $|B|$.

probability of an event. The conditional probability of the event A given B is the probability that A will occur given that B has occurred. This conditional probability is denoted as $\Pr(A|B)$ and it defined by

$$\Pr(A|B) = \frac{\Pr(A \cap B)}{\Pr(B)}, \tag{2.6}$$

provided that $\Pr(B) \neq 0$. If $\Pr(B) = 0$, then the conditional probability is undefined and without interest since the event B will not occur. If there are substantive differences between the unconditional probability of A, $(\Pr(A))$ and the conditional probability of A given B, then B is informative with respect to the occurrence of A. On the other hand, if $\Pr(A|B) \approx \Pr(A)$, then B provides little information about the occurrence of A. Furthermore, A and B are said to be independent events whenever $\Pr(A|B) = \Pr(A)$. Lastly, events A and B are independent if and only if $\Pr(A \cap B) = \Pr(A)\Pr(B)$, a statement that may be deduced from formula (2.6) and the definition of independence.

To utilize conditional probabilities in the analysis of political committees, consider a hypothetical experiment in which a committee is randomly drawn from among a list of all committees that have contributed in a particular election cycle. The event A is that of drawing a committee that has contributed to committee **A**. Since committees are randomly selected, $\Pr(A)$ is the proportion of committees that have identified **A** as a recipient of one or more of their contributions. Let $|A|$ denote the number of committees contributing to **A** and n denote the total number of committees. Then,

$$\Pr(A) = \frac{|A|}{n}.$$

The event B is defined in the same manner as A and so $\Pr(B)$ is the proportion of committees contributing to **B**. The probability that a randomly selected committee has contributed to both **A** and **B** during a particular election cycle is $\Pr(A \cap B) = |A \cap B|/n$. The conditional probability of A given B is defined by Eq. (2.6) but can be rewritten in terms of the numbers of contributors:

$$\begin{aligned} \Pr(A|B) &= \frac{|A \cap B|/n}{|B|/n} \\ &= \frac{|A \cap B|}{|B|}. \end{aligned} \tag{2.7}$$

Conditional probabilities may be used to address the deficiencies of the Jaccard similarity measure. Recall that when there are large differences in the frequencies of the events A and B, say $|A| << |B|$, then $J(A, B)$ must be small even if every contributor to **A** also contributed to **B** so that $A \subset B$. It's desirable to have a measure that conveys similarity of **B** to **A**. The conditional probability $\Pr(B|A)$ does so. To see why, suppose that $0 < |A| << |B|$ and that nearly every contributor to $\mathbf{A}$ also contributed to $\mathbf{B}$ and, hence, $A \cap B \approx A$. Then,

$$\begin{aligned} \Pr(A|B) &= \frac{|A \cap B|}{|B|} \approx \frac{|A|}{|B|} \approx 0 \\ \text{and } \Pr(B|A) &= \frac{|A \cap B|}{|A|} \approx 1. \end{aligned} \tag{2.8}$$

Since $\Pr(B|A)$ is nearly 1, it's very likely that any committee that contributes to $\mathbf{A}$ will also contribute to $\mathbf{B}$. In summary, an analysis of the committee similarity will be improved by utilizing conditional probabilities in addition to Jaccard similarities.

It's appropriate to think of the set of Federal Elections Commission records for a particular election cycle as a population and compute the exact probabilities of the events of interest given the experiment of randomly sampling from the list of all committees. Viewing the data set as a population usually is not justified. Instead, the usual situation is that the data are a sample from a larger population or process. For instance, if the data consist of all point-of-sale records collected by a business during a sample window, say, a single day or week, then the data should viewed as a sample from a larger population of records spanning a time span of interest, say, a fiscal quarter or a year. In this sample context, the proportions used above must be viewed as probability estimates, and usually we would define $|A|$ as the number times that the event A occurred during the sample window and n as the number of outcomes (sales) observed during the sample window. The *hat* notation, e.g., $\widehat{\Pr}(A) = |A|/n$ is used to emphasize the uncertainty and error associated with using the estimator $\widehat{\Pr}(A)$ in place of the true, unknown probability $\Pr(A)$. Finally, when probability estimates are computed as relative frequencies, as in this example, the term *empirical probabilities* is often used.

2.7.1 Computation

We now to turn the matter of computing the Jaccard similarity and the conditional probabilities for a large set of committee pairs. An algorithm for forming pairs is needed since the set of committees $\{\mathbf{A}, \mathbf{B}, \mathbf{C}, \ldots\}$ for a particular election cycle is too large to form the set manually. Once the set of pairs has been constructed, then an algorithm will process the pairs and compute the numbers of committees that have contributed to a particular committee, the number of contributors to both committees, and the number of contributors to at least one of the two committees. From these counts, the three similarity measures are computed. The last operation is to sort the pairs with respect to Jaccard similarity, from least to largest and print the committee names, Jaccard similarity, and conditional probabilities.

The `Python` method of *list comprehension* is a concise and computationally efficient method for building the set of pairs $\{(\mathbf{A},\mathbf{B}),(\mathbf{A},\mathbf{C}),(\mathbf{B},\mathbf{C}),\ldots\}$. List comprehension is an analogue of the mathematical syntax for set definition in which the elements of a population or sample space are identified as members of the set by whether or not they satisfy a specified condition. For example, the even integers are $W = \{x|x \bmod 2 = 0\}$.The mathematical definition of W is translated as *the set of values x such that x modulo 2 is 0*. The advantage of list comprehension is that the code is compact yet readily understood and computationally faster than other methods of building a list. A list comprehension expression is contained by a pair of brackets since brackets define lists in `Python`. The brackets contain an expression to be computed followed by one or more `for` or `if` clauses. For example, a list of squares can be constructed using the list comprehension `s = [i**2 for i in range(5)]`. The expression is `i**2` and there is a single `for` clause. An alternative to using list comprehension initializes `s` and then fills it:

```
s = [0]*5  # Create a 5-element list filled with zeros.
for i in range(5):
    s[i] = i**2
```

Another example builds the set of all pairs (i, j) with $i < j$ formed from the set $\{0, 1, \ldots, n\}$. Without list comprehension, a pair of nested `for` clauses are used in which the outer `for` clause iterates over $0, 1, \ldots, n - 1$ and the inner `for` clause iterates over $i + 1, \ldots, n$. Running the inner iteration from $i + 1$ to n insures that i will always be less than j and that all pairs will be formed. Code for forming the set without list comprehension follows:

```
pairs = set({})  # Create the empty set.
for i in range(n):
    for j in range(i+1, n+1, 1):
        pairs = pairs.union({(i,j)})
```

Note that a singleton set `{(i,j)}` is created containing the pair (i, j) before putting the pair into the set `pairs` using the `union` operator. The `union` operator requires that the two objects to be combined as one set are both sets.

Set comprehension is the analogue of list comprehension for building sets instead of lists. For building a set of pairs using set comprehension, two nested for clauses are required:

```
pairs = {(i,j) for i in range(n) for j in range(i+1, n+1, 1)}
```

Since the number of pairs is $n(n-1)/2$, where n is the number of elements from which to form the sets, the number of possible pairs is of the order n^2. When n is much larger than 100, the computational demands of building and processing the set of pairs may be excessive.

Returning to the problem of identifying pairs of similar committees among a large collection of committees, it will be necessary to determine the number of committee pairs that must be examined before computing similarities. Furthermore, it will be necessary to reduce the number of contributing committees in the contributor dictionary by removing some of the committees. For example, if the analysis is limited to political action committees that have made contributions to many recipients, then the population of interest is effectively limited to committees with expansive agendas such as protecting Second Amendment rights. Alternatively, the analysis may be limited to narrow-focus committees by selecting those committees that contributed to fewer than 50 recipients.

The next tutorial provides practice in working with dictionaries and programming the similarity measures discussed above.

2.8 Tutorial: Computing Similarity

The objective of this tutorial is to identify political campaign committees that are alike with respect to their contributors. For each political campaign committee, you are to construct a set of other committees from which the committee in question has received contributions. Let A and B denote two sets of contributing committees. Then, the similarity between the two committees, say, **A** and **B**, will be determined based on the numbers of common contributor committees, and three measures of similarity will be computed: $J(A, B)$, $\Pr(A|B)$ and $\Pr(B|A)$. The final step is to sort the committee pairs with respect to similarity and produce a short list of the most similar pairs of committees.

1. Decide upon an election cycle and retrieve the following files from the FEC website:

   a. The Committee Master file linking the committee identification code (field position 0) with the committee name (field position 1). The compressed Committee Master files are named `cm.zip`.
   b. The between-committee transaction file[6] linking the recipient committee (field position 0) with the contributing committee (field position 7). Between-committee transaction files are named `oth.zip`. The decompressed file has the name `itoth.txt`.

[6] Named *Any Transaction from One Committee to Another* by the FEC.

2. Process the Committee Master file and build a dictionary linking committee identification codes with committee names by setting the dictionary keys to be committee identification codes (located in field position 0 of the data file) and the dictionary values to be the committee names (field position 1 of the data file).

```
path = ’../cm.txt’
nameDict = {}
with open(path) as f:
    for line in f:
        data = line.split("|")
        if data[1] != ’’:
            nameDict[data[0]] = data[1]
```

3. Create a set named `committees` containing the committee identification numbers:

```
print(’Number of committees = ’,len(nameDict))
committees = set(nameDict.keys())
```

   The `.keys()` operator extracts the keys from a dictionary.

4. Construct another dictionary, call it `contributorDict`, in which the dictionary keys are the committee identification codes and the values are sets containing the names of the committees that have contributed to the committee identified by identification code. The value for a new key is created using braces to contain the contributing committee name. Additional committees are combined with the set using the instruction `A.union({a})`, where A is a set and a is an element. For example, $\{a\}$ is the singleton set containing a.

```
path = ’../itoth.txt’
contributorDict = {}
with open(path) as f:
    for line in f:
        data = line.split("|")
        contributor = data[0]
        if contributorDict.get(contributor) is None:
            contributorDict[contributor] = {data[7]}
        else:
            contributorDict[contributor]
                  = contributorDict[contributor].union({data[7]})
```

5. Determine the number of contributors by finding the number of keys in `contributorDict`. Compute the number of pairs.

```
n = len(contributorDict)
print(’N pairs = ’,n*(n-1)/2)
```

   If the number of pairs is large, say greater than 10^5, then its best to reduce the number of committees by selecting a subset. For example, you may limit the set of committees to only those that made at least m contributions in an election cycle. In the following code, key-value pairs are removed using the `pop()` function when the number of committees to which contributions were made is less than or equal to 500. We iterate over a list of dictionary keys because items cannot be deleted from a dictionary while iterating over the dictionary. Check that the number of remaining committees is sufficiently small, say less than 300.

```
for key in list(contributorDict.keys()):
    if len(contributorDict[key]) <= 500:
        contributorDict.pop(key, None)
n = len(contributorDict)
print(’N pairs = ’,n*(n-1)/2)
```

   What’s left in `contributorDict` are committees that are dispersing a lot of money to many political committees, and hence might be viewed as influential.

6. Iterate over `contributorDict` and print the lengths of each value to check the last block of code.

```
for key in contributorDict:
    print(nameDict[key], len(contributorDict[key]))
```

   The length of a contributor value is the number of committees that the committee has contributed to in the election cycle.

7. Extract the keys and save as a list:

```
contributors = list(contributorDict.keys())
```

8. Use list comprehension to build a list containing the $n(n-1)/2$ pairs:

```
pairs = [(contributors[i], contributors[j]) for i in range(n-1)
         for j in range(i+1, n, 1)]
```

9. Compute the similarity between pairs of committees by iterating over `pairs`. For each pair in the list `pairs`, extract the sets of contributors and the compute Jaccard similarity, $\Pr(A|B)$, and $\Pr(B|A)$. Then store the three similarity measures in a dictionary named `simDict`.

```
simDict = {}
for commA, commB in pairs:
    A = contributorDict[commA]  # Set of contributors to commA.
    nameA = nameDict[commA]
    B = contributorDict[commB]
    nameB = nameDict[commB]

    nIntersection = len(A.intersection(B))
    jAB = nIntersection/len(A.union(B))
    pAGivenB = nIntersection/len(B)
    pBGivenA = nIntersection/len(A)

    simDict[(nameA, nameB)] = (jAB, pAGivenB, pBGivenA)
```

The keys for the similarity dictionary are the pairs `(nameA, nameB)` consisting of the names of the committees rather than the committee identification codes.

Pairs may be used as dictionary keys because tuples are immutable. The statement `nIntersection/len(A.union(B))` will perform integer division if the `Python` version is less than 3.0; if your `Python` version is less than 3.0, then the denominator must be cast as a floating point number before division takes place, say:

```
    jAB = nIntersection/float(len(A.union(B)))
```

10. Sort the similarity dictionary using the instruction

```
sortedList = sorted(simDict.items(), key=operator.itemgetter(1),
             reverse=True)
```

The function `sorted` produces a list containing the key-value pairs of `simDict` sorted according to the magnitude of the Jaccard similarity since the `itemgetter()` argument is 1. We've used the optional argument `reverse = True` so that the sorted list goes from largest to smallest Jaccard similarity.

11. Since `sortedList` is a list, we can iterate over the pairs in the list in the following code segment. We refer to the keys of `simDict` as `committees` and the value as `simMeasures` in the `for` statement. `Python` allows the programmer to extract and name the elements of a list or tuple using an

assignment statement such as `nameA, nameB = committees`. Print the committee names, Jaccard similarity, and conditional probabilities from smallest Jaccard similarity to largest:

```
for committees, simMeasures in sortedList:
    nameA, nameB = committees
    jAB, pAB, pBA = simMeasures
    if jAB > .5:
        print(round(jAB, 3), round(pAB, 3),
            round(pBA, 3), nameA + ’ | ’ + nameB)
```

We could use indices to extract values from the similarity dictionary. The *i*th Jaccard similarity value is `sortedList[i][1][0]`, for example. The triple indexing of `sortedList` can be understood by noting that `sortedList` is a list in which the *i*th element is a pair, `sortedList[i]` is a key-value pair, `sortedList[i][0]` is the pair of committee names, and `sortedList[i][0][1]` is the second committee name.

The results of our analysis of the major contributors of 2012 election cycle data are summarized in Table 2.2. The analysis was limited to those committees that made at least 200 contributions during the 2012 election cycle. Because of this prerequisite, Table 2.2 consists almost entirely committees affiliated with large groups of individuals, more specifically, corporations, unions, and associations. The Comcast Corp & NBCUniversal PAC and the Verizon PAC both appeared twice in Table 2.2 showing that these political action committees were very active and also much alike with respect to the recipients of their contributions. The last line of the table shows the entry $\Pr(B|A) = .794$, from which it may be concluded that if the Johnson & Johnson PAC (committee **A**) contributed to a particular entity, then the probability that Pfizer PAC (committee **B**) also contributed to the same entity is .794. On the other hand, given that the Pfizer PAC has contributed to a particular entity, the probability that Johnson & Johnson PAC contributed as well to the committee is much less, .415. Both companies are global pharmaceutical companies. The similarity between the beer wholesalers and realtors is puzzling.

2.9 Concluding Remarks About Dictionaries

A irrevocable property of dictionary keys is that they are immutable. An immutable object resides in a fixed memory location. Immutability is key for optimizing dictionary operations. If we attempted to use a list consisting of two committee identification codes, for instance, `[commA, commB]`, then the `Python` interpreter will produce a `TypeError` because lists are mutable

Table 2.2 The five major committee pairs from the 2012 election cycle with the largest Jaccard similarity. Also shown are the conditional probabilities $\Pr(A|B)$ and $\Pr(B|A)$

Committee **A**	Committee **B**	$J(A,B)$	$\Pr(B\|A)$	$\Pr(A\|B)$
Verizon PAC	Comcast Corp & NBCUniversal PAC	.596	.693	.809
General Electric PAC	Comcast Corp & NBCUniversal PAC	.582	.713	.76
NEA Fund for Children and Public Education	Letter Carriers Political Action Fund	.571	.82	.653
National Beer Wholesalers Association PAC	National Association of Realtors PAC	.57	.651	.821
Verizon PAC	AT&T Federal PAC	.588	.656	.788
⋮	⋮	⋮	⋮	⋮
Johnson & Johnson PAC	Pfizer PAC	.375	.415	.794

and cannot be used as dictionary keys. Dictionary keys are stored in semi-permanent memory locations—the location does not change as long as the key-value pair is in the dictionary. If the key were to change, then the amount of memory needed to contain the key might change, and a different location would be needed. Semi-permanent memory locations allows the dictionary to be optimized and hence, operations on dictionaries are fast. Fast operations are important when working with large dictionaries and why dictionaries are used at every opportunity in this text.

2.10 Exercises

2.10.1 Conceptual

2.1. Show that the number of pairs satisfying $i < j$ that may be formed from $i, j \in \{1, 2, \dots, n\}$ is $\frac{n(n-1}{2}$ by rearranging the expression

$$\sum_{i=1}^{n}\sum_{j=1}^{n} 1 = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} 1 + \sum_{j=1}^{n-1}\sum_{i=j+1}^{n} 1 + \sum_{i=j=1}^{n} 1.$$

Then solve for $\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} 1$.

2.2. a. Give a formula for $C_{n,3}$, the number of sets or combinations that can be formed by choosing three items from n without replication. Recall that the standard convention in mathematics is to disallow an item to appear in a set more than once. For example, $\{a, b, b\}$ is not used since $\{a, b, b\} = \{a, b\}$. Furthermore, the order of arranging the elements of a set does not change the set since $\{a, b, c\} = \{c, a, b\}$. Therefore, there's only one set that consists of the items a, b, c. Compute $C_{5,3}$ and $C_{100,3}$.

b. Using list comprehension, generate all three-tuples (i, j, k) such that $0 \leq i < j < k < 5$.

c. Using set comprehension, generate all sets of three elements from the set $\{0, 1, 2, 3, 4\}$.

2.3. Consider the list constructed using the list comprehension

```
lst1 = [{(i, j),(j,i)} for i in range(4) for j in range(i, 4, 1)]
```

a. Describe the structure and contents of `lst1`.

b. Explain why the set contains a single pair when $i = j$.

c. Consider the list created from the instruction

```
  lst2 = [{(i, j),(j,i)} for i in range(4)
           for j in range(i, 4, 1) if i !=j]
```

Explain why `lst1` $\neq$ `lst2`.

2.4. Initialize a `Python` list to be `L = [(1,2,3),(4,1,5),(0,0,6)]`. Give the `Python` instructions for sorting `L` according to the first coordinate (or position) of the three-tuples. Give the `Python` instructions for sorting `L` according to the third coordinate of the three-tuples. Give the instruction for sorting in descending order (largest to smallest) using the second coordinate of the three-tuples.

2.10.2 Computational

2.5. This exercise is aimed at determining where political action committee (PAC) money goes. Usually, a PAC collects donations from individual contributors by mail or online solicitation and then donates money to candidates of their choice or to other PACs. Sometimes, the intentions of a PAC may not be transparent, and their donations to other PACs or candidates might not be in accord with the intention of the contributors. For this reason, there is interest in tracking the donations of PACs. Table 2.3 is an example showing a few of the PACs that received donations from the Bachmann for Congress PAC during the 2012 election cycle.

Choose an election cycle, identify a PAC of interest, and create a recipient dictionary listing all of the contributions that were received from the PAC. A convenient format for the recipient dictionary uses the recipient committee code as a key. The value should be a two-element list containing the total

Table 2.3 The top eight recipients of contributions from the Bachmann for Congress PAC during the 2010–2012 election cycle

Recipient	Amount ($)
National Republican Congressional Committee	115,000
Republican Party Of Minnesota	41,500
Susan B Anthony List Inc. Candidate Fund	12,166
Freedom Club Federal Pac	10,000
Citizens United Political Victory Fund	10,000
Republican National Coalition For Life Political Action Committee	10,000
Koch Industries Inc Political Action Committee (Kochpac)	10,000
American Crystal Sugar Company Political Action Committee	10,000

amount received by the recipient PAC and the name of the recipient PAC. PACs also contribute to individual candidates, so if you wanted, you could also track money that was received by individual candidates from the PAC of interest.

A couple of reminders may help:

1. Contributions made by PACs are listed in the FEC data files named *Any Transaction from One Committee to Another* and contained in zip files with the name `oth.zip`.
2. To convert a dictionary `C` into a sorted list named `sC`, largest to smallest, by sorting on the values, use

```
sC = sorted(C.iteritems(), key=operator.itemgetter(1),
    reverse = True)
```

   This will work if the value is a list and the first element of the lists are numeric.

2.6. Use the `timeit` module and function to compare execution times for two algorithms used for constructing sets of pairs $\{(i, j)|0 \leq i < j \leq n,$ for integers i and $j\}$. The first algorithm should use list comprehension and the second should join each pair to the set of pairs using the `union` operator. Set $n \in \{100, 200, 300\}$. Report on the computational time for both algorithms and choice of n. Comment on the differences.

2.7. The Consumer Financial Protection Bureau is a federal agency tasked with enforcing federal consumer financial laws and protecting consumers of financial services and products from malfeasance on the part of the providers of these services and products. The Consumer Financial Protection Bureau maintains a database documenting consumer complaints. Download the database from
http://www.consumerfinance.gov/complaintdatabase/#download-the-data.
Determine which ten companies were most often named in the product category *Mortgage* and the issue category of *Loan modification,collection, foreclosure*. The first record of the data file is a list of attributes.

Chapter 3
Scalable Algorithms and Associative Statistics

Abstract It's not uncommon that a single computer is inadequate to handle a massively large data set. The common problems are that it takes too long to process the data and the data volume exceeds the storage capacity of the host. Cleverly designed algorithms sometimes can reduce the processing time to an acceptable point, but the single host solution will eventually fail if data volume is sufficiently great. A far-reaching solution to the data volume problem replaces the single host with a network of computers across which the data are distributed and processed. However, the hardware solution is incomplete until the data processing algorithms are adapted to the distributed computing environment. A complete solution requires algorithms that are *scalable.* Scalability depends on the statistics that are being computed by the algorithm, and the statistics that allow for scalability are *associative* statistics. Scalability and associative statistics are the subject of this chapter.

3.1 Introduction

Suppose that the data set of concern is so massively large in volume that it must be divided and processed as subsets by separate host computers. In this situation, the host computers are connected by a network. A host computer is often called a *node* in recognition of its role as member of the network. Since the computational workload has been distributed across the network, the scheme is referred to as a distributed computing environment. Our interest lies in the algorithmic component of the distributed computing solution, and specifically the situation in which each node executes the same algorithm but on a subset of the data. When all nodes have completed their computations, the results from each node are combined. This is a good strategy if it works:

B. Steele et al., *Algorithms for Data Science*,
DOI 10.1007/978-3-319-45797-0_3

only one algorithm is needed and everything can be automated. If the nodes are garden-variety computing units, there's no obvious limit to the amount of data that can be handled.

The final result should not depend on how the data is divided into subsets. To be more precise, recall that a partition of a set A is a collection of disjoint subsets $A_1, \ldots, A_n$ such that $A = \cup_i A_i$. Now, consider divisions of the data that are partitions (hence no observation or record is included in more than one data set). An algorithm is *scalable* if the results are identical for all possible partitions of the data. If the scalability condition is met, then only the number of subsets and nodes need be increased if the data volume should increase. The only limitation is hardware and financial cost. On the other hand, if the algorithm yields different results depending on the partition, then we should determine the partition that yields the best solution. The question of what is the best solution is ambiguous without criteria to judge what's best. Intuition argues that the best solution is that obtained from a single execution of the algorithm using all the data. Under that premise, we turn to scalable algorithms.

The term scalable is used because the scale, or volume, of the data does not limit the functionality of the algorithm. Scalable data reduction algorithms have been encountered earlier in the form of data mappings. The uses of data mappings were limited to elementary operations such as computing totals and building lists. To progress to more sophisticated analyses, we need to be able to compute a variety of statistics using scalable algorithms, for example, the least squares estimator of a parameter vector $\boldsymbol{\beta}$ or a correlation matrix.

Not all statistics can be computed using a scalable algorithm. Statistics that can be computed using scalable algorithms are referred to as *associative.* The defining characteristic of an associative statistic is that when a data set is partitioned into a collection of disjoint subsets and the associative statistic is computed from each subset, the statistics can be combined or aggregated to yield the same value as would be obtained from the complete data set. If the function of the algorithm is to compute an associative statistic, then the algorithm is scalable.

To make these ideas more concrete, the next section discusses an example involving the Centers for Disease Control and Prevention's BRFSS data sets. Then, we discuss scalable algorithms for descriptive analytics. A tutorial guides the reader through the mechanics of summarizing the distribution of a quantitative variable via a scalable algorithm. Scalable algorithms for computing correlation matrices and the least squares estimator of the linear regression parameter vector $\boldsymbol{\beta}$ are the subject of two additional tutorials. This first look at predictive analytics demonstrates the importance of scalable algorithms and associative statistics in data science.

3.2 Example: Obesity in the United States

Mokdad et al. [40] describe an apparent rapid increase in the number of obese adults in the United States between 1990 and 1999. Similar trends have been observed by other researchers. The phenomenon has been labeled with the sobriquet *the obesity epidemic* in light of the apparent rapid increase in obesity rate and also because of negative effects of obesity. Most prominently, obesity is associated with type 2 diabetes, a chronic disease responsible for a number of conditions that negatively impact quality of life.

Mokdad et al. conducted their analysis using data collected by the U.S. Centers for Disease Control and Prevention.[1] The CDC's Behavioral Risk Factor Surveillance System (BRFSS) survey is now the largest periodic sample survey in the world. The CDC asks a sample of U.S. adult residents a large number of questions regarding health and health-related behaviors. In recent years, responses from more than 400,000 adults have been collected per annum. In the following tutorial, the reader will investigate whether the increasing trend in obesity observed in the last decade of the twentieth century has continued into the first decade of the twenty-first century.

It's common that the initial steps of data analysis involve computing measures of the center and spread of a distribution of a quantitative variable. If a variable of interest is categorical, then a numerical summarization of the data involves computing the proportions of observational units that possess a particular attribute or fall into a particular category. For instance, body mass index (kg/m^2) is a widely-used quantitative measure of body fat from which a categorical measure of obesity (obese or not) can be computed. A first analysis of the BRFSS data might then compare the estimated mean of body mass index and the proportions of obese residents for the year 2000 to the same statistics for the year 2010 to search for evidence supporting or refuting a continuation of the purported obesity epidemic. From a statistical perspective, we aim to estimate the body mass index mean μ of the population of U.S. adult residents at two points in time. More information can be gleaned about body mass index by computing and constructing a histogram showing the estimated proportion of the population with values falling in a set of intervals.

The first algorithmic task is to develop a scalable algorithm for computing estimates of μ and σ^2 using massively large data. Then, we'll develop a scalable algorithm for computing histograms and apply it to the BRFSS data sets and the body mass index variable.

[1] We discussed the BRFSS data briefly in Chap. 1, Sect. 1.2.

3.3 Associative Statistics

We begin with notation and terminology. Let $D = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$ denote a set of n observations. The ith observation $\mathbf{x}_i = [x_{i,1} \;\; x_{i,2} \;\; \cdots \;\; x_{i,p}]^T$ is a vector consisting of p real numbers. A *partition* of D is a collection of disjoint subsets $D_1, D_2, \ldots, D_r$ such that $D = D_1 \cup D_2 \cdots \cup D_r$. Let

$$\underset{d\times 1}{\mathbf{s}(D)} = [s_1(D) \;\; s_2(D) \;\; \cdots \;\; s_d(D)]^T$$

denote an associative statistic, a vector, of dimension $d \geq 1$. We say that a statistic is associative if it possesses an associativity property and is low-dimensional. We use the term *low-dimensional* informally to describe a statistic that may be stored without taxing computational resources. In a practical sense, associativity implies that the data set can be partitioned as r subsets and the statistic can be computed on each subset. At the completion of all r computations, the r associative statistics can be aggregated to obtain the value of the statistic computed on the complete data set. Consequently, there is no information loss or indeterminacy resulting from distributed processing of the data, and algorithms that compute associative statistics are scalable.

Concisely, a statistic $\mathbf{s}$ is associative if, given a partition $\{D_1, D_2, \ldots, D_r\}$ of D, the statistics $\mathbf{s}(D_1)$, $\mathbf{s}(D_2)$, …, $\mathbf{s}(D_r)$ can be combined to produce the value $\mathbf{s}(D)$. An example of an associative statistic computed from a set of n real numbers, say $D = \{x_1, x_2, \ldots, x_n\}$, is the two-element vector

$$\underset{2\times 1}{\mathbf{s}(D)} = \begin{bmatrix} \sum_{i=1}^n x_i \\ n \end{bmatrix}.$$

The elements of $\mathbf{s}(D)$ are $s_1(D) = \sum_{i=1}^n x_i$, and $s_2(D) = n$. Associativity holds because $\mathbf{s}(D_1 \cup D_2) = \mathbf{s}(D_1) + \mathbf{s}(D_2)$. For example, suppose that D is partitioned as $D_1 = \{x_1, \ldots, x_m\}$ and $D_2 = \{x_{m+1}, \ldots, x_n\}$. Then,

$$\begin{aligned} \mathbf{s}(D_1) + \mathbf{s}(D_2) &= \begin{bmatrix} \sum_{i=1}^m x_i \\ m \end{bmatrix} + \begin{bmatrix} \sum_{i=m+1}^n x_i \\ n-m \end{bmatrix} \\ &= \begin{bmatrix} \sum_{i=1}^m x_i + \sum_{i=m+1}^n x_i \\ m+n-m \end{bmatrix} \\ &= \begin{bmatrix} \sum_{i=1}^n x_i \\ n \end{bmatrix} = \mathbf{s}(D). \end{aligned}$$

From $\mathbf{s}(D)$, we can estimate the population mean using the sample mean $\widehat{\mu} = s_1/s_2$. The median is an example of a statistic that is not associative. For example, if $D = \{1, 2, 3, 4, 100\}$, $D_1 = \{1, 2, 3, 4\}$, and $D_2 = \{100\}$, then $\text{median}(D_1) = 2.5$, and there is no method of combining $\text{median}(D_1)$ and $\text{median}(D_2) = 100$ to arrive at $\text{median}(D) = 3$ that succeeds in the general case.

3.4 Univariate Observations

Usually, a preliminary objective of data analysis is to describe the center and spread of the distribution of a variable. This may be accomplished by estimating the population mean μ and variance σ^2 from a sample of univariate data $D = \{x_1, x_2, \ldots, x_n\}$. An estimator of the variance σ^2 is needed in addition to the estimator of the mean $\widehat{\mu} = \sum x_i/n$, say, the average squared difference between the observations and the sample mean:

$$\begin{aligned}\widehat{\sigma}^2 &= n^{-1}\sum (x_i - \widehat{\mu})^2 \\ &= n^{-1}\sum x_i^2 - \left(n^{-1}\sum x_i\right)^2 .\end{aligned} \tag{3.1}$$

From Eq. (3.1), we may deduce that the associative statistic

$$\underset{3\times 1}{\mathbf{s}(D)} = \begin{bmatrix} \sum_{i=1}^n x_i \\ \sum_{i=1}^n x_i^2 \\ n \end{bmatrix} \tag{3.2}$$

may be used to estimate the mean and variance. Let $\mathbf{s}(D) = [s_1 \quad s_2 \quad s_3]^T$. Then, the estimators are

$$\begin{aligned}\widehat{\mu} &= \frac{s_1}{s_3}, \\ \widehat{\sigma}^2 &= \frac{s_2}{s_3} - \left(\frac{s_1}{s_3}\right)^2 .\end{aligned} \tag{3.3}$$

The statistic $\mathbf{s}(D)$ shown in Eq. (3.2) is associative because addition is associative; for example, $\sum_{i=1}^n x_i = \sum_{i=1}^m x_i + \sum_{i=m+1}^n x_i$ for integers $1 \leq m < n$. Thus, a scalable algorithm for computing estimators of the mean and variance computes the associative statistic $\mathbf{s}(D)$ and then the estimates according to Eq. (3.3). If the volume of D is too large for a single host, then D can be partitioned as $D_1, \ldots, D_r$ and the subsets distributed to r network nodes. At node j, $\mathbf{s}(D_j)$ is computed. When all nodes have completed their respective tasks and returned their statistic to a single host, we compute $\mathbf{s}(D) = \sum_j^r \mathbf{s}(D_j)$ followed by $\widehat{\mu}$ and $\widehat{\sigma}^2$ using formula (3.3).

Let's consider a common situation. The data set D is too large to be stored in memory but it's not so large that it cannot be stored on a single computer. These data may be a complete set or one subset of a larger partitioned set. In any case, the data at hand D is still too large to be read at once into memory. The memory problem may be solved using two algorithmic approaches sufficiently general to be applied to a wide range of problems. For simplicity and concreteness, we describe the algorithms for computing the associative statistic given in Eq. (3.2).

A *block* is a subset of D consisting of contiguous lines or records that is sufficiently small to reside in memory. Suppose the blocks are $D_1, D_2, \ldots, D_r$ and D_j contains n_j observations. Then, we may write

$$D = \{\underbrace{x_1, \ldots, x_{n_1}}_{D_1}, \underbrace{x_{n_1+1}, \ldots, x_{n_1+n_2}}_{D_2}, \ldots, \underbrace{x_{n_1+\cdots+n_{r-1}+1}, \ldots, x_{n_1+\cdots+n_r}}_{D_r}\}. \tag{3.4}$$

The blocks form a partition of D because every observation belongs to exactly one subset. The algorithms are:

1. Process D one block at a time. Compute $\mathbf{s}(D_j)$, $j = 1, 2, \ldots, r$. At the completion of processing D_j, store $\mathbf{s}(D_j)$ in a dictionary using j as the key and $\mathbf{s}(D_j)$ as the value. Alternatively, store $\mathbf{s}(D_j)$ as row j of a $r \times 3$ matrix. At completion, compute $\mathbf{s}(D_1) + \mathbf{s}(D_2) + \cdots + \mathbf{s}(D_r) = \mathbf{s}(D)$.
2. A slightly simpler algorithm builds $\mathbf{s}(D)$ as each line is read by updating $\mathbf{s}$. Before processing the data, initialize the associative statistic $\mathbf{s} = [s_1 \;\; s_2 \;\; s_3]^T$ as a vector of zeros. The initialization step is written in short as $[0 \;\; 0 \;\; 0]^T \to \mathbf{s}$ where $a \to b$ means assign a to b. Upon reading the jth observation $x_j \in D$, update $\mathbf{s}$ by computing

$$\begin{aligned} s_1 + x_j &\to s_1 \\ s_2 + x_j^2 &\to s_2 \\ s_3 + 1 &\to s_3. \end{aligned} \tag{3.5}$$

   This one-line-at-a-time algorithm is a scalable algorithm. We can see this by partitioning D as n singleton sets or blocks $D_j = \{x_j\}, j = 1, 2, \ldots, n$. The statistic $\mathbf{s}$ applied to a datum x_j is $\mathbf{s}(\{x_j\}) = [x_j \;\; x_j^2 \;\; 1]^T$. Therefore, the process of updating $\mathbf{s}(\{x_1, x_2, \ldots, x_{j-1}\})$ computes

$$\begin{aligned} \mathbf{s}(\{x_1, x_2, \ldots, x_{j-1}\}) + \mathbf{s}(\{x_j\}) &= \mathbf{s}\left(\cup_{i=1}^{j-1} D_i\right) + \mathbf{s}(D_j) \\ &= \mathbf{s}\left(\cup_{i=1}^{j} D_i\right). \end{aligned}$$

   The sets D_j are not created in practical applications, we just update the associative statistic as shown in formula (3.5). If the algorithm is to be used repeatedly, be aware that reading a data set one line at-a-time *may* be slower than reading the data file in blocks. In our experience, one line at-a-time reading is not noticeably slower using `Python`.

The next section provides an example of a statistic commonly used in exploratory data analysis: the histogram. It's often not realized as such, but a histogram is a statistic that is rendered and interpreted visually. Moreover, the histogram can be cast as an associative statistic and so it is useful in the analysis of massively large data sets.

3.4.1 Histograms

A thorough analysis of a variable often involves a visual display of its empirical, or sample, distribution. The term *empirical* is used to distinguish a distribution constructed from a sample versus the true distribution that would be obtained if all values across a population were observed. Histograms and the related boxplots are commonly used for this purpose. When data volume is large, then the histogram becomes the visual of choice because the distribution may be shown in fine detail. The level of detail is usually under the control of the analyst. Visually, a histogram shows the empirical distribution as a set of contiguous rectangles plotted across the range of the variable. The height of a rectangle is proportional to the frequency (and relative frequency) of values falling within the interval defined by the rectangle base.

Figure 3.1 shows a pair of overlain histograms built from sample distributions of body mass index (kg/m^2) of U.S. residents. Body mass index is body weight scaled by the individual's height. Scaling accounts for height differences between individuals and allows for comparisons of weight regardless of height. The histograms were constructed using data collected in the course of the U.S. Centers for Disease Control and Prevention's Behavioral Risk Factor Surveillance System survey (see Sect. 1.2 for more details). The red histogram was constructed from 1,020,126 observations collected in the years 2000, 2001, 2002, and 2003 and the blue histogram was collected from 2,848,199 observations collected in the years 2011, 2012, 2013, and 2014. The histograms may be used to examine the assertion that the U.S. has suffered an epidemic of obesity [26]. Since an individual is considered to be obese if their body mass index is at least 30 kg/m^2, Fig. 3.1 provides visual confirmation that the percent of obese adults has increased. The percent of obese individuals in the year 2000 data set was 19.84% whereas 28.74% of the respondents in the 2014 data set had a body mass index exceeding the threshold. The increase in the percent obese was $44.8\% = 100(28.74 - 19.84)/19.84$.

The histograms provide more information, though. There's been a shift toward larger values of body mass index from the first to the second time period. Otherwise, the distributions are similar in shape. Both histograms are slightly skewed to the right having longer right tails than left tails. Skewness to the right implies that there are relatively more individuals with very large values of body mass index than individuals with very small body mass index.[2]

[2] Right-skewness is common when a variable is bounded below as is the case with body mass index since no one may have a body mass index less than or equal to zero.

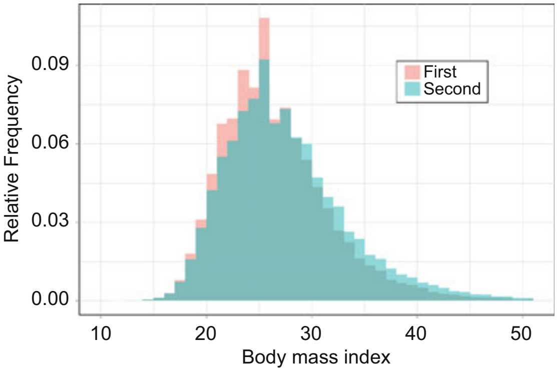


Fig. 3.1 Histograms of body mass index constructed from two samples of U.S. residents. The first sample was collected in the years 2000 through 2003 and the second sample was collected in the years 2011 through 2014. All data were collected by the U.S. Centers for Disease Control and Prevention, Behavioral Risk Factor Surveillance System

3.4.2 Histogram Construction

A histogram is a set of pairs. Each pair corresponds to one of the rectangles that form the visual histogram. One element of the pair is the interval that defines the base of the rectangle and the second element specifies the height. The height is either the number of observations that fall in the interval or the relative frequency of observations falling into the interval. The union of the intervals span the observed range of a variable of interest. Accordingly, we define a histogram (mathematically) as the set of pairs

$$H = \{(b_1, p_1), \ldots, (b_h, p_h)\}. \tag{3.6}$$

The number of intervals is h. The first interval is $b_1 = [l_1, u_1]$ and for $i > 1$, the ith interval is $b_i = (l_i, u_i]$. The second element of the pair, p_i, is the relative frequency of observations belonging to the interval. The term p_i is often used as an estimator of the proportion of the population belonging to b_i. The interval b_{i+1} takes its lower bound from the previous upper bound, i.e., $l_{i+1} = u_i$. We show the intervals as open on the left and closed on the right but there's no reason not to define the intervals to be closed on the left and open on the right instead. In any case, equal-length intervals are formed by dividing the range into h segments. Usually, every observation in the data set belongs to the base of the histogram, $[l_1, u_h]$. Admittedly, the stipulation that all observations are included in the histogram sometimes is a disadvantage as it may make it difficult to see some of the finer details.

Suppose that the data set D is massively large and cannot reside in memory. A scalable algorithm is needed to construct the histogram H, and formulating the algorithm requires a precise description of the process of building H. In brief, H is computed by counting the numbers of observations falling in each interval. The intervals must be constructed before counting, and hence, the data set must be processed twice, both times using a scalable algorithm. The first pass determines the number and width of the intervals. The second pass computes the numbers of observations each interval.

An algorithm for constructing a histogram begins with determining the smallest and largest observations in D. Let $x_{[1]} = \min(D)$ denote the smallest observation and $x_{[n]} = \max(D)$ denote the largest. Every observation in the data set must be examined to determine $x_{[1]}$ and $x_{[n]}$, and so the first pass through the data does nothing more than compute $x_{[1]}$ and $x_{[n]}$. Then, the range is computed as $x_{[n]} - x_{[1]}$. The interval width is $w = (x_{[n]} - x_{[1]})/h$, where h is a choice for the number of intervals. The intervals are

$$\begin{aligned} b_1 &= \left[x_{[1]}, x_{[1]} + w\right] \\ &\vdots \\ b_i &= \left(x_{[1]} + (i-1)w, x_{[1]} + iw\right] \\ &\vdots \\ b_h &= \left(x_{[1]} + (h-1)w, x_{[n]}\right] . \end{aligned}$$

In essence, the algorithm maps the data set D to a set of intervals, or bins, $B = \{b_1, \ldots, b_h\}$ and we may write $D \mapsto B$.

The second algorithm maps D and B to a dictionary C in which the keys are the intervals and the values are the counts of observations falling into a particular interval, and we may write $(D, B) \mapsto C$.

Computationally, the second algorithm fills the dictionary C by counting the number of observations belonging to each interval. Determining which interval an observation belongs to amounts to testing whether the observation value belongs to b_i, for $i = 1, \ldots, h$.[3] When the interval is found that contains the observation, the count of observations belonging to the interval is incremented by one, and the next observation is processed. After all observations are processed, the relative frequency of observations belonging to each interval is computed by dividing the interval count by n. The dictionary may then be rendered as a visual.

If the algorithm is to be scalable, then the statistics from which H is built must be associative. The key to associativity and scalability is that a single set of intervals is used to form the histogram base.

As described above, the first pass through D computes the two-element vector $\mathbf{s}(D) = [\min(D) \;\; \max(D)]^T$. We'll argue that $\mathbf{s}(D)$ is associative by supposing that $D_1, D_2, \ldots, D_r$ are a partition of D. Let

$$\mathbf{s}(D_j) = \begin{bmatrix} \min(D_j) \\ \max(D_j) \end{bmatrix}, j = 1, \ldots, r. \tag{3.7}$$

Let $s_1(D) = \min(D)$ and $s_2(D) = \max(D)$ so that $\mathbf{s}(D) = [s_1(D) \;\; s_2(D)]^T$. Then,

[3] Of course, once it's been determined that the observation belongs to an interval, there's no need to test any other intervals.

$$\begin{aligned} s_1(D) &= \min(D) \\ &= \min(D_1 \cup \cdots \cup D_r) \\ &= \min\{\min(D_1), \ldots, \min(D_r)\} \\ &= \min\{s_1(D_1), \ldots, s_1(D_r)\}. \end{aligned} \tag{3.8}$$

Similarly, $\max(D) = \max\{\max(D_1), \ldots, \max(D_r)\}$. Changing the notation, we see that $s_2(D) = \max\{s_2(D_1),\ldots, s_2(D_r)\}$. Since $\mathbf{s}(D)$ can be computed from $\mathbf{s}(D_1)$, …, $\mathbf{s}(D_r)$, the statistic $\mathbf{s}$ is associative.

The range and intervals of the histogram depend entirely on the associative statistic $\mathbf{s}$ and the choice of h. Given h, the same set of intervals is created no matter how D is partitioned. Thus, a scalable algorithm that computes B is feasible.

The second algorithm fills the dictionary $C = \{(b_1, c_1), \ldots, (b_h, c_h)\}$ by determining c_j, the count of observations in the data set D that belong to b_j, for $j = 1, \ldots, h$. If D has been partitioned as r subsets, then B and a partition D_j are used to construct the jth set of counts. Mathematically, $(D_j, B) \mapsto C_j$. Then, the sets $C_1, \ldots, C_r$ are aggregated by adding the counts across sets for each interval.

3.5 Functions

The next tutorial instructs the reader to create a user-defined function. Reasons for using functions are to make a program easier to understand and to avoid repeatedly programming the same operation. A function consists of a code segment that is executed by a one line instruction such as `y = f(x)`. The code segment should have a clearly defined purpose and generally, the code will be used more than once, say, in more than one location in a program, or in multiple programs, or in a `for` loop.

The function definition is located not within the main program but at the top of the program or in a separate file. However, it's useful to write the code that belongs in a function in the main program. Writing the code in the location where the function will be called allows the programmer to access the values of the variables within the function code for debugging. Once the code segment is included in a function, the variables become local and are undefined outside the function and cannot be inspected. When the code segment executes correctly, then move the code out of the main program and into a function.

The function is initialized by the keyword `def` and is ended (usually) by the `return` statement. For example, the function `product` computes the product of the two arguments `x` and `y` passed to the function:

```
def product(x,y):
    xy = x*y
    return xy
```

The function is called using the statement `z = product(a,b)`. The variable `xy` is local to the function and referencing it outside the program will raise a `NameError`. A return statement is not necessary; for example, another version of the function `product` prints the product of `x` and `y`:

```
def product(x,y):
    print(x*y)
```

Note that `xy` will be undefined outside of the function because it's not returned.

The function must be compiled by the `Python` interpreter before the function is called. Therefore, it should be positioned at the top of the file; alternatively, put it in a file with a `py` extension, say `functions.py`, that contains a collection of functions. The function `product` is imported in the main program using the instruction

```
from functions import product
```

3.6 Tutorial: Histogram Construction

The objective of this tutorial ostensibly is to construct Fig. 3.1. In the process of building the histograms, the reader will estimate the distribution of body mass index of adult U.S. residents for two periods separated by approximately 10 years. The tutorial also expands on the use of dictionaries for data reduction and exposes the reader to weighted means. The data sets used in the tutorial originate from the Centers for Disease Control and Prevention (CDC) and are remarkable for the amount of largely untapped information contained within.

The data were collected in the course of the Behavioral Risk Factor Surveillance System's (BRFSS) annual survey of U.S. adults. Telephone interviews of a sample of adults were conducted in which a wide range of questions on health and health-related behavior were asked of the respondents. The respondents were selected by sampling land-line and cell phone numbers, calling the numbers, and asking for interviews. Land-line and cell phone numbers were sampled using different protocols and so the sampled individuals did not have the same probability of being sampled. Consequently, certain sub-populations, say young land-line owners may be over- or under-represented in the sample. Disregarding the sampling design may lead to biased estimates of population parameters. To adjust for unequal sampling probabilities, the CDC has attached a sampling weight to each observation. The sampling weights may be incorporated into an estimator to correct for unequal sampling probabilities, a subject which we expand on in Chap. 7. A brief discussion is in order though.

First, consider estimation of a population mean μ. An estimator of μ can be expressed as

$$\widehat{\mu} = \textstyle\sum_{j=1}^{n} w_j x_j, \tag{3.9}$$

where w_j is a weight reflecting the contribution of x_j towards the estimator. The traditional sample mean is a weighted mean—we see this if w_j is defined to be $w_j = 1/n$ for each $j = 1, \ldots, n$. Of course, every observation has the same weight and contribution towards the estimate. Any set of weights may be used provided that all weights are non-negative and sum to one. Often, weights are used to reduce the bias of an estimator.

The sample proportion is also a sample mean provided that the variable being averaged is an indicator variable. The indicator variable identifies whether or not the jth sampling unit possesses a particular attribute. For instance, we may define $I_{\geq 30}(x_j)$ as an indicator variable identifying obese individuals. Obesity is defined by a body mass index of 30 kg/m^2 or more. Thus, the indicator variable is

$$I_{\geq 30}(x) = \begin{cases} 1, & \text{if } x \geq 30, \\ 0, & \text{if } x < 30, \end{cases} \tag{3.10}$$

where x is the body mass index of the individual. We say that I is an indicator variable for the event $\{x \geq 30\}$. The sample mean of $I_{\geq 30}(x_1), \ldots, I_{\geq 30}(x_n)$ is the proportion of individuals in the sample that are obese.

In the context of histograms, the property of interest is whether or not a body mass index value x_j is contained in interval b_i. The proportion of the population with membership in b_i is estimated by the sample mean of the indicator variables, say

$$p_i = n^{-1} \sum_{j=1}^{n} I_i(x_j), \tag{3.11}$$

where $I_i(x)$ is an indicator variable of the event $\{x \in b_i\}$. Since we want to use sampling weights with the BRFSS data, we define the estimator of the proportion of the population with membership in b_i as $p_i = \sum_{j=1}^{n} w_j I_i(x_j)$ assuming that the weights are non-negative and sum to one. If the weights are non-negative but do not sum to one, then we may scale the weights by computing

$$p_i = \frac{\sum_j w_j I_i(x_j)}{\sum_j w_j}, \tag{3.12}$$

which amounts to using a set of weights $v_1 = w_1/\sum w_j, \ldots, v_n = w_n/\sum w_j$ that sum to one.

Since our aim is to compare adult distributions of body mass index from two decades, we use 4 years of data from each decade and thus eight data files in the analysis.

1. Create a directory for storing the data files. We recommend that you keep the files related to this book in a single directory with sub-directories for each chapter. Some data files and some functions will be used in more than one chapter so it's convenient to build a sub-directory to save these data files and a sub-directory to contain the functions.[4] The following directory structure is suggested:

```
Algorithms
    DataMaps
        PythonScripts
        RScripts
        Data
    ScalableAlgorithms
        PythonScripts
        RScripts
        Data
    Data
        LLCP2014.ASC
        LLCP2013.ASC
    ModuleDir
        functions.py
```

   The directory `ModuleDir` is to contain user-written modules. A module is a collection of functions that may be loaded and used by any `Python` script. Using functions and modules promotes organized code and reduces the likelihood that you'll write code to perform the same operations more than once.

2. Navigate to the Behavioral Risk Factor Surveillance System data portal http://www.cdc.gov/brfss/annual_data/annual_data.htm. Retrieve the files listed in Table 3.1 and place them in the data directory created in instruction 1. Extract or decompress each file and delete the zip files.

Table 3.1 BRFSS data file names and sub-string positions of body mass index, sampling weight, and gender. Positions are one-indexed

		Body mass index		Sampling weight		Gender
Year	File	Start	End	Start	End	field
2000	cdbrfs00asc.ZIP	862	864	832	841	174
2001	cdbrfs01asc.ZIP	725	730	686	695	139
2002	cdbrfs02asc.ZIP	933	936	822	831	149
2003	CDBRFS03.ZIP	854	857	745	754	161
2011	LLCP2011.ZIP	1533	1536	1475	1484	151
2012	LLCP2012.ZIP	1644	1647	1449	1458	141
2013	LLCP2013.ZIP	2192	2195	1953	1962	178
2014	LLCP2014.ZIP	2247	2250	2007	2016	178

[4] Chapter 7 uses these BRFSS data files in all of the tutorials.

3. If you want to look at the contents of one of the data files, open a Linux terminal and submit a command of the form:

```
cat LLCP2011.ASC | more
```

   Each record in a BRFSS data file is a character string and without delimiters to identify specific variables. The file format is *fixed-width*, implying that variables are located according to an established and unchanging position in the string (recall that the record is a character string). Consequently, variables are extracted as substrings from each record. Regrettably, string or *field* position depends on year and the field positions must be determined anew each time a different year is processed. Table 3.1 contains the field positions for several variables. The field positions are exactly as presented in the BRFSS codebooks. The codebooks describe the variables and field positions. For example, https://www.cdc.gov/brfss/annual_data/2014/pdf/codebook14_llcp.pdf is the year 2014 codebook.

   Table 3.1 field positions are one-indexed. When one-indexing is used, the first character in the string `s` is `s[1]`. `Python` uses zero-indexing to reference characters in a string[5] so we will have to adjust the values in Table 3.1 accordingly.

4. Create a `Python` script. The first code segment creates a dictionary that contains the field positions of body mass index and sampling weight. We'll create a dictionary of dictionaries. The outer dictionary name is `fieldDict` and the keys of this dictionary are years, though we use only the last two digits of the year rather than all four. The first instruction in the following code segment initializes `fieldDict`. The keys are defined when the dictionary is initialized.

   The values of `fieldDict` are dictionaries in which the keys are the variable names and the values of these *inner dictionaries* are pairs identifying the first and last field positions of the variable. The dictionaries for year 2000 (`field[0]`) and 2001 (`field[1]`) are shown in the two lines following the initialization of `fieldDict`:

```
fieldDict = dict.fromkeys([0, 1, 2, 3, 11, 12, 13, 14])
fieldDict[0] = {’bmi’:(862, 864), ’weight’:(832, 841)}
fieldDict[1] = {’bmi’:(725, 730), ’weight’:(686, 695)}
```

5. Using the information in Table 3.1, add the remaining inner dictionaries entries to `fieldDict`, that is, for the years 2002, 2003, 2011, 2012, 2013,

[5] The first character in the string in `Python` is `s[0]`.

and 2014. Check your code by printing the contents of `fieldDict` and comparing to Table 3.1. Iterate over year and print:

```
for year in fieldDict:
    print(year, fieldDict[year])
```

6. We’ll use `fieldDict` in the several other tutorials and add additional variables and years. To keep our scripts short and easy to read, move the code segment that builds `fieldDict` to a function. For the moment, put the function definition at the top of your `Python` script.

```
def fieldDictBuild():
    fieldDict[0] = {’bmi’:(862,864),’weight’:(832,841)}
    ...
    fieldDict[14] = {’bmi’:(2247,2250),’weight’:(2007,2016)}
    return fieldDict
```

7. Call the function with the instruction `fieldDict = fieldDictBuild()`. Print `fieldDict` and check that it is agrees with the entries in Table 3.1.
8. The next two instructions direct the reader on building a module to contain `fieldDictBuild`. Begin by creating a directory named `ModuleDir` and a `Python` script named `functions.py`. Instruction 1 provides a suggestion on the structure and names of the directories. Remove `fieldDict` from its position at the top of the script and put it in `functions.py`. Now `functions.py` is a *module*—a collection of definitions and functions that may be called from other programs. The function `fieldDictBuild` creates `fieldDict` by executing the instruction

```
fieldDict = functions.fieldDictBuild()
```

However, before this function call will execute successfully, the `functions` module must be imported using the instruction

```
from ModuleDir import functions
```

`ModuleDir` is the directory containing the file `functions.py`. The directory may contain other modules that are unrelated to the purposes of `functions`.
9. Lastly, if the directory containing `functions.py` is not the same as the location of the script being executed, then the interpreter must be instructed on where to search for `functions.py`. If this is the case, then direct the compiler where to search for the module. Assuming that the full

path to `functions.py` is `/home/Algorithms/ModuleDir/functions.py`, put the following instruction at the top of the script:

```
import os,sys

parentDir = r'/home/Algorithms/'
if parentDir not in set(sys.path):
    sys.path.append(parentDir)
    print(sys.path)
from ModuleDir import functions
dir(functions)
```

Your path, (`parentDir`), may be different.[6] Notice that the path to the directory omits the name of the directory containing the function. Adding `r` in front of the path instructs `Python` to read backslashes literally. You'll probably need this if you're working in a Windows environment.

When you modify `functions.py`, it will have to be reloaded by the interpreter for the changes to take effect. You have to instruct the interpreter to reload it or else you have to restart the interpreter.[7] You can reload a module using a function from a library. If you're using `Python 3.4` or later, then import the library `importlib` using the instruction `import importlib`. The instructions are

```
import importlib
reload(functions)  # Python 3.4 or above
```

to reload `functions`. If `reload` does not update a change to `functions.py`, then restart the console. Restarting a `Python` console will also (re)load all of the modules. If your version of `Python` is 3.3 or less, then import the library `imp` and reload `functions` with the instructions:

```
import imp
imp.reload(functions)  # Python 2 and 3.2-3.3
```

[6] It will not begin with `/home/...` if your operating system is Windows.

[7] In `Spyder`, close the console, thereby killing the kernel, and start a new console to restart the interpreter.

After reloading, check the contents of the module `functions` with the instruction `dir(functions)`. The `dir(functions)` will list all of the functions that are available including a number of built-in functions.[8]

10. The function `fieldDictBuild` will build `fieldDict` when called so:

```
fieldDict = functions.fieldDictBuild()
```

Add the instruction to the script after the `reload(functions)` instruction.

11. Build a `for` loop that will process all of the files located in the data directory. First, create a list containing the names of the files in the directory. Then iterate over the list and as the iteration progresses, test whether the current file name is one of the BRFSS data files listed in Table 3.1. This is accomplished by extracting the two character substring occupying positions 6 and 7 from the file name. If the two-character string cannot be converted to an integer, then the file name is not one of the BRFSS data files and a `ValueError` will be thrown by the `Python` interpreter. Execute the conversion of string to integer with an exception handler so that any file that is in the directory and is not BRFSS data file will be passed over without causing the program to terminate.

```
path = r’../Data/’  #  Set the path to match your data directory.
fileList = os.listdir(path) # Creates a list of files in path
for filename in fileList:
    try:
        shortYear = int(filename[6:8])
        year = 2000 + shortYear

        fields = functions.fieldDict[shortYear]
        sWt, eWt = fields[’weight’]
        sBMI, eBMI = fields[’bmi’]

        file = path+filename
        print(file,sWt, eWt,sBMI, eBMI)
    except(ValueError, KeyError):
        pass
```

The field positions of the sampling weight and body mass index are extracted in the middle block of three instructions. The first instruction extracts the `fields` dictionary from `fieldDict` using the two-digit year as the key. Then, the starting and ending positions of the variables are extracted. The starting and ending positions are the field positions translated from the BRFSS codebook. The BRFSS codebook for a specific year

[8] Execute `functions.py` if your function in `functions.py` is not compiling despite calling the `reload` function.

can be found on the same webpage as the data file.[9] The codebook lists the field positions using *one-indexing*. One-indexing identifies the first field in the string as column 1. However, `Python` uses zero-indexing for strings and so we will have to adjust when extracting the values.

12. The following code segment processes each data file as the program iterates over the list of files. The code must be nested within the `try` branch of the exception handler (instruction 11). Insert the code segment after the print statement above.

```
with open(file, ’r’, encoding=’latin-1’) as f:
    for record in f:
        print(len(record))
```

In `Python 3`, the instruction `with open` forces the file to close when the `for` loop is interrupted or when the end of the file is reached. We do not need the instruction `f.close()`. Hence, all instructions that are to be carried out while the file is being read must be nested below the `with open` instruction.

13. In the `for` loop of instruction 12, extract body mass index and sampling weight from each record by *slicing*. If a string named `record` is sliced using the instruction `record[a:b]`, then the result is a substring consisting of the items in fields $a, a+1, \ldots, b-1$. Note that the character in field b is not included in the slice. Convert the sampling weight string to a float using the one-index field positions `sWt` and `eWt`. Also extract the string containing the body mass index value using `sBMI` and `eBMI` extracted in instruction 11.

```
weight = float(record[sWt-1:eWt])
bmiString = record[sBMI-1:eBMI]
```

The next code segment converts `bmiString` to a float.

14. The BRFSS format and missing value code for body mass index depends on year so there will be year-specific instructions for translating strings to floats. Further, the decimal point has been left out of the string and has to be inserted. The following code translates `bmiString` to a float value if the missing value code is not encountered. If a missing value code is encountered, then body mass index will be assigned the value 0. Since none of the histogram intervals contain 0, a record with a missing value code will be effectively omitted from the construction of the histograms.

[9] The codebook contains a wealth of information about the data and data file structure.

```
bmi = 0
if shortYear == 0 and bmiString != ’999’:
    bmi = .1*float(bmiString)
if shortYear == 1 and  bmiString != ’999999’:
    bmi = .0001*float(bmiString)
if 2 <= shortYear <= 10 and bmiString != ’9999’:
    bmi = .01*float(bmiString)
if shortYear > 10 and bmiString != ’    ’:
    bmi = .01*float(bmiString)
print(bmiString, bmi)
```

The length of the blank string must be the same as the length of `bmiString`.

15. When the conversion of the string containing body mass index to the decimal expression works correctly, then turn it into a function by placing the declaration

```
def convertBMI(bmiString, shortYear):
```

before the code segment. Indent the code segment and add the instruction `return bmi` at the end of the code segment.

16. Add the instruction to call the function:

```
bmi = convertBMI(bmiString, shortYear)
```

after the function. Run the program. If it is successful, then move the definition of `convertBMI` to `functions.py`. The function will not be available until the `functions` is recompiled, so execute the script `functions.py`. Call the function using the instruction

```
bmi = functions.convertBMI(bmiString, shortYear)
```

17. Go back to the beginning of the program and set up a dictionary to contain the histograms. One histogram will be created for each year, and each histogram will be represented by a dictionary that uses intervals as keys and sums of sampling weights as values. (Ordinarily, the value would be a count of observations that fell between the lower and upper bounds of the interval). Each histogram interval is a tuple in which the tuple elements are the lower and upper bounds of the interval. The value is a float since it is a sum of sampling weights. The set of intervals corresponding to a histogram are created once as a list using list comprehension:

```
intervals = [(10+i, 10+(i+1)) for i in range(65)]
```

Place this instruction before the for loop that iterates over fileList. The first and last keys are (10,11) and (74,75). It will become apparent momentarily that the histogram spans the half-open interval $(10, 75]$ because of the way we test whether an observation falls in an interval. A few individuals have a body mass index outside this range. We will ignore these individuals since there are too few of them to affect the histogram shape and including them interferes with readability.

18. Build a dictionary of histograms in which the keys are years and the values are dictionaries.

```
years = [2000, 2001, 2002, 2003, 2011, 2012, 2013, 2014]
histDict = {}
for year in years:
    histDict[year] = dict.fromkeys(intervals,0)
```

The value associated with the key year is a dictionary. The keys of these inner dictionaries are the histogram intervals for the year. The values of the inner dictionary are initialized as 0.

19. Return to the for loop that processes fileList. We'll fill the histogram corresponding to the data file or equivalently, the year, as the file is processed. The first step is to identify the histogram to be filled by adding the instruction histogram = histDict[year] immediately after extracting the field positions for weight and body mass index (instruction 11).
20. We will assume that a ValueError has not be thrown and thus body mass index and sampling weight have been extracted successfully from the record. Increment the sum of the weights for the histogram interval that contains the value of body mass index. The for loop below iterates over each interval in histogram. The lower bound of the interval is interval[0] and the upper bound is interval[1].

```
for interval in histogram:
    if interval[0] < bmi <= interval[1]:
        histogram[interval] += weight
        break
```

The break instruction terminates the for loop when the correct interval has been found.

This code segment must be located inside the for loop initiated by for record in f so that it executes every time bmiString is converted to the float bmi. Indentation should be the same as the statement if shortYear > 10 and bmiString != ‘ ’:.

When the end of the file has been reached, histogram will contain the sum of the weights shown in the numerator of Eq. (3.12). The dictionary histDict[year] will also have been filled since we set histogram = histDict[year] and the result of this instruction is that histDict[year] and histogram reference the same location in memory. You can test this by executing print(id(histogram), id(histDict[year])). The function id reveals the unique identifier of the object.[10]

21. It may be of interest to count the number of body mass index values that are outside the interval $(10, 75]$. Initialize two variables before the files are processed by setting them equal to zero. Give them the names outCounter and n. Add the following instructions and indent them so that they execute immediately after the code segment shown in instruction 20.

```
n += 1
outCounter += int(bmi < 10 or bmi > 75)
if n % 10000 == 0:
    print(year,n,outCounter)
```

Thus far, the program functions as a mapper algorithm by mapping records to annual histograms. The next code segment functions as a reducer algorithm by mapping annual histograms to decadal histograms. The code segment executes after all the data files have been processed.

22. Initialize a two-element list decadeWts to contain the sampling weight totals for each decade. Create a dictionary named decadeDict with the same structure as histDict except that the keys are decades instead of years. The value associated with a decade key is a histogram dictionary. The keys of the histogram dictionaries are the intervals and the values are sampling weight totals. The code follows:

```
decadeWts = [0]*2
decades = [0, 1]
decadeDict = {}
for decade in decades:
    decadeDict[decade] = dict.fromkeys(intervals, 0)
```

The list intervals was created earlier (instruction 17).

23. Construct a for loop that iterates over the list years and maps the annual histograms to the appropriate decadal histograms:

[10] It's informative to submit the instruction a = b = 1 at the console. Then, submit a = 2 and print the value of b. The moral of this lesson is be careful when you set two variables equal.

```
for year in years:
    decade = int(year/2005)
    histogram = histDict[year]
```

The instruction `decade = int(year/2005)` produces the largest integer less than or equal to 2005/`year`, thereby determining the decade as 0 or 1.

24. As the `for` loop of instruction 23 iterates over year, compute the sum of the sampling weights for the year. Specifically, extract the sampling weight sum associated with each interval and increment the sampling weight sum in the decade dictionary:

```
for interval in histogram:
    weightSum = histogram[interval]
    decadeDict[decade][interval] += weightSum
    decadeWts[decade] += weightSum
```

Since the `for` loop is to execute for each year, it must be aligned with the instruction `histogram = histDict[year].`

25. We may now scale the decade histograms to contain the estimated proportions of the population with body mass index falling in a particular interval.

```
for decade in decadeDict:
    histogram = decadeDict[decade]
    for interval in histogram:
        histogram[interval] = histogram[interval]/decadeWts[decade]
```

Again, we've used the fact that assigning one variable equal to another variable only generates two names for the same variable (or memory location).

26. We will use the `Python` module `matplotlib` to graph the histograms for the two decades. In preparation for plotting, import the plotting function `pyplot` from `matplotlib` and create a list `x` containing the midpoints of the histogram intervals. Also create a list `y` containing the estimated proportions associated with each interval. Exclude from `x` and `y` the intervals beyond 50 since relatively few individuals have a body mass index greater than 50.

```
import matplotlib.pyplot as plt

x = [np.mean(pair) for pair in intervals if pair[0] < 50]
y = [decadeDict[0][pair] for pair in intervals if pair[0] < 50]
plt.plot(x, y)
y = [decadeDict[1][pair] for pair in intervals if pair[0] < 50]
plt.plot(x, y)
plt.legend([str(label) for label in range(2)], loc=’upper right’)
```

Two histograms are plotted on the same plot.

The histograms do not have the traditional rectangular or step form. If you include the argument `drawstyle=’steps’` in the call `plt.plot(x, y)`, then the histogram will appear as a series of steps. (We find that it is not as easy to discriminate between the two histograms if they appear as a series of steps). If the histograms are to appear on separate figures, then set the figure number before the histogram is plotted.

```
x = [np.mean(pair) for pair in intervals if pair[0] < 50]
y = [decadeDict[0][pair] for pair in intervals if pair[0] < 50]
plt.figure(1)
plt.plot(x, y, drawstyle=’steps’)
y = [decadeDict[1][pair] for pair in intervals if pair[0] < 50]
plt.figure(2)
plt.plot(x, y, drawstyle=’steps’)
```

27. There is visual evidence of a difference between decades with respect to the distributions of body mass index but it does not seem to us to be very convincing evidence supporting the assertion that the U.S. is suffering an obesity epidemic. Compute an estimate of the proportion of the populations that are classified as obese, or equivalently, that have a body mass index greater than or equal to 30 kg/m^2 for the two decades. The computation for the first decade is

```
print(sum([decadeDict[0][(a,b)] for a, b in decadeDict[0] if a >= 30]))
```

28. Compute the percent change relative to the earlier decade, say $100(\widehat{\pi}_{2010} - \widehat{\pi}_{2000})/\widehat{\pi}_{2000}\%$. Does this statistic support the assertion that an epidemic is occurring?[11]

[11] We think so.

3.6.1 Synopsis

We have reduced millions of observations on body mass index to a very compact summary—two histograms consisting of the estimated proportion of adult U.S. residents falling into 65 one-unit body mass index intervals. From the two histograms, we have clear, empirical evidence of a shift in body mass index over roughly a 10 year time span.

Often, boxplots are better than histograms when the objective is primarily a visual contrast of two or more distributions. Boxplots are preferable to histograms when the number of observations is small and it's desirable to visually identify outliers. The boxplot statistics are the 25th, 50th and 75th percentiles, collectively referred to as the quartiles. Finding the quartiles requires the data to be ordered, and if the data are partitioned as subsets $D_1, D_2, \ldots, D_r$, and separately processed to compute r sets of quartiles, then there is no method that will aggregate the quartiles from each subset and obtain the exact quartiles of D. Quartiles, and hence, boxplot statistics are not associative. This raises the problem of determining an algorithm for computing the boxplot statistics from a massively large data set—there's no obvious path forward and the problem becomes mired in compromises and approximations. In contrast, the path for building histograms from massively large data sets is straightforward and unambiguous.

3.7 Multivariate Data

In this section the reader is introduced to basic algorithms for multivariate data processing. We continue to develop the theme of scalability and associative statistics. It should be kept in mind that multivariate analysis is a broad area, drawing on a wide variety of methods from statistics and numerical linear algebra. Most the methods are sophisticated both mathematically and computationally and thus beyond the scope of this text. In fact, one cannot really get very far into the subject without some knowledge of matrix algebra. Therefore, we proceed under the assumption that the reader has a familiarity with matrix notation and arithmetic operations involving vectors and matrices. Chapter 1, Sect. 1.10.1 provides a short review.

The data are presumed to be a set of multivariate observations collected by sampling a population or observing a process. An example of a process is the stream of price quotations generated by trading activity on the NASDAQ stock exchange.[12] In any minute of trading, a quotation may be produced on any of the 3100 or so stocks, and if the quotations are aggregated by the minute, then we may think of trading as a process generating a stream of

[12] NASDAQ is the abbreviation for the National Association of Securities Dealers Automated Quotations system.

observations on a 3100-element vector. The elements of each vector are the most recent asking price of the stocks. The analysis of this stream in real-time has become both highly profitable and in some quarters, objectionable. We'll work with a univariate form of the NASDAQ stream and forecasting in Chap. 11. The Behavioral Risk Factor Surveillance System discussed in Sect. 3.6 exemplifies population sampling. From each respondent, the answers to a number of questions are obtained, and so each question is in essence one of many variables. The reader will use these data and the algorithms of this section to investigate relationships between obesity, income, and education in U.S. adults.

More generally, the objectives of multivariate data analysis routinely include estimating the mean level and variance of the variables, computing the sample correlation between variables, and estimating the coefficients of a linear model that predicts one of the variables from one or more of the other variables. Before developing scalable algorithms for accomplishing these objectives, some notation and terminology needs to be established.

3.7.1 Notation and Terminology

Suppose that the data consists of observations on n units and on each, the values of p variables are observed. The data set is denoted as $D = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$ and the ith observation in D is a vector

$$\underset{p\times 1}{\mathbf{x}_i} = [x_{i,1}\ x_{i,2}\ \cdots\ x_{i,p}]^T \tag{3.13}$$

in which the jth element, $x_{i,j}$, is an observation on the jth variable.

From a statistical perspective, the data often are treated as n realizations of a multivariate random vector $\mathbf{X} = [X_1\ X_2\ \cdots\ X_p]^T$ with expectation $\mathrm{E}(\mathbf{X}) = \boldsymbol{\mu}$ and variance matrix $\mathrm{var}(\mathbf{X}) = \boldsymbol{\Sigma}$ where

$$\begin{aligned}\underset{p\times 1}{\boldsymbol{\mu}} &= [\mathrm{E}(X_1)\ \mathrm{E}(X_2)\ \cdots\ \mathrm{E}(X_p)]^T, \\ &= [\mu_1\ \mu_2\ \cdots\ \mu_p]^T.\end{aligned} \tag{3.14}$$

We set $\mu_i = \mathrm{E}(X_i)$. The variance matrix is

$$\underset{p\times p}{\boldsymbol{\Sigma}} = \begin{bmatrix} \sigma_1^2 & \sigma_{12} & \cdots & \sigma_{1p} \\ \sigma_{21} & \sigma_2^2 & \cdots & \sigma_{2p} \\ \vdots & & \ddots & \vdots \\ \sigma_{p1} & \sigma_{p2} & \cdots & \sigma_p^2 \end{bmatrix}.$$

The diagonal elements $\sigma_1^2, \ldots, \sigma_p^2$ are the variances of each univariate variable. The standard deviation $\sigma_j = \mathrm{E}[(X_j - \mu_j)^2]^{1/2}$ may be interpreted as the average (absolute) difference between the mean μ_j and the realized values of the variable. The off-diagonal elements of $\boldsymbol{\Sigma}$ are referred to as the covariances. The principal use of the covariances is to describe the strength and direction of association between two variables. Specifically, the population correlation coefficient

$$\rho_{jk} = \frac{\sigma_{jk}}{\sigma_j \sigma_k} = \rho_{kj} \tag{3.15}$$

quantifies the strength and direction of linear association between the jth and kth random variables. The correlation coefficient is bounded by -1 and 1, and values of ρ_{jk} near 1 or -1 indicate that the two variables are nearly linear functions of each other. Problem 3.6 asks the reader to verify this statement. When ρ_{jk} is positive, the variables are positively associated and as the values of one variable increase, the second also tends to increase. If $\rho_{jk} < 0$, the variables are negatively associated and as the values of one variable increase, the second tends to decrease. Values of ρ_{jk} near 0 indicate that the relationship, if any, is not linear.

The correlation matrix is

$$\underset{p\times p}{\boldsymbol{\rho}} = \begin{bmatrix} 1 & \rho_{12} & \cdots & \rho_{1p} \\ \rho_{21} & 1 & \cdots & \rho_{2p} \\ \vdots & & \ddots & \vdots \\ \rho_{p1} & \rho_{p2} & \cdots & 1 \end{bmatrix}.$$

Together, $\boldsymbol{\mu}$, $\boldsymbol{\Sigma}$, and $\boldsymbol{\rho}$ provide a substantial amount of information about a population or process. However, these quantities rarely can be determined precisely and must be estimated from the data. Multivariate data analysis usually begins with estimation of these parameters.

3.7.2 Estimators

The mean vector $\boldsymbol{\mu}$ is estimated by the sample mean vector

$$\underset{p\times 1}{\overline{\mathbf{x}}} = n^{-1} \begin{bmatrix} \sum_i^n x_{i,1} \\ \sum_i^n x_{i,2} \\ \vdots \\ \sum_i^n x_{i,p} \end{bmatrix}.$$

A scalable algorithm for computing $\overline{\mathbf{x}}$ is straightforward since $\overline{\mathbf{x}}$ is a function of a vector of sums, and the sum is an associative statistic. A scalable algorithm for computing the sample mean vector reads the data in blocks or line-by-line to compute the sums $\sum_i^n x_{i,j}, j = 1, \ldots, p$. It's not so obvious but an estimator of σ_j^2 may also be computed from an associative statistic using a scalable algorithm.

We will not use the traditional moment estimator of σ_i^2 but instead use an estimator that is nearly equivalent and not as messy for developing the estimator as a function of an associative statistic. Specifically, we use n as the denominator instead of the traditional $n-1$. The choice of n simplifies computations but introduces a downward bias in the estimators. When n is not small, but instead is at least 100, the bias is negligible. In any case, the variance estimator is

$$\begin{aligned}\widehat{\sigma}_j^2 &= \frac{\sum_i (x_{i,j}-\overline{x}_j)^2}{n}\\ &= n^{-1}\left(\textstyle\sum_i x_{i,j}^2 - n\overline{x}_j^2\right)\\ &= n^{-1}\textstyle\sum_i x_{i,j}^2 - \overline{x}_j^2.\end{aligned} \tag{3.16}$$

The estimator of the covariance between variables j and k is

$$\begin{aligned}\widehat{\sigma}_{jk} &= \frac{\sum_i (x_{i,j}-\overline{x}_j)(x_{i,k}-\overline{x}_k)}{n}\\ &= n^{-1}\left(\textstyle\sum_i x_{i,j}x_{i,k} - n\overline{x}_j\overline{x}_k\right)\\ &= n^{-1}\textstyle\sum_i x_{i,j}x_{i,k} - \overline{x}_j\overline{x}_k.\end{aligned} \tag{3.17}$$

Our estimator $\widehat{\sigma}_{jk}$ also differs from the traditional moment estimator of σ_{jk} by using n as the denominator in place of $n-1$ for the previous reason—there's no practical difference and the mathematics are simpler.

The variance matrix $\boldsymbol{\Sigma}$ is estimated by

$$\begin{aligned}\widehat{\boldsymbol{\Sigma}} &= \begin{bmatrix}\widehat{\sigma}_1^2 & \cdots & \widehat{\sigma}_{1p}\\ \vdots & \ddots & \vdots\\ \widehat{\sigma}_{p1} & \cdots & \widehat{\sigma}_p^2\end{bmatrix}\\ &= \begin{bmatrix} n^{-1}\sum\limits_i x_{i,1}^2 - \overline{x}_1^2 & \cdots & n^{-1}\sum\limits_i x_{i,1}x_{i,p} - \overline{x}_1\overline{x}_p\\ \vdots & \ddots & \vdots\\ n^{-1}\sum\limits_i x_{i,p}x_{i,1} - \overline{x}_p\overline{x}_1 & \cdots & n^{-1}\sum\limits_i x_{i,p}^2 - \overline{x}_p^2\end{bmatrix}.\end{aligned} \tag{3.18}$$

The outer product[13] of $\overline{\mathbf{x}}$ with itself is a $p \times p$ matrix given by

$$\underset{p\times 1}{\overline{\mathbf{x}}}\,\underset{1\times p}{\overline{\mathbf{x}}}^T = \begin{bmatrix}\overline{x}_1^2 & \cdots & \overline{x}_1\overline{x}_p\\ \vdots & \ddots & \vdots\\ \overline{x}_p\overline{x}_1 & \cdots & \overline{x}_p^2\end{bmatrix}.$$

[13] The *inner product* of a vector $\mathbf{w}$ with itself is the scalar $\mathbf{w}^T\mathbf{w}$ (Chap. 1, Sect. 1.10.1).

The outer product leads to a simpler alternative formula to Eq. (3.18):

$$\widehat{\boldsymbol{\Sigma}} = n^{-1}\mathbf{M} - \overline{\mathbf{x}}\overline{\mathbf{x}}^T, \tag{3.19}$$

where

$$\underset{p\times p}{\mathbf{M}} = \begin{bmatrix} \sum x_{i,1}^2 & \sum x_{i,1}x_{i,2} & \cdots & \sum x_{i,1}x_{i,p} \\ \sum x_{i,2}x_{i,1} & \sum x_{i,2}^2 & \cdots & \sum x_{i,2}x_{i,p} \\ \vdots & \vdots & \ddots & \vdots \\ \sum x_{i,p}x_{i,1} & \sum x_{i,p}x_{i,2} & \cdots & \sum x_{i,p}^2 \end{bmatrix}$$

is the raw or uncentered *moment* matrix. The statistic $\mathbf{M}$ is composed of sums and is associative. Thus, $\mathbf{M}$ and $\widehat{\boldsymbol{\Sigma}}$ can be computed using a scalable algorithm. An algorithm for simultaneously computing $\mathbf{M}$ and $\overline{\mathbf{x}}$ will be described momentarily.

The estimator of $\boldsymbol{\rho}$ is the correlation matrix

$$\mathbf{R} = \begin{bmatrix} 1 & r_{12} & \cdots & r_{1p} \\ r_{21} & 1 & \cdots & r_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ r_{p1} & r_{p2} & \cdots & 1 \end{bmatrix},$$

where

$$r_{jk} = \frac{\sum (x_{i,j} - \overline{x}_j)(x_{i,k} - \overline{x}_k)}{\widehat{\sigma}_j \widehat{\sigma}_k}. \tag{3.20}$$

The matrix $\mathbf{R}$ is symmetric because $r_{jk} = r_{kj}$ for each $1 \le j, k \le p$. Furthermore, $\mathbf{R}$ is a product of two matrices $\mathbf{D}$ and $\widehat{\boldsymbol{\Sigma}}$ given by

$$\mathbf{R} = \mathbf{D}\widehat{\boldsymbol{\Sigma}}\mathbf{D}, \tag{3.21}$$

where $\mathbf{D}$ is the diagonal matrix

$$\mathbf{D} = \begin{bmatrix} \widehat{\sigma}_1^{-1} & 0 & \cdots & 0 \\ 0 & \widehat{\sigma}_2^{-1} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \widehat{\sigma}_p^{-1} \end{bmatrix}.$$

Computing $\mathbf{R}$ is easy using formula (3.21); of course, this depends on the programming language having implemented a matrix product operator.

The computation of $\overline{\mathbf{x}}$, $\widehat{\boldsymbol{\Sigma}}$ and $\mathbf{R}$ is expedited if the data vectors are augmented by concatenating 1 to each vector. The moment matrix obtained from the augmented data vectors is the topic of the next section.

3.7.3 The Augmented Moment Matrix

Each data vector is augmented by concatenating a 1 before the data values. The ith the augmented vector is then

$$\underset{(p+1)\times 1}{\mathbf{w}_i} = \begin{bmatrix} 1 \\ \underset{p\times 1}{\mathbf{x}_i} \end{bmatrix} = \begin{bmatrix} 1 \\ x_{i,1} \\ \vdots \\ x_{i,p} \end{bmatrix}.$$

The *outer product* of $\mathbf{w}_i$ with itself is a $(p+1)\times(p+1)$ matrix

$$\mathbf{w}_i\mathbf{w}_i^T = \begin{bmatrix} 1 \\ x_{i,1} \\ \vdots \\ x_{i,p} \end{bmatrix} \begin{bmatrix} 1 \; x_{i,1} \cdots x_{i,p} \end{bmatrix} = \begin{bmatrix} 1 & x_{i,1} & \cdots & x_{i,p} \\ x_{i,1} & x_{i,1}^2 & \cdots & x_{i,1}x_{i,p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{i,p} & x_{i,p}x_{i,1} & \cdots & x_{i,p}^2 \end{bmatrix}.$$

The sum of n outer products forms the augmented matrix $\mathbf{A}$:

$$\begin{aligned} \underset{(p+1)\times(p+1)}{\mathbf{A}} &= \textstyle\sum_i \mathbf{w}_i\mathbf{w}_i^T \\ &= \begin{bmatrix} n & \sum x_{i,1} & \sum x_{i,2} & \cdots & \sum x_{i,p} \\ \sum x_{i,1} & \sum x_{i,1}^2 & \sum x_{i,1}x_{i,2} & \cdots & \sum x_{i,1}x_{i,p} \\ \sum x_{i,2} & \sum x_{i,2}x_{i,1} & \sum x_{i,2}^2 & \cdots & \sum x_{i,2}x_{i,p} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \sum x_{i,p} & \sum x_{i,p}x_{i,1} & \sum x_{i,p}x_{i,2} & \cdots & \sum x_{i,p}^2 \end{bmatrix}. \end{aligned} \tag{3.22}$$

The *augmented* moment matrix differs from the moment matrix $\mathbf{M}$ by having an additional row along the top and left side that contains n and the sums of each variable. Hence, the augmented matrix may be expressed as

$$\mathbf{A} = \begin{bmatrix} \underset{1\times 1}{n} & \underset{1\times p}{n\overline{\mathbf{x}}^T} \\ \underset{p\times 1}{n\overline{\mathbf{x}}} & \underset{p\times p}{\mathbf{M}} \end{bmatrix}. \tag{3.23}$$

The augmented moment matrix is an associative statistic, and from $\mathbf{A}$, the statistics $\mathbf{M}$ and $n\overline{\mathbf{x}}$ are readily extracted. With $\mathbf{M}$ and $n\overline{\mathbf{x}}$, the variance matrix $\widehat{\boldsymbol{\Sigma}}$ can be computed using formula (3.19). Likewise, $\mathbf{D}$, and $\mathbf{R}$ are easily computed. A computationally efficient algorithm will compute $\mathbf{A}$ by iterating over $\mathbf{x}_1, \ldots, \mathbf{x}_n$. The estimates $\overline{\mathbf{x}}$, $\widehat{\boldsymbol{\Sigma}}$, and $\mathbf{R}$ are computed from $\mathbf{A}$ using matrix operations.

In a distributed environment, the data set D is partitioned as subsets $D_1, D_2, \ldots, D_r$ and each subset is processed separately. Subset D_j yields an augmented matrix $\mathbf{A}_j$, and $\mathbf{A}_1, \ldots, \mathbf{A}_r$ are joined at completion of the r processes as

$$\mathbf{A} = \sum_{j=1}^{r} \mathbf{A}_j.$$

Then, $\mathbf{M}$ and $\overline{\mathbf{x}}$ are extracted from $\mathbf{A}$ and $\widehat{\boldsymbol{\Sigma}}$, $\mathbf{D}$, and $\mathbf{R}$ are computed.

3.7.4 Synopsis

The sequence of computations used to compute the sample mean vector, sample variance, and sample correlation matrix are

1. Partition the data set D as r subsets (if necessary) and distribute the subsets on separate nodes. Carry out steps 2 and 3 on nodes $j = 1, 2, \ldots, r$.
2. At node j, initialize $\mathbf{A}_j$ as a $(p+1) \times (p+1)$ matrix of zeros.
3. At node j, read D_j sequentially, one record at a time or in blocks. With the ith record:

   a. Extract $x_{i,1}, \ldots, x_{i,p}$.
   b. Form the augmented observation vector $\mathbf{w}_i$ from $x_{i,1}, \ldots, x_{i,p}$.
   c. Update $\mathbf{A}_j$ by computing $\mathbf{A}_j + \mathbf{w}_i\mathbf{w}_i^T \rightarrow \mathbf{A}_j$.

4. Transfer $\mathbf{A}_1, \ldots, \mathbf{A}_r$ to a single node and compute $\mathbf{A} = \sum_{j=1}^{r} \mathbf{A}_j$.
5. Extract the first row of $\mathbf{A}$ and compute the sample mean vector $\overline{\mathbf{x}}$.
6. Extract $\mathbf{M}$ from $\mathbf{A}$.
7. Compute $\widehat{\boldsymbol{\Sigma}}$ from n, $\overline{\mathbf{x}}$ and $\mathbf{M}$.
8. Construct $\mathbf{D}$.
9. Compute $\mathbf{R} = \mathbf{D}\widehat{\boldsymbol{\Sigma}}\mathbf{D}$.

3.8 Tutorial: Computing the Correlation Matrix

Obesity is sometimes said to be a disease of the poor [35]. From a clinical perspective, obesity is an indicator of excessive body fat, a condition that has been observed to be associated with a number of chronic diseases. For the sake of argument, we point out that claiming obesity to be a disease of the poor is contradicted by national-level obesity rates. Specifically, a comparison across countries reveals that obesity rates tend to be greater in high-income countries compared to middle- and low-income countries [43]. An opportunity to ascertain whether there is indeed a correlation between income and obesity among adult residents of the United States presents itself in the databases of Behavioral Risk Factor Surveillance System. If we entertain the hypothesis that obesity is related to diet, and diet is related knowledge of nutrition, then educational level also may be associated with obesity. We may as well include education in our investigation. The BRFSS asks survey respondents to report household income, or more precisely to identify an income bracket containing their income. Respondents also report their highest attained level of education, body weight, and height. The CDC computes body mass index from the reported weight and height of the respondents.

The clinical definition of obesity is a body mass index greater than or equal to 30 kg/m^2. From the data analytic standpoint, obesity is a binary variable and not suited for computing correlation coefficients.[14] Furthermore it's derived from a quantitative variable, body mass index. Body mass index generally is a better variable from an analytical standpoint than obesity. One need only realize that a person is obese if their body mass index is 30 or 60 kg/m^2. The consequences of excessive body fat differ substantially between the two levels of body mass index yet the binary indicator of obesity does not reflect the differences. Our investigation will use body mass index and examine associations by computing the (pair-wise) correlation coefficients between body mass index, income level, and education level of respondents using BRFSS data.

Income level is recorded in the BRFSS databases as an ordinal variable. An ordinal variable has properties of both quantitative and categorical variables. The values x and y of an ordinal variable are unambiguously ordered so that there's no debate of whether $x < y$ or vice versa. On the other hand, the practical importance of numerical difference $x-y$ may depend on x and y. For example, there are eight values of income: $1, 2, \ldots, 8$, each of which identifies a range of annual household incomes. A value of 1 identifies incomes less than \$10,000, a value of two identifies incomes between \$10,000 and \$15,000, and so on. It's unclear whether a one-level difference has the same meaning at the upper and lower range of the income variable. Education is also an ordinal variable ranging from 1 to 6, with 1 identifying the respondent as never having attended school and 6 representing 4 or more years of college. Values of income and education outside these ranges denote a refusal to answer the question or inability to answer the question. We will ignore such records.

Rather than computing each correlation coefficient associated with the three pairs of variables one at a time, a single correlation matrix containing all of the coefficients will be computed. The BRFSS data files for the years 2011, 2012, 2013, and 2014 will suffice for the investigation. We build on the tutorial of Sect. 3.6. The tutorials of Sect. 3.10 and of Chaps. 7 and 8 build on this tutorial, so as a practical matter, you should plan on re-using your code.

1. Create a `Python` script and import the necessary `Python` modules:

[14] Pearson's correlation coefficient is a measure of linear association. Linear association is meaningful when the variables are quantitative or ordinal.

```
import sys
import os
import importlib

parentDir = '/home/Algorithms/'
if parentDir not in set(sys.path):
    sys.path.append(parentDir)
    print(sys.path)
from ModuleDir import functions
reload(functions)
dir(functions)
```

The full path to `functions.py` is assumed to be `/home/Algorithms/ModuleDir/functions.py`.

2. Table 3.2 provides the field locations for the income and education variables for the years of interest. Add the field locations of the income and education entries to the dictionary `fieldDict` from the tutorial of Sect. 3.6. If you worked through the tutorial of Sect. 3.6, then the field dictionary definitions reside in the file `functions.py`. Add dictionary entries for income and education for the years 2011 through 2014. For example, the new specification for year 2014 should look like

```
fieldDict[14] = {'bmi':(2247, 2250),'weight':(2007, 2016),
                 'income':(152, 153),'education':150}
```

Table 3.2 BRFSS data file names and field locations of the income, education, and age variables

		Income		Education	Age	
Year	File	Start	End	field	Start	End
2011	LLCP2011.ZIP	124	125	122	1518	1519
2012	LLCP2012.ZIP	116	117	114	1629	1630
2013	LLCP2013.ZIP	152	153	150	2177	2178
2014	LLCP2014.ZIP	152	153	150	2232	2233

3. Create the field dictionary using the function `fieldDictBuild` that was built in the tutorial of Sect. 3.6.

```
fieldDict = functions.fieldDictBuild()
```

4. Read all the files in the directory containing the data files. You may be able to use your code from the Tutorial of Sect. 3.6 for this purpose. Ignore any files from a year before 2011 and any other files not listed in Table 3.2 by intentionally creating an exception of the type `ZeroDivisionError`.

An exception is an error that is detected during execution. It’s not necessarily fatal if trapped with an exception handler.

```
path = r’../Data/’
fileList = os.listdir(path)
for filename in fileList:
    try:
        shortYear = int(filename[6:8])
        if shortYear < 11:
            1/0
        year = 2000 + shortYear
        file = path + filename
        fields = functions.fieldDict[shortYear]
    except(ValueError, ZeroDivisionError):
        pass
```

If `shortYear` is less than 11, then a `ZeroDivisionError` exception is created by the instruction `1/0`. The exception directs program flow to the `except` branch and none of the remaining instructions in the `try` branch are executed. The `pass` statement directs program flow back to the top of the `for` loop and the next file name in `fileList` is processed. A `ValueError` exception will be generated if the string `filename[6:8]` does not consist of digits and again program flow returns to the next file name in `fileList`.

5. Extract the field positions for income, body mass index, and education from the dictionary `fields`:

```
sInc, eInc = fields[’income’]
sBMI, eBMI = fields[’bmi’]
fEduc = fields[’education’]
```

This code segment is to execute immediately after extracting `fields` from `fieldDict`.

6. Now that the necessary information regarding data file field positions have been determined, read the file by iterating over the file object (`f`) and, as a test that the program is reading the data, print the first 10 characters in each record.

```
with open(file, encoding = "latin-1") as f:
    for record in f:
        print(record[:10])
```

This code segment executes immediately after extracting the field positions of the three variables.

7. Extract the income variable from `record` and test whether the income string is missing. Income is missing if the field consists of two blanks, i.e., `’  ’`. If this is the case, then assign the integer 9 to income. We’ll use the value 9 as a flag indicating that the record should be ignored.

```
incomeString = record[sInc-1:eInc]
if incomeString != ’  ’:
    income = int(incomeString)
else:
    income = 9
```

8. The next tutorial also uses `income`, and so to reduce effort, copy the code segment and create a function for processing the income string. The function and its call can be set up as

```
def getIncome(incomeString):
    if incomeString != ’  ’:
        income = int(incomeString)
    else:
        income = 9
    return income

income = functions.getIncome(record[sInc-1:eInc])
```

   Notice that the argument passed to `getIncome` is the string containing the response to the income question. Compute and print `income` using the old code and by calling the function. When the function works correctly, move the function definition to the module `functions.py` and keep the function call in place. Remove the code segment that translates the string to integer from the main program.
9. Read the education character and test for blanks. The education field is a single character in width and so `record[fEduc-1]` extracts the variable from the string. Missing data is identified by testing for one blank rather than the two-character blank used for income.
10. Move the code segment for processing education to a function named `getEducation` and put the function in the module `functions.py`. Test the function as you did with income. Insert a call to the function after the call to `getIncome`.
11. Read the body mass index string and convert it to a float. The previous tutorial created a function named `convertBMI` to perform the task of converting the string to floating point type (see item 14 of Sect. 3.6).
12. Check that the code is functioning correctly by printing the values of income, body mass index, and education as they are processed. Remove the print statement when the code is functioning as expected.

13. If the three variables, income, body mass index, and education, have been successful converted to integers or floats, then create a vector **w** containing values of the variables.

```
if education < 9 and income < 9 and 0 < bmi < 99:
    w = np.matrix([1,income,bmi,education]).T
```

The `Numpy` function `np.matrix` creates a two-dimensional matrix. Passing a list to `Numpy` produces a $1 \times p$ matrix (mathematically, a row vector). Assigning the `.T` attribute to `np.matrix([1,income,bmi,education])` sets `w` to be the transpose of the row vector—therefore, a column vector of length four. The vector `w` is created as a `Numpy` matrix rather than a `Numpy` array so that we may easily compute matrix products using `w`.

14. Initialize the augmented matrix **A** before the `for` loop iterates over `fileList`. The dimension of **A** is $q \times q$ where $q = p + 1 = 4$ is the length of $\mathbf{w}_i$.

```
q = 4
A = np.zeros(shape=(q, q))
```

15. After computing **w**, add the outer product of **w** with itself to the augmented matrix **A**:

```
A += w*w.T
```

The program flow should look like this:

```
for filename in os.listdir(path):
    try:
        ...
        with open(file, encoding = ’latin-1’) as f:
            for record in f:
                ...
                if education < 9 and income < 9 and 0 < bmi < 99:
                    w = np.matrix([1,income,bmi,education]).T
                    A += w*w.T
                    n = A[0,0]  # Number of valid observations
```

16. Instead of waiting until all of the data has been processed to compute the correlation matrix, we'll compute it whenever the number of valid observations is a multiple of 10,000 since it takes some time to process all of the data. The following code segment computes the mean vector $\overline{\mathbf{x}}$ and extracts the moment matrix **M** from **A**:

```
if n % 10000 == 0:
    M = A[1:,1:]
    mean = np.matrix(A[1:,0]/n).T
```

The mean vector $\overline{\mathbf{x}}$ is extracted from the first row of $\mathbf{A}$ as a $1 \times q$ matrix. The syntax `A[1:,0]` extracts a sub-vector from the first column of `A` beginning with the second row and ending with the last row. Once again, the `.T` operator is used to form `mean` as a column vector.

17. Compute the variance matrix estimate $\widehat{\boldsymbol{\Sigma}} = n^{-1}\mathbf{M} - \overline{\mathbf{x}}\overline{\mathbf{x}}^T$ using the instruction

```
SigmaHat  = M/n - mean*mean.T
```

18. Construct the diagonal matrix $\mathbf{D}$ with diagonal elements $\widehat{\sigma}_1^{-1}, \ldots, \widehat{\sigma}_p^{-1}$:

```
s = np.sqrt(np.diag(SigmaHat))   # Extract the diagonal from S and
                                 #  compute the square roots.
D  = np.diag(1/s)                # Create a diagonal matrix.
```

Note that the `Numpy` function `diag` has been used in two different ways. In the first application, `np.diag(SigmaHat)` extracted the diagonal elements as a vector `s` because `diag` was passed a matrix. In the second application, `diag` was passed a vector containing the reciprocals of the standard deviation estimates $\widehat{\sigma}_i^{-1}, i = 1, \ldots, p$, and the function inserted the vector as the diagonal of a $p \times p$ matrix of zeros.

19. The product of two conformable `Numpy` matrices `A` and `B` is computed by the instruction `A*B`. Compute the correlation matrix $\mathbf{R} = \mathbf{D}\widehat{\boldsymbol{\Sigma}}\mathbf{D}$.
20. Verify that your code works correctly by computing $\mathbf{R}$ after the 10,000 observations have been processed. You may use the instruction `sys.exit()` to stop execution immediately after computing $\mathbf{R}$. The diagonal elements of $\mathbf{R}$ must be equal to 1. and the off-diagonal elements must be between -1 and 1; if not, then there is an error in the code.

    If your correlation matrix does not conform to these constraints, then it may be helpful to check the calculations of $\widehat{\boldsymbol{\Sigma}}$ and $\mathbf{D}$ using `R`. First, initialize a matrix to contain the data using the instruction `X = np.zeros(shape = (10000,3))`. Store the data as it is processed. Because `X` is zero-indexed, subtract one from `n`:

```
if n <= 10000:
    X[n-1,:] = [x for x in w[1:]]
```

21. Write the data to a file using the following instruction when `n` has reached 10,000:

```
np.savetxt('../X.txt',X,fmt='%5.2f')
```

The file will be space delimited. The field width is at least 5 and there are two places to the right of the decimal point for each number. Using `R`, read the data into a data frame using the instruction

```
X = read.table('../X.txt')
```

The `R` instructions `var(X)` and `cor(X)` will compute $\frac{n}{n-1}\widehat{\boldsymbol{\Sigma}}$ and $\mathbf{R}$. The `R` function `diag` is equivalent to the `Numpy` function of the same name.

22. Process the entire file and write the correlation matrix to an output file using the `Numpy` function `savetxt`. The correlation matrix is shown in Table 3.3.

Table 3.3 The sample correlation matrix between income, body mass index, and education computed from BRFSS data files from years 2011, 2012, 2013, and 2014, $n = 1{,}587{,}668$

	Income	Body mass index	Education
Income	1	−.086	.457
Body mass index	−.086	1	−.097
Education	.457	−.097	1

There's a very weak association between body mass index and income since the correlation coefficient between the variables is −.086.

3.8.1 Conclusion

A correlation coefficient r_{ij} between −.3 and .3 indicates weak or little linear association between variables i and j. Moderate levels of association are indicated by $.3 \le |r_{ij}| \le .7$, and strong linear association is indicated by values greater than .7 in magnitude. Based on Table 3.3, it is concluded that there is little association between body mass index and income and between body mass index and education. The negative association indicates that there is a tendency for higher levels of income and education to be associated with lower levels of body mass index. Considering the complex relationship between diet, behavior, genetics, and body mass, the results are not unexpected. The data do not directly measure these variables, but we might expect that income and education would reflect these variables. In hindsight, these variables are inadequate proxies for diet quality and nutritional knowledge. We're not ready to give up on these data though. Let us investigate a related question with the data: to what extent do income, education, and body mass index jointly explain variation in a person's perception of their health? The next section introduces a method of investigating the question.

3.9 Introduction to Linear Regression

Linear regression is a key methodology of data science. It's extremely useful for investigating the relationships between variables and for prediction, among other purposes. Methodological aspects of linear regression are discussed in Chap. 6. In this section, we solve the computational hurdle of how to compute linear regression estimates from a massively large data set. The principal algorithm is a straightforward extension of the previous algorithm for computing correlation matrices. Though the emphasis of this section is computing, a brief outline of the salient aspects of linear regression helps set the stage.

In the linear regression setting, one of the variables is to be modeled or predicted as a function of p predictor variables. Specifically, a model is adopted of the expected value of the target random variable Y given a concomitant predictor vector $\mathbf{x}$. For example, we may ask the question *to what extent may a person's health be explained by demographic attributes?* To investigate the question, a measure of health is identified and used as the target variable and measures of income, education, and body mass index are used as predictor variables.

A linear regression model will provide a simple approximation of what undoubtedly is a complex relationship. The extent to which the model adequately explains the target variable takes on special importance in the context of the question of interest. Our algorithm will compute a measure of adequacy or *model fit* known as the adjusted coefficient of determination.

Before proceeding, let's consider whether correlation—the correlation matrix specifically—suffices for addressing the question posed above. The correlation matrix measures the degree of linear association between pairs of variables drawn from a set of variables. The notion of one variable being jointly related to more than one other variable cannot be explored satisfactorily through correlation. But linear regression revolves around the notion of a target variable responding to, or being explained by a *set* of predictor variables. Thus, linear regression analysis is the path forward and almost always yields more insight than an examination of the correlation matrix. Computationally, this additional insight comes cheaply since remarkably little has to be done to the correlation matrix algorithm of Sect. 3.8 to compute linear regression estimates and the coefficient of determination.

3.9.1 The Linear Regression Model

Our interest lies in the relationship between a target variable and a set of predictor variables. The target has a preeminent role in the analysis whereas the predictor variables are in a sense, subordinate to the target. The linear regression model describes the target value as two terms: one describing the expected value of the target as a function of the predictors and second being a random term that cannot be explained by the predictors. In statistical terms, the target variable is a random variable Y and the predictor vector $\mathbf{x}$ is considered to be a vector of known values. The model is $Y = f(\mathbf{x}) + \varepsilon$. The term $f(\mathbf{x})$ is unknown and needs to be estimated, and the term ε is a random variable, commonly assumed to be normal in distribution, and basically, a nuisance. A recurrent theme of regression analysis is minimizing the importance of ε by finding a good form for the model $f(\mathbf{x})$. We need not be concerned with the distribution of ε at this point.

For example, Y may represent an individual's health measured as a score between 1 and 6, and $\mathbf{x}$ may represent concomitant measurements on variables thought to be predictive of overall health. To investigate the relationship or to predict an unobserved target value using observed predictor variables, a linear model is adopted of the expected value of Y. We may think of observing many individuals, all with the same set of predictor variable values. Their health scores will vary about some mean value, and the linear regression model describes that mean as a linear function of the predictor variables. It is hoped that the magnitudes of the random terms are small so that $Y \approx f(\mathbf{x})$. If this is the case, then the target can be predicted from the linear model with some degree of accuracy.

The linear regression model specifies that $\mathrm{E}(Y|\mathbf{x})$, the expected value, or mean of Y, is

$$\mathrm{E}(Y\,|\,\mathbf{x}) = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p.$$

The predictor vector consists of the constant 1 and the predictor variable values, hence, $\mathbf{x} = [1 \;\, x_1 \cdots \;\, x_p]^T$. The *parameter vector* is $\boldsymbol{\beta} = [\beta_0 \;\, \beta_1 \;\, \cdots \;\, \beta_p]^T$. The coefficients $\beta_0, \ldots, \beta_p$ are unknown and must be estimated. The linear regression model may be expressed more compactly in matrix form as

$$\mathrm{E}(Y\,|\,\mathbf{x}) = \mathbf{x}^T\boldsymbol{\beta}. \tag{3.24}$$

The inner product of $\mathbf{x}$ and $\boldsymbol{\beta}$ is said to be a linear combination of the p variables comprising $\mathbf{x}$. The model of Y can be written as $Y = \mathbf{x}^T\boldsymbol{\beta} + \varepsilon$.

We turn now to the problem of estimating $\boldsymbol{\beta}$ using a set of data. The data consist of pairs of observations on the target variable Y and the associated predictor vector $\mathbf{x}$. So, let $D = \{(y_1, \mathbf{x}_1), \ldots, (y_n, \mathbf{x}_n)\}$ denote the data set. The ith predictor vector is $\mathbf{x}_i = [1 \;\, x_{i,1} \cdots \;\, x_{i,p}]^T$. If we are to choose between two estimators of $\boldsymbol{\beta}$, say $\widehat{\boldsymbol{\beta}}$ and $\widetilde{\boldsymbol{\beta}}$, we prefer the estimator that produces predictions that are, on average, closer to the actual values.

With this in mind, the estimator of $\boldsymbol{\beta}$ is the parameter vector $\widehat{\boldsymbol{\beta}}$ that minimizes the sum of the squared differences between the target values y_i and the predictions $\mathbf{x}_i^T\widehat{\boldsymbol{\beta}}, i = 1, \ldots, n$. The term *residual* is applied to describe what has not been accounted for by the model and the parameter estimate $\widehat{\boldsymbol{\beta}}$; thus, the ith residual is $y_i - \mathbf{x}_i^T\widehat{\boldsymbol{\beta}}$.

3.9.2 The Estimator of β

The least squares estimator of $\boldsymbol{\beta}$ is determined by finding the vector that minimizes the sum of the squared residuals

$$S(\boldsymbol{\beta}) = \textstyle\sum_{i=1}^n \left(y_i - \mathbf{x}_i^T\boldsymbol{\beta}\right)^2. \tag{3.25}$$

Minimizing $S(\boldsymbol{\beta})$ with respect to $\boldsymbol{\beta}$ is the *least squares criterion.* The objective function $S(\cdot)$ can be expressed in matrix form by stacking the predictor vectors as rows of the matrix

$$\underset{n\times q}{\mathbf{X}} = \begin{bmatrix} \underset{1\times q}{\mathbf{x}_1^T} \\ \vdots \\ \underset{1\times q}{\mathbf{x}_n^T} \end{bmatrix}.$$

The matrix $\mathbf{X}$ is referred to as the design or model matrix. We also set up the n-vector $\mathbf{y} = [y_1 \;\; y_2 \;\; \cdots \;\; y_n]^T$ to contain the target observations. Then,

$$\mathrm{E}(\mathbf{Y}|\mathbf{X}) = \underset{n\times q}{\mathbf{X}}\underset{q\times 1}{\boldsymbol{\beta}} \tag{3.26}$$

is the n-vector of expected values. Furthermore, we may write the objective function as an inner product of differences between the vector of observed values and the vector of expectations:

$$S(\boldsymbol{\beta}) = (\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^T(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}). \tag{3.27}$$

Of course, $\boldsymbol{\beta}$ is unknown, and the task at hand is to determine the best possible estimator. The best estimator is the vector $\widehat{\boldsymbol{\beta}}$ that minimizes $S(\boldsymbol{\beta})$ given that we judge estimators by the least squares criterion. The form of the estimator can be determined by differentiating $S(\boldsymbol{\beta})$ with respect to $\boldsymbol{\beta}$, setting the vector of partial derivatives equal to the zero vector, and solving the resulting system of equations for $\boldsymbol{\beta}$. The system, often referred to as the normal equations, is

$$\underset{q\times q}{\mathbf{X}^T\mathbf{X}}\underset{q\times 1}{\boldsymbol{\beta}} = \underset{q\times n}{\mathbf{X}^T}\underset{n\times 1}{\mathbf{y}}. \tag{3.28}$$

Exercise 3.3 guides the reader through the derivation. The solution to the normal equations, and therefore the least squares estimator of $\boldsymbol{\beta}$, is

$$\widehat{\boldsymbol{\beta}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}. \tag{3.29}$$

The solution shown in Eq. (3.29) requires that $\mathbf{X}^T\mathbf{X}$ be invertible; if $\mathbf{X}^T\mathbf{X}$ is singular, then the computation of an estimator of $\boldsymbol{\beta}$ is more difficult. The immediate problem is that the solution for $\widehat{\boldsymbol{\beta}}$ expressed in formula (3.29) may not be computationally tractable as it requires $\mathbf{X}$, a matrix of n rows, and n may be so large that $\mathbf{X}$ cannot easily be stored. A computational algorithm for computing $\widehat{\boldsymbol{\beta}}$ that avoids constructing $\mathbf{X}$ is not difficult to develop since we can use the same methods as were used for computing the correlation matrix.

The algorithm for computing the correlation matrix solves the problem of an intractably large design matrix because $\mathbf{X}^T\mathbf{X} = \mathbf{A}$ (formula (3.22)) and the dimension of $\mathbf{A}$ is $q \times q$. The matrix $\mathbf{A}$ is the augmented matrix and an associative statistic. The vector $\mathbf{z} = \mathbf{X}^T\mathbf{y}$ is also associative, and so the two terms in the solution shown in Eq. (3.29) can be computed without much difficulty. Changing notation to write the solution involving the associative statistics sets the solution to be

$$\widehat{\boldsymbol{\beta}} = \underset{q\times q}{\mathbf{A}^{-1}}\ \underset{q\times 1}{\mathbf{z}} \tag{3.30}$$

Not only are the matrices $\mathbf{A}$ and $\mathbf{z}$ small, but they can be computed without holding all of the data in memory. To understand why $\mathbf{A} = \mathbf{X}^T\mathbf{X}$ (formula (3.22)), let $\mathbf{x}_k$ denote the $n \times 1$ *column* vector of observations on the kth predictor variable, $k = 1, 2, \ldots, p$, and $\mathbf{1}$ denote a $n \times 1$ vector of ones. Then, $\mathbf{X}^T\mathbf{X}$ can be expressed as a matrix consisting of the q^2 inner products since

$$\begin{aligned}
\underset{q\times q}{\mathbf{X}^T\mathbf{X}} &= \underset{q\times n}{\begin{bmatrix} \mathbf{1}^T \\ \mathbf{x}_1^T \\ \vdots \\ \mathbf{x}_p^T \end{bmatrix}} \underset{n\times q}{\begin{bmatrix} \mathbf{1} & \mathbf{x}_1 & \cdots & \mathbf{x}_p \end{bmatrix}} \\
&= \underset{q\times q}{\begin{bmatrix} \mathbf{1}^T\mathbf{1} & \mathbf{1}^T\mathbf{x}_1 & \cdots & \mathbf{1}^T\mathbf{x}_p \\ \mathbf{x}_1^T\mathbf{1} & \mathbf{x}_1^T\mathbf{x}_1 & \cdots & \mathbf{x}_1^T\mathbf{x}_p \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{x}_p^T\mathbf{1} & \mathbf{x}_p^T\mathbf{x}_1 & \cdots & \mathbf{x}_p^T\mathbf{x}_p \end{bmatrix}} \\
&= \begin{bmatrix} n & \sum_i x_{i,1} & \cdots & \sum_i x_{i,p} \\ \sum_i x_{i,1} & \sum_i x_{i,1}^2 & \cdots & \sum_i x_{i,1}x_{i,p} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_i x_{i,p} & \sum_i x_{i,p}x_{i,1} & \cdots & \sum_i x_{i,p}^2 \end{bmatrix} = \sum_{i=1}^{n} \mathbf{x}_i\mathbf{x}_i^T.
\end{aligned} \tag{3.31}$$

Hence, $\mathbf{A} = \mathbf{X}^T\mathbf{X}$, since $\mathbf{A}$ is the sum of the outer products. Formula (3.31) shows that $\mathbf{A}$ is associative. Similarly, the second term $\mathbf{z} = \mathbf{X}^T\mathbf{y}$ in Eq. (3.29) also is an associative statistic since

$$\begin{aligned}\underset{q\times 1}{\mathbf{z}} &= \underset{q\times n}{\mathbf{X}^T}\underset{n\times 1}{\mathbf{y}}\\ &= \underset{q\times 1}{\begin{bmatrix}\mathbf{1}^T\mathbf{y}\\ \mathbf{x}_1^T\mathbf{y}\\ \vdots\\ \mathbf{x}_p^T\mathbf{y}\end{bmatrix}} = \begin{bmatrix}\sum y_i\\ \sum x_{i,1}y_i\\ \vdots\\ \sum x_{i,p}y_i\end{bmatrix} = \sum \underset{q\times 1}{\mathbf{x}_i}\underset{1\times 1}{y_i}\,.\end{aligned} \tag{3.32}$$

The vector $\mathbf{x}_i^T$ is the ith *row* vector. (We're using the index i for row vectors and k column vectors.) Both $\mathbf{A}$ and $\mathbf{z}$ can be computed by iterating over observation pairs $(\mathbf{x}_1, y_1)$, ..., $(\mathbf{x}_n, y_n)$. It's useful to have one further expression for $\widehat{\boldsymbol{\beta}}$ that explicitly shows the predictor vectors:

$$\widehat{\boldsymbol{\beta}} = (\textstyle\sum_{i=1}^n \mathbf{x}_i\mathbf{x}_i^T)^{-1}\sum \underset{q\times 1}{\mathbf{x}_i}\underset{1\times 1}{y_i}\,. \tag{3.33}$$

Let $\mathbf{t}(D) = (\mathbf{A}, \mathbf{z})$ denote the pair of statistics. It's also associative since $\mathbf{A}$ and $\mathbf{z}$ are both associative. A scalable algorithm for computing $\mathbf{t}(D)$ is obtained by a minor modification of the algorithm for computing the correlation matrix. As before, the algorithm iterates over the observations in the data set $D = \{(y_1, \mathbf{x}_1), \ldots, (y_n, \mathbf{x}_n)\}$. The preliminary steps are:

1. Initialize $\mathbf{A}$ as a $q \times q$ matrix of zeros.
2. Initialize $\mathbf{z}$ as a q-vector of zeros.

Process D sequentially and carry out the following steps:

1. From the ith pair of D, extract $y_i, x_{i,1}, \ldots, x_{i,p}$ and construct $\mathbf{x}_i$ according to
$$\{x_{i,1} \;\cdots\; x_{i,p}\} \to [1 \;\; x_{i,1} \;\cdots\; x_{i,p}]^T = \underset{q\times 1}{\mathbf{x}_i}\,.$$
It's assumed that the intercept β_0 is to be included in the model. If not, then $\mathbf{x}_i$ is not augmented with 1.
2. Update $\mathbf{A}$ and $\mathbf{z}$ according to
$$\mathbf{A} + \mathbf{x}_i\mathbf{x}_i^T \to \mathbf{A}.$$
and
$$\mathbf{z} + y_i\mathbf{x}_i \to \mathbf{z}.$$
3. If the data has been partitioned as sets $D_1, \ldots, D_r$, then compute $\mathbf{A}_1, \ldots, \mathbf{A}_r$ and $\mathbf{z}_1, \ldots, \mathbf{z}_r$ as described above. Upon completion, compute

$$\begin{aligned}\mathbf{A} &= \textstyle\sum_{j=1}^{r} \mathbf{A}_j \\ \mathbf{z} &= \textstyle\sum_{j=1}^{r} \mathbf{z}_j.\end{aligned} \tag{3.34}$$

4. Compute $\widehat{\boldsymbol{\beta}} = \mathbf{A}^{-1}\mathbf{z}$.

3.9.3 Accuracy Assessment

In general, assessing the accuracy of a predictive function (not necessarily a linear regression prediction function) is best accomplished using cross-validation or bootstrapping. We'll discuss cross-validation in Chap. 9. Fortuitously, elementary and informative accuracy estimates for regression-based prediction functions can be computed essentially at the same time that the parameter estimates are computed. We'll develop one such measure of accuracy that provides a measure of the value of the predictor variables towards explaining variation in the target variable. It will be used to determine the value of income, education, and body mass index toward explaining variation in health score.

Recall that standard deviation estimator $\widehat{\sigma} = \left[n^{-1}\sum(y_i - \widehat{\mu})^2\right]^{1/2}$ measures the average absolute error resulting from using $\widehat{\mu}$ as an estimator of the expected value of Y. Without information about Y_i in the form of a predictor vector $\mathbf{x}_i$, μ is traditionally estimated by the sample mean $\overline{y}$. A similar estimator of the error is computed when regression is used to estimate the ith expected value of the response variable. The *regression mean squared error* σ^2_{reg} is the expected squared error associated with using $\mathbf{x}_i^T\widehat{\boldsymbol{\beta}}$ as an estimator of the expected value of Y_i, and $\widehat{\sigma}_{\text{reg}}$ is the estimated error incurred by using $\mathbf{x}_i^T\widehat{\boldsymbol{\beta}}$ as an estimator of the expected value of Y_i. An informative regression model ought to yield a substantially smaller error than the sample mean $\overline{y}$. The next step is to develop a measure of model accuracy that is based on the reduction in error provided by the regression model compared to the sample mean.

Estimators of σ^2 and σ^2_{reg} are

$$\begin{aligned}\widehat{\sigma}^2 &= \frac{\sum_{i=1}^{n}(y_i - \overline{y})^2}{n} \\ \text{and } \widehat{\sigma}^2_{\text{reg}} &= \frac{\sum_{i=1}^{n}(y_i - \widehat{y}_i)^2}{n-q},\end{aligned} \tag{3.35}$$

where $\widehat{y}_i = \mathbf{x}_i^T\widehat{\boldsymbol{\beta}}$ is the ith *fitted* value. Except in special circumstances, it's difficult to interpret $\widehat{\sigma}^2_{\text{reg}}$ without reference to $\widehat{\sigma}^2$. This difficulty motivates a relative measure of model fit, the *adjusted coefficient of determination*

$$R^2_{\text{adjusted}} = \frac{\widehat{\sigma}^2 - \widehat{\sigma}^2_{\text{reg}}}{\widehat{\sigma}^2}. \tag{3.36}$$

The adjusted coefficient of determination measures the relative reduction in mean square error when comparing the fitted regression model to simplest of data-based prediction functions, namely, $\overline{y}$. The range of R^2_{adjusted} is 0–1, and values of R^2_{adjusted} greater than .8 usually indicate that the fitted regression model is useful for prediction. Furthermore, R^2_{adjusted} may be interpreted as the proportion of the variation in Y that is explained by the fitted regression model.

3.9.4 Computing $R^2_{adjusted}$

Turning now to the matter of computing R^2_{adjusted}, formula (3.36) is a simple function of the terms $\widehat{\sigma}^2$ and $\widehat{\sigma}^2_{\text{reg}}$. Equation (3.35) suggests that the data must be processed twice to compute the terms: the first time to compute $\overline{y}$ and $\widehat{\boldsymbol{\beta}}$, and the second time to compute the sum of the squared deviations. However, we saw earlier (formula (3.16)) that $\widehat{\sigma}^2$ may be computed using a single pass through the data since $\widehat{\sigma}^2 = n^{-1}\sum y_i^2 - \overline{y}^2$. The regression mean square error also can be computed without a second pass through the data since

$$\widehat{\sigma}^2_{\text{reg}} = \frac{\sum_i y_i^2 - \mathbf{z}^T\widehat{\boldsymbol{\beta}}}{n-q}, \tag{3.37}$$

where

$$\underset{q\times 1}{\mathbf{z}} = \sum_{i=1}^{n} \underset{q\times 1}{\mathbf{x}_i}\, \underset{1\times 1}{y_i}\,.$$

Exercise 3.8 guides the reader in a verification of formula (3.37).

As the algorithm iterates over observation pairs $(y_1, \mathbf{x}_1), \ldots, (y_n, \mathbf{x}_n)$ in the calculation of $\widehat{\boldsymbol{\beta}}$, we accumulate the sum of squares $\sum_i y_i^2$. The term $\mathbf{z}$ is necessarily accumulated for the computation of $\widehat{\boldsymbol{\beta}}$ and so is available to compute $\widehat{\sigma}_{\text{reg}}$ when the algorithm completes.

If n is much larger than q so that $n - q \approx n$, then computing R^2_{adjusted} is even simpler. Instead of using the correct denominator $n - q$ in computing $\widehat{\sigma}^2_{\texttt{reg}}$, we'll use n. Then,

$$\begin{aligned} R^2_{\text{adjusted}} &= \frac{\widehat{\sigma}^2 - \widehat{\sigma}^2_{\texttt{reg}}}{\widehat{\sigma}^2} \\ &\approx \frac{n^{-1}\sum_i y^2 - \overline{y}^2 - n^{-1}\left(\sum_i y^2 - \mathbf{z}^T\widehat{\boldsymbol{\beta}}\right)}{\widehat{\sigma}^2} \\ &= \frac{n^{-1}\mathbf{z}^T\widehat{\boldsymbol{\beta}} - \overline{y}^2}{\widehat{\sigma}^2}. \end{aligned} \tag{3.38}$$

The alternative to computing $\widehat{\sigma}^2_{\text{reg}}$ as described above utilizes a second pass through the data file. The sole purpose is to compute the sum of the squared prediction errors

$$\sum_{i=1}^{n}(y_i - \mathbf{x}_i^T\widehat{\boldsymbol{\beta}})^2 = (n-q)\widehat{\sigma}^2_{\text{reg}}. \tag{3.39}$$

3.10 Tutorial: Computing $\widehat{\boldsymbol{\beta}}$

For a long time it has been argued that income is related to health [16, 17]. The evidence supporting this contention is clouded by the difficulty of quantifying a condition as complex as health. The problem is also confounded with confidentiality issues that arise with potentially sensitive data on the health of individuals. In this tutorial, we investigate the question by exploring data from the Behavioral Risk Factor Surveillance System Survey (BRFSS). Specifically, we build a regression model of a very simple measure of health, the answer to the question *would you say that in general your health is:* The question is multiple choice and the possible answers are shown in Table 3.4.

Table 3.4 Possible answers and codes to the question *would you say that in general your health is:*

Code	Descriptor
1	Excellent
2	Very good
3	Good
4	Fair
6	Poor
7	Don't know or not sure
9	Refused
Blank	Not asked or missing

After discarding records containing the codes 7, or 9, or an empty string, the recorded variable is ordinal and suitable for regression analysis.

There is an abundance of demographic variables recorded in the BRFSS data files but we limit the predictor variables to a few demographic descriptors including of course, annual household income. Our objectives are limited to assessing the relative importance of the variables income, body mass index, and education toward predicting health. We also would like to judge whether income is more important than the other variables. The objectives are pursued by fitting a linear regression model of the health measure and examining the estimated effect of a 1 unit change of each on health.

The BRFSS data files for the years 2011, 2012, 2013, and 2014 will once again be used. In addition to income, body mass index, and education extracted in the previous tutorial, we will need to extract responses to the health question. Table 3.5 shows the field positions the variable. Valid responses are integers between 1 and 6 inclusive. Records with general health responses that are entered in the data files as 7, 9, or blank will be ignored.

Table 3.5 BRFSS data file names and field positions of the general health variable. The general health variable contains a respondent’s assessment of their health. Table 3.4 describes the general health codes and meaning of each

Year	File	Position
2011	LLCP2011.ZIP	73
2012	LLCP2012.ZIP	73
2013	LLCP2013.ZIP	80
2014	LLCP2014.ZIP	80

This tutorial builds on the correlation tutorial of Sect. 3.8. The instructions that follow assume that the reader has written a script that computes a correlation matrix for some of the BRFSS variables. If you do not have a script, then you should go through the tutorial and create it.

1. Load modules and functions that were used in the previous tutorial:

```
import sys
import os
import importlib

parentDir = ’/home/Algorithms/’
if parentDir not in set(sys.path):
    sys.path.append(parentDir)
    print(sys.path)
from ModuleDir import functions
reload(functions)
dir(functions)
```

   Execute the script and correct any errors.
2. Edit the module `functions.py` and add the field position entries for the general health variable to `fieldDict`. The field positions are listed in Table 3.5. The dictionary entry for the year 2011 may appear as

```
fieldDict[11] =  {’genhlth’:73, ’bmi’:(1533, 1536), ’income’:124, 125),
    ’education’:122}
```

3. Create the field dictionary using the function `fieldDictBuild`.

```
fieldDict = functions.fieldDictBuild()
```

4. Create a list of files in your data directory and iterate through the list. Process the files from the years 2011, 2012, 2013, and 2014.

```
path = r'.../Data/'
fileList = os.listdir(path)
print(fileList)
n = 0
for filename in fileList:
    try:
        shortYear = int(filename[6:8])
        if shortYear < 11:
            1/0
        year = 2000 + shortYear

        file = path+filename
        print(filename)
        fields = fieldDict[shortYear]
    except(ValueError, ZeroDivisionError):
        pass
```

   Since there is an exception handler in the code to skip files that are not data files, we also use the exception handler to skip data from years preceding 2011. The last statement in the code segment retrieves the field positions for the current year.
5. Extract the field positions for education, income, and body mass index using the functions that you created in the previous tutorial. Get the field position `fGH` for general health.

```
fEduc = fields['education']
sInc, eInc = fields['income']
sBMI, eBMI = fields['bmi']
fGH = fields['genhlth']
```

6. Process the data file by iterating over records. Extract the predictor variable values:

```
file = path + filename
with open(file, encoding = "latin-1") as f:
    for record in f:
        education = functions.getEducation(record[fEduc-1])
        income = functions.getIncome(record[sInc-1:eInc])
        bmi = functions.convertBMI(record[sBMI-1:eBMI],shortYear)
```

   This code segment executes within the `for` loop that iterates over `fileList`. Therefore, the `for record in f` statement must have the same indentation as the code segment of instruction 5. The next code segment handles invalid values of general health.

7. Map the string genHlthString to the integer genHlth as follows. First test whether the string is a blank. If it is, then assign -1 to genHlth. If the string is not blank, translate the string to an integer. Then test whether the integer is 7 or 9 (or simply greater than 6), and if so, assign -1 to genHlth. No other transformation is necessary.

   Print the values of genHlthString and genHlth as the program iterates over records to check your code. No other values of genHlth besides those in $\{-1, 1, 2, 3, 4, 5, 6\}$ should be produced.
8. Encapsulate the code segment from instruction 7 as a function. The definition and return statements are:

```
def getHlth(HlthString):
    ...
    return genHlth
```

9. Call the function so:

```
    y = functions.getHlth(record[fGH-1])
```

   Check that the function getHlth works correctly by printing the return value y. When the function works correctly, move it to the functions module. Delete the code segment from the main program. Call getHlth after computing bmi.
10. If the extracted values are all valid, then form the predictor vector $\mathbf{x}$ from education, income, and body mass index. The vector $\mathbf{x}$ is created as a 4×1 Numpy matrix so that we may easily compute the outer product $\mathbf{x}\mathbf{x}^T$. If one or more extracted values are not valid, then a ZeroDivisionError exception is created and program flow is directed to the next record.

```
try:
    if education < 9 and income < 9 and 0 < bmi < 99 and y != -1:
        x = np.matrix([1, income, education, bmi]).T
        n += 1
    else:
        1/0
    except(ZeroDivisionError):
        pass
```

11. The next set of operations build the matrices $\mathbf{A}$ and $\mathbf{z}$ from $\mathbf{x}$ and y. But before computing the matrices $\mathbf{A}$ and $\mathbf{z}$, it is necessary to initialize

them as matrices containing zeros because they will be computed by successively adding additional terms to the matrices. Initializations must take place before any of the data files are processed. The initialization code segment is

```
q = 4
A = np.zeros(shape = (q, q))
z = np.matrix(np.zeros(shape = (q, 1)))
sumSqrs = 0
n = 0
```

The variable `sumSqrs` will contain $\sum_i y_i^2$ (needed to compute $\widehat{\sigma}^2$).

12. Returning to the inner `for` loop, as each record is processed, update $\mathbf{A}$ and $\mathbf{z}$. This code follows the test for valid values of the target and predictor variables.

```
A += x*x.T
z += x*y
sumSqrs += y**2
n += 1
```

These instructions are executed *only* if the test for valid values is `True` (instruction 10). Thus, the code segment follows immediately after updating $\mathbf{A}$.

13. After processing a large number of observations, compute

$$\widehat{\boldsymbol{\beta}} = \mathbf{A}^{-1}\mathbf{z}. \tag{3.40}$$

You can compute $\mathbf{A}^{-1}$ using the `Numpy` function `linalg.inv()`, say, `invA = np.linalg.inv(A)`, and then compute `betaHat = invA*z`. However, from a numerical standpoint it's preferable not to compute $\mathbf{A}^{-1}$ but instead solve the linear system of equations $\mathbf{A}\boldsymbol{\beta} = \mathbf{z}$ for $\boldsymbol{\beta}$ using a LU-factorization optimized for symmetric matrices.[15] The solution to the system will be $\widehat{\boldsymbol{\beta}}$. Compute $\widehat{\boldsymbol{\beta}}$ after processing successive sets of 10,000 observations:

```
if n  % 10000 == 0 and n != 0:
    b = np.linalg.solve(A,z)
    print('\t'.join([str(float(bi)) for bi in b]))
```

[15] The LU-factorization method is faster and more accurate than computing the inverse and then multiplying.

The calculation of $\widehat{\boldsymbol{\beta}}$ shown here is carried out by means of an LU-factorization. The `join` operator creates a string from the elements of $\widehat{\boldsymbol{\beta}}$ and inserts a tab character in between each element.

14. The next task is to compute

$$R^2_{\text{adjusted}} = \frac{\widehat{\sigma}^2 - \widehat{\sigma}^2_{\text{reg}}}{\widehat{\sigma}^2} \tag{3.41}$$

using the variance estimates

$$\begin{aligned} \widehat{\sigma}^2 &= n^{-1}\textstyle\sum_i y_i^2 - \overline{y}^2 \\ \text{and } \widehat{\sigma}^2_{\text{reg}} &= n^{-1}\left(\sum y_i^2 - \mathbf{z}^T\widehat{\boldsymbol{\beta}}\right). \end{aligned} \tag{3.42}$$

All terms necessary to compute R^2_{adjusted} have been computed except for $\overline{y} = n^{-1}\sum_i y_i$. The sum $\sum_i y_i$ is stored in row zero of $\mathbf{z}$. Hence, $\widehat{\sigma}^2$ is computed using the instruction

```
ybar = z[0]/n
varEst = sumSqrs/n - ybar**2
```

15. Compute the adjusted coefficient of determination. We use the approximation formula for $R^2_{\texttt{adjusted}}$ from Eq. (3.38).

```
rAdj = (z.T.dot(b)/n - ybar**2)/varEst
```

The `.dot` operator is used to compute the inner product of `Numpy` arrays `z` and `b`. Since the `Numpy` function `linalg.solve` does not return a matrix (it returns an array), the matrix multiplication operator `*` will not multiply `z` and `b` correctly. You can determine the type of an object using the function `type()`.

16. Print the value of R^2_{adjusted} and $\widehat{\boldsymbol{\beta}}$ whenever n mod 10,000 $= 0$ and $n > 0$ by putting the following code within the `if` branch of the conditional statement of instruction 13.

```
bList = [str(round(float(bi),3)) for bi in b]
print(n,'\t'.join(bList),str(round(float(rAdj),3)))
```

The print statements and all of the intermediate calculations needed to compute R^2_{adjusted} should execute whenever $\widehat{\boldsymbol{\beta}}$ is computed. In the first line of the code segment, `b` is converted from a `Numpy` matrix to list of strings.[16]

17. The ith parameter estimate $\widehat{\beta}_i$ can be interpreted as the estimated change in $\mathrm{E}(Y|\mathbf{x})$ if ith predictor variable increases by one unit and all other variables are held fixed.[17] For example, the parameter estimate $\widehat{\beta}_1 = -.145$ associated with income implies that a one unit change in the income variable is associated with an average improvement in general health score of $-.145$. We call this change an improvement because a general health score 1 identifies excellent health and six identifies poor health. Recall that the income variable is ordinal, and that a 1 unit change in the income variable corresponds to roughly \$5000 more in income per year. A one unit increase in body mass index is associated with an increase of $\widehat{\beta}_3 = .037$ in the health score. Body mass index is computed as weight in kilograms divided by height^2 and so, if an individual gains 10 kg (or 22 lbs), then we estimate the general health score to increase by .37. It's very difficult to say which variable is more important because the units (dollars and kilograms are not comparable).
18. Remove education from the model and compute R_{adjusted}. The difference between R_{adjusted} with all three values in the model and only income and body mass index provides a measure of the importance of education. As the difference is .016, we interpret education as accounting for $19.9 - 18.3 = 1.6\%$ of the variation in general health score, not a very impressive amount.

3.10.1 Conclusion

The tutorial found an adjusted coefficient of determination of .199, indicating that the variables explained 19.9% of the variation in the general health score. The model's failure to explain much of the variation in general health score is understandable because income and education do not directly affect health. Despite there being a great deal of variation in general health score that is not explained by this model, we see that differences in body mass index and income are associated with differences in the perceived health of respondents as measured by the health score.

If we remove income from the model, the coefficient of determination is less—.134 and if we remove body mass index instead, $R_{\text{adjusted}} = .158$. From this comparison, we infer that income is a relatively more important determinant of general health score than education or body mass index. This result supports the notion that income and health are connected.

[16] `Numpy` matrices and arrays cannot be rounded even if they are of length 1 or 1 × 1 in dimension.

[17] Interpretation of regression coefficients is discussed at length in Chap. 6.

3.11 Exercises

3.11.1 Conceptual

3.1. Suppose that $\mathbf{s}$ is a statistic and that for any two sets D_1 and D_2 that form a partition of the set D, $\mathbf{s}(D_1 \cup D_2) = \mathbf{s}(D_1) + \mathbf{s}(D_2)$. Suppose that $r > 2$ and $D_1, D_2, \ldots, D_r$ is also a partition of D. Argue that $\mathbf{s}(D) = \mathbf{s}(D_1) + \mathbf{s}(D_2) + \cdots + \mathbf{s}(D_r)$.

3.2. Show that the statistic $\mathbf{t}(D) = (\mathbf{A}, \mathbf{z})$ (Eqs. (3.22) and (3.32)) is an associative statistic. Recall that pairs are added component-wise; hence, $(x_1, y_1) + (x_2, y_2) = (x_1 + x_2, y_1 + y_2)$.

3.3. (Requires multivariable calculus.) Show that the least squares estimator is the solution to the normal equations (3.28). Specifically, minimize the objective function $S(\boldsymbol{\beta})$ (Eq. (3.25)) with respect to $\boldsymbol{\beta}$. Differentiate $S(\boldsymbol{\beta})$ with respect to $\boldsymbol{\beta}$. Set the vector of partial derivatives equal to $\underset{q\times 1}{\mathbf{0}}$ and solve for $\boldsymbol{\beta}$. The solution is the least squares estimator.

3.4. Recall that $\widehat{\sigma}^2$ can be written as

$$\widehat{\sigma}^2 = \frac{\sum(x_i - \widehat{\mu})^2}{n},$$

and consider the variance estimator presented in Eq. (3.16).

(a) Verify that Eq. (3.3) are correct; that is, verify that $\widehat{\mu}$ and $\widehat{\sigma}^2$ can be computed from s_1, s_2, and s_3 according to the formulae.

(b) Conventional statistical texts advocate using the sample variance

$$s^2 = \frac{\sum(x_i - \widehat{\mu})^2}{n-1}.$$

to estimate σ^2 because it is unbiased. In fact, $\widehat{\sigma}^2$ is a biased estimator of σ^2. Show that the difference between $\widehat{\sigma}^2$ and s^2 tends to 0 as $n \to \infty$. Consequently, when n is large, the bias of $\widehat{\sigma}^2$ is negligible.

3.5. Let

$$\underset{n\times 1}{\mathbf{j}} = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}$$

denote the *summation* vector of length n.

(a) Show that for any n-vector $\mathbf{x}$, $n^{-1}\mathbf{j}^T\mathbf{x} = \overline{x}$.

(b) Suppose that the p-vector of sample means $\overline{\mathbf{x}}$ is to be computed from the data matrix $\underset{n\times p}{\mathbf{X}}$. Show that

$$\overline{\mathbf{x}}^T = (\mathbf{j}^T\mathbf{j})^{-1} \underset{1\times n}{\mathbf{j}^T} \underset{n\times p}{\mathbf{X}}. \tag{3.43}$$

(c) Assuming that $\mathbf{j}$ and $\mathbf{X}$ have been constructed as `Numpy` matrices, give a one-line Python instruction that will compute $\overline{\mathbf{x}}^T$.

(d) Note the similarity between Eq. (3.43) and the least squares estimator of $\boldsymbol{\beta}$. What can you deduce about the sample mean as an estimator?

3.6. Suppose that $X_2 = aX_1 + b$, where X_1 is a random variable with finite mean and variance and $a \neq 0$ and b are real numbers. Show that the population correlation coefficient $\rho_{12} = 1$.

3.7. Show that

$$\mathbf{R} = \mathbf{D}\widehat{\boldsymbol{\Sigma}}\mathbf{D}$$

by writing out some of the elements of $\widehat{\boldsymbol{\Sigma}}\mathbf{D}$ and then the elements of the product $\mathbf{D}(\widehat{\boldsymbol{\Sigma}}\mathbf{D})$.

3.8. Prove formula (3.37) by expanding $\sum(y_i - \widehat{y}_i)^2$ as

$$\sum(y_i - \widehat{y}_i)^2 = \mathbf{y}^T\mathbf{y} - 2\mathbf{y}^T\mathbf{X}\widehat{\boldsymbol{\beta}} + \widehat{\boldsymbol{\beta}}^T\mathbf{X}^T\mathbf{X}\widehat{\boldsymbol{\beta}}. \tag{3.44}$$

Then, prove that $\widehat{\boldsymbol{\beta}}^T\mathbf{X}^T\mathbf{X}\widehat{\boldsymbol{\beta}} = \mathbf{z}^T\widehat{\boldsymbol{\beta}}$.

3.11.2 Computational

3.9. The standard deviation of the ith parameter estimator $\widehat{\beta}_i$ is estimated by the square root of the ith diagonal element of the matrix

$$\widehat{\mathrm{var}}(\widehat{\boldsymbol{\beta}}) = \widehat{\sigma}^2_{\mathrm{reg}}(\mathbf{X}^T\mathbf{X})^{-1}. \tag{3.45}$$

An approximate large-sample 95% confidence interval for β_i is

$$\left[\widehat{\beta}_i - 2\sqrt{\widehat{\mathrm{var}}(\widehat{\beta}_i)}, \widehat{\beta}_i + 2\sqrt{\widehat{\mathrm{var}}(\widehat{\beta}_i)}\right]. \tag{3.46}$$

Compute confidence intervals for the parameters β_1, β_2, and β_3 estimated in Sect. 3.10.

3.10. Mokdat et al. [40] estimate the percent of obese U.S. adult residents to be 19.8% and 21.5% in years 2000 and 2001 from an analysis of BRFSS data. Determine whether their calculations are correct by modifying the tutorial of Sect. 3.6 to compute the proportion of obese respondents for each of the 8 years listed in Table 3.1.

3.11. Use one of the BRFSS data sets to estimate the proportion of females in the U.S. adult population. Compute an estimate as the sample proportion of females among all observations. The gender variable field positions are listed in the BRFSS codebooks. The year 2014 codebook can be viewed at the CDC website https://www.cdc.gov/brfss/annual_data/2014/pdf/codebook14_llcp.pdf. Males are identified in the BRFSS data files by the value 1 and females are identified by 2. Let x_j denote the gender of the jth respondent.

Compute two estimates: the conventional sample proportion, and a weighted proportion using the BRFSS sampling weights. The two estimators are

$$p_1 = n^{-1}\textstyle\sum_{j=1}^{n} I_F(x_j)$$

$$\text{and } p_2 = \frac{\sum_{j=1}^{n} w_j I_F(x_j)}{\sum_{j=1}^{n} w_j}, \tag{3.47}$$

where $I_F(x_j)$ takes on the value 1 if the jth sampled individual is female and 0 otherwise, w_j is the BRFSS sampling weight assigned to the jth observation, and n is the number of observations. Compare to published estimates of the proportion of females in the adult U.S. population. The U.S. Census Bureau reports that in 2010, 50.8% of the population were female.

3.12. The University of California Irvine Machine Learning Repository [5] maintains a data set related to household electric power consumption. Data loggers were connected to three sub-meters that measured consumption on three separate circuits. Data were collected by minute for approximately 47 months from December 2006 and November 2010. More information on the study can be obtained at the University of California, Irvine Machine Learning Repository. Active power is the response variable. Reactive power largely reflects energy loss due to resistance and is ignored in this exercise.

(a) Map the data from consumption per minute (of active power) to consumption per hour (active power) by building a dictionary, `hourlyDict` of key-value pairs where the keys are day and hour pairs, for instance, $(02/11/2007, 13)$ and values are lists containing the consumption for the three sub-meters during the hour, e.g., $[3, 16, 410]$.

(b) Compute the correlation matrix for the three circuits using the hourly consumption data contained in `hourlyDict`.

(c) Map the hourly data contained in `hourlyDict` to hour. Do this by creating an hour-of-the-day dictionary `hourDict` where the keys are $0, 1, \ldots, 23$ and the values are the means of all measurements obtained for a particular hour of the day. On a single plot, graph mean consumption against hour for each of the three sub-meters and describe the pattern of use by hour.

Chapter 4
Hadoop and MapReduce

Abstract In this chapter we consider situations in which a single host computer is inadequate because the data volume or processing demand exceeds the capacity of the host. A popular solution distributes the data and computations across a network of computers or a short-lived network created for the task (a cluster). In this scenario, each cluster node (a computing unit) stores and processes a subset of the data. The results are merged as one when all nodes have been completed their tasks. For this solution to succeed, the computational algorithm must conform to a certain structure and the cluster execution must be managed. The Hadoop environment and the MapReduce programming design provide the management and algorithmic structure. Hadoop is a collection of software and services that builds the cluster, distributes the data across the cluster, and controls the data processing algorithms and the transmission of results. The MapReduce programming design insures scalability, and scalability insures that the results are independent of the cluster configuration. The reader is guided through an introductory application of Hadoop and MapReduce after a discussion of the essential components.

4.1 Introduction

Google, Inc. developed the MapReduce programming design [14, 15]. The motivation for MapReduce was the need for building and regularly updating an index of all internet web pages. As the number of web pages increased, the computational effort necessitated a method of distributing the computational load to multiple processing units. Upon completion, the results from individual processors were to be combined with no information loss. This criterion implies that algorithm results should be independent of the number of processing units and the way in which the data are distributed to units. The algorithm output should be the same for any hardware configuration in

B. Steele et al., *Algorithms for Data Science*,
DOI 10.1007/978-3-319-45797-0_4

other words. We're familiar with this property—it was named scalable in Chap. 3. The Google Inc. solution has become known as *MapReduce*.

To use a MapReduce algorithm in a distributed computing environment, more is needed though. A system is needed to partition the data set into subsets, distribute the subsets to processing units (sometimes referred to as *DataNodes)* and to control the processes, and finally transfer the results back from the data nodes to a single master computer (the *NameNode*). These tasks are accomplished by a collection of software programs known as the Hadoop ecosystem.

4.2 The Hadoop Ecosystem

The Apache Software Foundation is an organization that maintains and imposes organization on the development of open-source Hadoop-related projects. The Hadoop architecture accommodates a variety of methods for distributed computing [37]. It can be said then, that Hadoop is a collection of open-source Apache projects managed and controlled by the Apache Software Foundation with the understanding that individuals outside of Apache are important contributors to Hadoop. It should not be assumed that Hadoop developers are necessarily attached to the Apache Software Foundation. Commercial enterprises have also developed implementations of Hadoop aimed at enhancing and streamlining its use.[1]

A cluster is a set of two or more computers connected by a high-speed network. The constituent computers are usually referred to as nodes or hosts, and in the Hadoop ecosystem, the nodes can be classified as either master or worker nodes. The NameNode is the master, and it controls the Hadoop ecosystem through two subsystems, the Hadoop File Distributed System (HDFS) and the resource allocation and management system known as YARN, short for Yet Another Resource Negotiator. Under YARN, the master node assigns computational processes to the worker nodes with the objective of optimally utilizing the cluster resources. Processes being carried out on a DataNode are managed by a secondary manager called a node manager.

4.2.1 The Hadoop Distributed File System

The purpose of the Hadoop Distributed File System is to partition and distribute the data across a cluster as a collection of blocks (i.e., subsets). The client, or user, communicates with HDFS through the application program interface (API). The distribution of data blocks to DataNodes and the moni-

[1] Notable commercial vendors are Cloudera, Hortonworks, and MapR.

toring of DataNodes takes place on a master computer named the *NameNode*. HDFS maintains a directory on the NameNode of all files and locations where data is stored. No data is stored on the NameNode, but instead data blocks are located on the *DataNodes*. The probability of a fatal error occurring on at least one DataNode is intolerably large when the number of nodes is large.[2] Exercise 4.1 shows that the failure of at least one node is more likely than not for large clusters even when the likelihood of failure of any one node is small. Therefore, redundancies are built into the cluster by distributing each data block to several DataNodes. When a node fails, other nodes take over the data processing tasks assigned to the failed node. The primary NameNode is backed up by a secondary NameNode in case the NameNode should fail. Figure 4.1 is a simplified diagram of a Hadoop ecosystem.

The NameNode initiates the transfer of data blocks from the client to the DataNodes after having established that all DataNodes are ready to receive data. For example, Amazon Web Service Elastic MapReduce sets the default block size at 128 MB and usually creates three replicates of each block. For instance, data block 1 may be copied to DataNode A, and copied from A to B, and lastly from B to C. Data block 2 may be copied to B, then B to C, and C to D. Algorithm output is retrieved in a similar fashion.

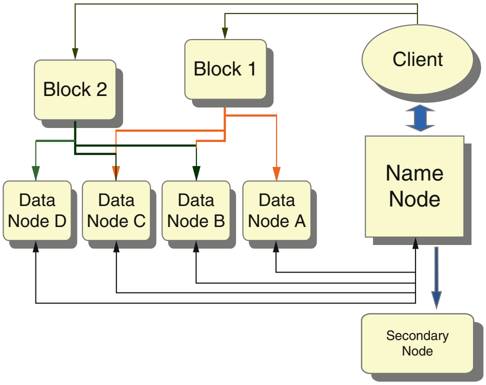


Fig. 4.1 Flow chart showing the transfer of data from the NameNode to the DataNodes in the Hadoop ecosystem

We'll use Amazon's ElasticMapReduce (EMR) for distributed computing. Under this system, the cluster has a short lifespan because it's created specifically for the task submitted by the user. EMR identifies a set of available computers within a very large network of computers and assigns the computers to the cluster. The NameNode takes control of cluster and through Hadoop, orchestrates the execution of the task. When the task has completed, the computers are released back to the network.

[2] A cluster created and controlled by HDFS may consist of thousands of nodes.

4.2.2 MapReduce

Data processing in the Hadoop ecosystem must conform to a particular structure known as *MapReduce*. MapReduce consists of three sequential stages. In the first, the mapper maps the data to a set of key-value pairs. Next, the shuffle stage arranges the key-value pairs so that all instances of a particular key are located on a single DataNode. The third stage is the reducer, wherein the data are reduced to a set of statistics, or lists, tables, or some other form of summarization conceived of by the analyst. The data analyst creates the mapper and reducer for example, as `Python` scripts. The shuffle is executed by Hadoop.

When Hadoop starts, the data are partitioned as g blocks and distributed to DataNodes $N_1, N_2, \ldots, N_g$. The next stage is the beginning of data reduction via MapReduce. In the following sections, we take a closer look at the components of MapReduce.

4.2.3 Mapping

Mapping is the first step of a MapReduce algorithm. The purpose of the mapper is to organize the data for reduction. The organizational structure is closely related to the dictionary structure discussed in Chap. 2. On each DataNode, the data records are mapped to key-value pairs. The keys allow scalability to be built into the MapReduce algorithm, and hence are essential for optimal processing of massively large data sets. The choice of keys are responsibility of the analyst and so must also be consistent with the objectives of data analysis. Looking ahead for a moment, all records associated with a particular key will be reduced together, and no values associated with different keys will be reduced together. So, if the objective is to estimate diabetes prevalence by age class, then age class is an appropriate choice for the keys. If prevalence is also to be estimated by gender, then using age class as a key will lead to significant difficulties. Observations on males will end up on different DataNodes, and computing the prevalence rate for males will require, at best, an inordinate amount of bookkeeping. If the keys are age and gender pairs, then the MapReduce solution to estimating gender-specific rates is relatively simple.

The importance of the keys is manifested in the next stage of MapReduce, the shuffle. The shuffle step moves the key-value pairs around the cluster so that all pairs with a particular key are located on a single DataNode. For example, suppose that k_i is the ith key, for $i = 1, \ldots, g$, and that there are n_i observations associated with k_i. Every one of these n_i records will be written to a particular DataNode. The reducer will reduce all n_i observations associated k_i according to the instructions coded by the analyst in the reducer algorithm. As long the data analyst intended to generate key-specific output, say, a sample mean and standard deviation for group i, for $i = 1, \ldots, g$, there will be no information loss associated with the distributed computing solution. If some other value associated with a different key is needed for the group i computation, then the algorithm is likely to fail.

For example, the tutorial of Sect. 7.3 guides the reader through estimation of diabetes incidence for each U.S. state.[3] Incidence is the annual rate of change in diabetes prevalence among adults. Estimation is accomplished by setting the keys to be the states. There are several choices for the values. For example, the values may be a pair consisting of the binary indicator of diabetes for a particular respondent and the observation year. Alternatively, the mapper may produce keys that identify the state and county. The first example generates 52 unique keys (one for each of the 50 states, Puerto Rico, and the District of Columbia). The second example generates about 3100 unique keys, one per county. The number of records generated by the mappers across all data blocks is the same as the total number of records (assuming that the mapper did not eliminate any records because of invalid data values). Using county as part of the key forces the analyst to generate county-specific estimates. The reducer will not be able to compute a single estimate for each state since observations from the same state, but different counties may end up on different DataNodes after the shuffle. However, county-specific estimates can be aggregated to compute state estimates at the later time provided that the analyst plans for this aggregation.

The mapper may also reduce the data if the reduction generates an associative statistic. For example, if the keys are states, then the values may be a list of three integers: the year, the number of sampled individuals, and the number of sampled diabetics. An example of a key-value pair generated on a particular DataNode is $(k_1, v_1) = (\texttt{AK}, [2000, 1002, 77])$. We deduce that the data block residing on the DataNode contained observations on 1002 Alaskan individuals in year 2000 and 77 of these were diabetics. If there was one other key-value pair generated from Alaska on a different DataNode, say $(k_2, v_2) = (\texttt{AK}, [2000, 1310, 62])$, then the reducer would compute the prevalence estimate for Alaska in year 2000 to be $(77 + 62)/(1002 + 1310) = .060$.

The actual output of the mapper program would be formatted differently since the shuffle algorithm expects that the delimiter between the key and value to be the tab character. For instance, an output record of the mapper algorithm would be `AK \t 2000,1002,77` (`\t` is the tab character).

The reducer algorithm would compute the least squares estimates given the model

$$\pi_{i,j} = \beta_{0,i} + \beta_{1,i}\text{year}_j, \tag{4.1}$$

where $\pi_{i,j}$ is the true prevalence of diabetes in state i and year j, and $\beta_{0,i}$ and $\beta_{1,i}$ are the intercept and the rate of change (or incidence) for state i. This analysis is discussed in detail in Sect. 7.2.2.

[3] Specifically, the 50 U.S. states, District of Columbia, and Puerto Rico.

The mapper often performs other operations besides building the key-value pairs. For example, the mapper may discard incomplete records and ignore variables that are not used by the reducer algorithm.

4.2.4 Reduction

The role of the reducer algorithm is to reduce the data to a useful form for interpretation. Commonly, the reducer output is a set of summary statistics for each key. The diabetes incidence example of Sect. 4.2.3 illustrates. In that situation, the reducer will compute the least squares estimate of incidence $\widehat{\beta}_{i,1}$ for each state (Eq. (4.1)).

An example of how the MapReduce algorithm may be used when there are no natural divisions by which to form keys supposes that a single least squares model of diabetes incidence is to be fit using all of the observations. The MapReduce algorithm is to compute the least squares estimator of the parameter vector $\boldsymbol{\beta}$ using the associative statistics $\mathbf{A}$ and $\mathbf{z}$ according to $\widehat{\boldsymbol{\beta}} = \mathbf{A}^{-1}\mathbf{z}$ (formula (3.30)). Suppose that that $\mathbf{A}$ and $\mathbf{z}$ must be computed in a distributed environment. Now, the state origin of an observation is irrelevant for the intended use of the data. But keys are needed for the shuffle, so as the mapper reads the data file, we maintain a record counter n and set the key for the nth record to be $k_n = n \bmod 10$. Thus, the keys will be values in the set $K = \{0, 1, \ldots, 9\}$ and the keys partition the data set into ten subsets roughly equal with respect to numbers of records. The shuffle will direct all observations with key k_n to a single DataNode, say N_j. The reducer algorithm operating on node N_j will compute the associative statistic $\mathbf{t}(D_j) = (\mathbf{A}_j, \mathbf{z}_j)$ where

$$\mathbf{A}_j = \sum_{(y_i,\mathbf{x}_i)\in D_j} \mathbf{x}_i\mathbf{x}_i^T \text{ and } \mathbf{z}_j = \sum_{(y_i,\mathbf{x}_i)\in D_j} y_i\mathbf{x}_i, \tag{4.2}$$

for $j = 1, \ldots, r$, where r is the number of DataNodes. (See Eqs. (3.22) and (3.32)). We cannot predict that the number of DataNodes will be K since some keys may be shuffled to the same DataNode. We can say that $r \leq 10$, however. The last phase of the analysis occurs after the MapReduce algorithm has completed. The reducer output $\mathbf{t}(D_1), \ldots, \mathbf{t}(D_r)$ is used to compute the least squares estimate

$$\widehat{\boldsymbol{\beta}} = \left(\sum_{j=1}^r \mathbf{A}_j\right)^{-1} \sum_{j=1}^r \mathbf{z}_j = \mathbf{A}^{-1}\mathbf{z}. \tag{4.3}$$

The incidence estimate computed using all states is $\widehat{\beta}_1$. As a final remark, *any* labeling with sufficiently many unique labels could be used as keys. It's critical, though, that the statistic being computed by the reducer is associative when there are no natural divisions of the data. Otherwise, there is no assurance that the reducer output can be aggregated in a well-determined, optimal fashion. If this is the case, then the algorithm is not scalable even though Hadoop and MapReduce are used. It follows that a MapReduce algorithm is not necessarily a scalable algorithm.

4.3 Developing a Hadoop Application

We take a two stage approach to using Hadoop for distributed computing. The first stage produces the MapReduce code in the guise of mapper and reducer `Python` programs. Programming the mapper and reducer is done in a single local-host environment. A standalone computer constitutes a local-host environment. The programs may be coded in an integrated development environment (IDE) such as `Spyder`, though toward the end of the development cycle, the programs must be modified to operate through a command-line interface. The command-line interface is not particularly difficult to use, but it is different from the familiar graphic user interfaces that most of us are used to. The tutorial of Sect. 4.6 helps the reader with gaining some familiarity with the command line.

The second stage of using Hadoop sets up and runs the mapper and reducer programs in a Hadoop distributed computing environment. We'll use Amazon Web Services because of its wide availability and stable user interface. Details are deferred until the tutorial of Sect. 4.7. We digress in the next section to describe a publicly available data source of some importance in the healthcare domain.

4.4 Medicare Payments

Medicare is the federal health insurance program administered by the Centers for Medicare and Medicaid Services (CMS) for individuals that are 65 years or older. Younger individuals with certain disabilities or diseases are also enrolled. Medicaid is a similar health insurance program for individuals with limited income and resources, and so Medicaid participants are generally younger than Medicare participants. In 2010, 48 million people were enrolled in Medicare. Of these, 40 million were 65 years or older in age and the remainder were younger than 65.

Medicare and Medicaid expenditures are enormous. CMS reported expenditures of \$618.7 billion in 2014, amounting to approximately 20% of the estimated total national health expenditures.[4] For comparison, private health insurance expenditures were estimated to be \$991.0 billion—an estimate that is *less* than the combined Medicare and Medicaid estimated expenditure [12].

[4] Medicaid expenditures were \$495.8 billion.

To their credit, the Centers for Medicare and Medicaid Services is actively engaged in providing data to the public on expenditures. We'll investigate geographic variability in Medicare payments by analyzing one such public data set, the *Physician and Other Supplier Public Use File*. These data contain information on provider charges and CMS reimbursements to healthcare professionals for medical services and procedures provided to Medicare beneficiaries. Attributes of particular interest besides the payments and charges are the provider name and zip code, the Healthcare Common Procedure Codes (HCPC), and the narrative descriptions of the procedure codes. Data files are available for years 2012, 2013, and 2014.[5]

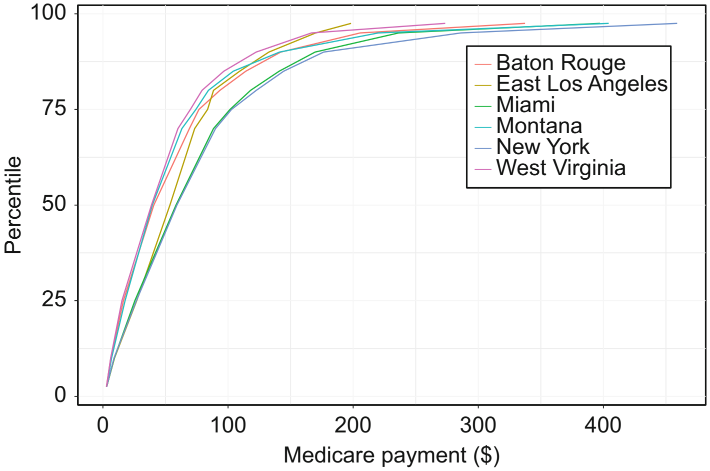


Fig. 4.2 The distribution of average medicare payments for 5 three-digit zip codes

Figure 4.2 shows the distribution of Medicare payments made to physicians and other providers for the years 2012 and 2013 in several areas. We see that the median (i.e., the 50th percentile) of payments made in the Miami and New York City areas were approximately $60, approximately $40 in Montana, Baton Rouge, and south-central West Virginia. Regional variation in payments is not unexpected as some of the factors that influence medical costs vary regionally, for instance, rent. Others have observed greater degrees of variation in Medicare payments [59]. Since the U.S. Centers for Medicare and Medicaid Services are responsible for equitably reimbursing providers for the costs of services provided to their clients, the reader will attempt to corroborate the existence of differences in Medicare payments made to providers by examining the distribution of payments in several areas besides those shown in Fig. 4.2.

[5] There are minor formatting differences between year 2014 data file and the data files for the years 2012 and 2013.

4.5 The Command Line Environment

An additional challenge of negotiating a new and sometimes enigmatic environment presents itself in the next tutorial. This environment is often called the command line. It has an enthusiastic if small following among data scientists. We work in the command line environment because it is through the command line that mapper and reducer algorithms interface with Hadoop. By running the mapper and reducer programs from the command line, we can identify and correct errors that are expensive and more time-consuming to correct in a distributed computing environment. The `Python` scripts of the tutorial differ from all other `Python` programs encountered thus far in this book in two respects. First, the scripts are launched from the command line, and second, data files are handled differently.

Up to this point in the text, we've assumed that the reader has been writing `Python` scripts in an integrated development environment such as `Sypder`. These environments automate the process of submitting a script for execution by transparently passing instructions to the operating system. Writing and debugging scripts tends to be easier and faster than the alternative: editing the script using a text editor and executing the script from the command line. Now, we will execute scripts from the command line. The user may use an IDE for editing the script. There's an advantage to using an IDE with an editor that checks for syntax errors and other helpful features such as automatic indentation control.

At the command line, the user and Linux operating system interact through a shell program. The default Linux shell is the *Bourne Again Shell*, known as `bash`. Windows too has a shell, imaginatively called the Windows shell. We use the shell by opening a Linux terminal window or a Windows Command Prompt window as the case may be, and typing and submitting commands from the command line. We discuss the command line instructions necessary to carry out the tutorial but little more. The reader interested in learning more about the command line environment is directed to *Data Science at the Command Line* [30].

4.6 Tutorial: Programming a MapReduce Algorithm

The objective of this tutorial is to build a MapReduce algorithm that will summarize the distribution of Medicare payments for one or more three-digit zip code regions. A thorough analysis itself is quite involved as there are approximately 900 three-digit zipcode regions. Since the main focus is the algorithm itself, we will use the MapReduce algorithm only to compare a handful of subjectively selected zip codes as a preliminary, first-look at regional variation. The reader is to extract six zip code areas that vary substantially with respect to annual income of residents, population density, and demographic characteristics of their residents. The areas, named after the United States Postal Service facility that serves each area are shown in Table 4.1. The reader may of course add their own zipcode region or any other regions to the list. The Wikipedia webpage `https://en.wikipedia.org/wiki/List_of_ZIP_code_prefixes` describes the three-digit zipcode regions of the United States.

Table 4.1 Some well-known three-digit zip code prefixes and the name of the USPS Sectional Center Facility that serves the zip code area [68]

Location	Three-digit zip code
Boulder, CO	803
Detroit, MI	482
Stamford, CT	069
Santa Monica, CA	904
Stockton, CA	952
Wolf Point, MT	592

To review, writing the MapReduce algorithm is stage one of a two stage process of data reduction via distributed computing and Hadoop. In practice, it's efficient and often necessary to develop the MapReduce algorithm using a subset of the full data set since stage one of our protocol is carried out in a local-host environment with relatively limited computational resources. The second stage, which is tackled in the following tutorial, moves the MapReduce algorithm in a Hadoop environment unconstrained by resource limitations.

1. Gain access to the shell by opening a terminal.
   a. In Linux, you may open a terminal window from the Application Launcher by typing `terminal` in the `Search` box. The terminal will open at the root directory. Type `pwd` (followed by the `Enter` key) to show the current directory. We will work in the home directory, so change the current directory by typing `cd /home` followed by `Enter`.
   b. In Windows, you can click on the start button or press the Windows key on the keyboard. Type `command prompt` in the search box and press the `Enter` key.
2. Examine the home directory.
   a. In Linux, type `ls -l` (list files and directories), and compare to the list shown by the file manager.
   b. In Windows, type `dir` at the command line and press the enter key. Compare to the list displayed by `Windows Explorer`.
3. Create a top-level directory and two sub-directories named `Data` and `PythonScripts`. Issue the instruction `mkdir MapReduce` to create the top-level directory. Then, move to the directory with `cd MapReduce` and

create the sub-directories named Data and PythonScripts. The instruction rmdir Data will remove the directory Data. These commands also work in the Windows shell.

4. Navigate to the Centers for Medicare & Medicaid Services webpage

```
https://www.cms.gov/Research-Statistics-Data-and-Systems/
  Statistics-Trends-and-Reports/
  Medicare-Provider-Charge-Data/Physician-and-Other-Supplier.html
```

Follow the link Medicare Physician and Other Supplier Data CY 2013 to the 2013 data file. Under the heading Detailed Data - Tab Delimited Format:, click on the link titled Medicare Physician and Other Supplier PUF, CY2013, Tab Delimited format, agree to the terms of the license, and download the primary data file Medicare_Provider_Util_Payment_PUF_a_CY2013.zip to your data directory.

5. Unzip the file in your Data directory. From the Linux command line, navigate to the data directory using the change directory instruction cd Data. Then, submit the instruction

```
unzip Medicare_Provider_Util_Payment_PUF_CY2013.zip
```

The change directory instruction will fail if you are not in the MapReduce directory. You may give the full path, say, cd /home/.../MapReduce/Data, if you're not in the MapReduce directory to move to the Data directory. The instruction pwd will show the full path at your current location.

6. Examine the structure of the data file by writing the first 20 records to the Linux terminal window:

```
cat Medicare_Provider_Util_Payment_PUF_CY2013.txt | more
```

The function name cat is an abbreviation of *concatenate*. The pipe symbol (|) separates the optional argument more from the function name. Without more, the entire file will be written to the console. Pressing Enter writes the next line to the terminal. The Ctrl-c keystroke combination will terminate cat and return control of the command line to the user. The analogous Windows instruction is

```
type Medicare_Provider_Util_Payment_PUF_CY2013.txt | more
```

7. Return to the CMS data repository (instruction 4) and download the year 2012 data file.

4.6.1 The Mapper

The mapper is programmed as Python script.

8. Open an editor (preferably, a Python IDE) and create an empty script. If your operating system is Linux, then the *first* line of the script must be

```
#!/usr/bin/env python
```

   This line is sometimes called the *shebang*. The character pair #! instructs the Linux interpreter to turn over the execution of the script to the interpreter that follows, specifically, the Python interpreter. The location of the Python interpreter is specified as /usr/bin/env. The first character of the shebang is the Python comment symbol and it’s purpose is to prevent the Python interpreter from attempting to execute the directive that follows the comment. (The directive is intended for the shell and it does not respect the Python comment symbol.)

   The shebang is not necessary if the operating system is Windows.
9. Add instructions to the script to import the sys module. Save the file with the name mapper.py.
10. The standard input, named sys.stdin, is a default file object. Previously, data files were processed by explicitly creating a file object using the open function. One of the arguments was the data file name. Then, the data was read line-by-line by iterating over the file object. The code looked something like this:

```
with open(fileName, ’r’) as f:
    for record in f:
```

   But now, the standard input (sys.stdin) is being used without the creation step. Add the following code segment to your script:

```
for record in sys.stdin:
    variables = record.split(’\t’)
    print(variables)
    sys.exit(’Terminated’)
```

   The data file is tab-delimited and the record.split instruction splits the string record on the tab character. The result is a list of sub-strings named variables, each of which corresponds to a variable in the data set.

   Add the code segment to your script. If you attempt to run the program from an IDE, the program will start and wait for input from sys.stdin which will not be forthcoming. You may have to terminate and restart the IDE.

The program will be run from the command line instead of within an integrated development environment. But before mapper.py may be run at the command line, permission has to be given.

11. Permit mapper.py to execute from the command line (this is necessary under Linux, but not under Windows). Navigate to the directory containing mapper.py. It will be ../MapReduce/PythonScripts if you followed the instructions above. Since you need to be a superuser to change permissions in Linux, the first step is to gain superuser level rights. Superuser refers to a user who has all rights and permissions to modify files and directories and execute programs. A superuser has more privileges and control over the operating system than an ordinary user. A superuser has greater potential for damaging the system. Most users should not take superuser rights until necessary and relinquish them when no longer necessary.

    Type su for superuser at the command line. You'll be asked for the superuser or *root* password. As a superuser, you may give permission to mapper.py to execute by issuing the following instruction from the command line

```
chmod +x mapper.py
```

    If you're in the data directory, then the command chmod +x ../mapper.py will find mapper.py and change its permissions.

12. The mapper program will consume a data file that is piped to the standard input using a command line instruction of the form

```
cat datafile | ./mapper.py
```

    The instruction cat directs bash to begin streaming (or writing) the data file. The pipe symbol directs the data stream to the device following the pipe; in this instance, the device is mapper.py. The symbol pair ./ instructs the Linux interpreter to process the code contained in mapper.py. The shebang in mapper.py instructs the bash interpreter to allow the Python interpreter to take over the process.

    From the PythonScripts directory (containing mapper.py), execute mapper.py using the instruction

```
cat ../Data/Medicare_Provider_Util_Payment_PUF_CY2013.txt
 | ./mapper.py
```

    The script will consume the first record in the data file, print the variables, and terminate when the instruction sys.exit() is processed.

If you’re using Windows, you’ll have to locate the `python.exe` interpreter (sometimes a nontrivial task). It might be best to find the location using the Windows Explorer. Search for `python.exe`. When the interpreter has been located, copy the path, let’s call it `pathToPython`, from the path bar at the top of the window. The command line instruction to execute `mapper` *from the data directory* is almost the same:

```
type Medicare_Provider_Util_Payment_PUF_CY2013.txt
    | pathToPython\python mapper.py
```

13. Process all of the data records. First, introduce a variable to `mapper.py` that counts the number of records, then insert an instruction in the `for` loop that will print the counter every 10,000 records.

    Move the `sys.exit()` statement to the end of the script and outside of the `for` loop.

    Execute `mapper.py` using the instruction

    ```
    cat ../Data/Medicare_Provider_Util_Payment_PUF_CY2013.txt | ./mapper.py
    ```

    The keystroke `Ctrl-c` issued at the command line will terminate the program.
14. There’s two items of information that are necessary for our objectives: the leading three digits of the zip code and the Medicare payment. However, let’s also reconstruct the names of the providers. The provider name has to be built as a single string from the first and last name and the middle initial, if there is a middle initial.

    ```
    try:
        lastName = variables[1]
        firstName = variables[2]
        middleInitial = variables[3]
        provider = lastName + ’_’ + firstName + ’_’ + middleInitial
        zipcode = variables[10][:3]
        payment = round(float(variables[26]), 2)
    except(ValueError):
        pass
    ```

    The underscore character is used to separate the first and last name and the middle initial.
15. A valid record should be written as an output string according to

    ```
    print(zipcode + ’\t’ + provider + ’|’ + str(payment))
    ```

The print instruction belongs within the `try` branch of the exception handler after extracting the payment variable.

The tab delimiter between zip code and provider is critical since it separates the Hadoop key from the value (zip code is the key and the value consists of the provider and payment pair). If you're writing code for Hadoop, the tab character should not be used except to delimit keys and values.

The print statement will direct the string constructed from the zip code, provider, and payment to the standard output. Instruction 12 did not specify an output file for the output stream created by the print statement. Consequently, the output was directed to the terminal window. If we had extended the instruction by adding `> output.txt` to the command, then the output would have been directed to the file `output.txt` and overwritten its contents.

When no further data is received from the standard input, iteration ends and the program terminates.

16. Run the program from the command line using the appropriate instruction shown in item 12. If the output is correct, then execute the program again and this time, pipe the output to a file:

```
cat ../Data/Medicare_Provider_Util_Payment_PUF_CY2013.txt
         | ./mapper.py > ../Data/mapperOut.txt
```

One should be careful with the `>` character since the target (in this case `mapperOut.txt`) will be overwritten.

17. If both the year 2012 and 2013 data files were to be processed by the mapper, then the Linux instruction would be of the form `cat datafile1 datafile2 | ./mapper.py`. The Windows instruction would be of the form `type datafile1 datafile2 | pathToPython \python mapper.py`.
18. Examine the output file:

```
cat ../Data/mapperOut.txt | more
```

When the mapper and reducer programs are used with Hadoop in a distributed environment, the shuffle step takes place after the mapper executes and before the reducer commences execution. The shuffle will direct all output records with key k_i to one DataNode. It should be assumed that records with key k_j are completely inaccessible to the reducer that is processing the records with key k_i, assuming that $i \neq j$.

4.6.2 The Reducer

The next stage of developing the MapReduce algorithm is to program the data reduction map, otherwise known as the reducer. The output of the reducer is a list of percentiles for any set of user-selected three-digit zip codes.[6] To compute a percentile for zip code z, we need a list containing all of the payments that were made to a provider with an address in the zip code. A dictionary is convenient for building lists of payments from each selected zip code. The dictionary keys will be three-digit zip codes and the values will be payment lists.

We can use a container type from the `Python` module `collections` to simplify the code and thereby reduce the likelihood of a programming error. Specifically, we use the dictionary type `defaultdict` and use the attribute `setdefault` when appending a payment to a payment list. Using `defaultdict` avoids a test of whether the zip code has already been entered as a key in the dictionary before appending the payment to the zip code payment list.

19. Begin by creating a script named `reducer.py`. The program will read the output of the mapper from the standard input. Enter the shebang on the first line if your operating system is Linux.
20. Add the instruction to import `sys`. Import the function `defaultdict` from the module `collections`. Import `numpy` as `np`. Lastly, import the `pyplot` functions from `mathplotlib` as `plt`:

```
import matplotlib.pyplot as plt
```

21. Initialize the dictionary using the instruction

```
zipcodeDict = defaultdict(list)
```

The values in `zipcodeDict` are to be lists. If instead we had specified `defaultdict(set)`, then the values would be sets.

22. Add the statement `sys.exit('Complete')` to the end of the script.
23. If necessary, instruct the shell to allow `reducer.py` to execute as was done for the mapper in instruction 11. Execute the program using the shell instruction

```
./reducer.py
```

[6] Recall that p% of a distribution is smaller in value than the pth percentile and $(100-p)$% of the distribution is larger. Thus, the median is the 50th percentile.

If you're not in the same directory as the reducer program, then the path to the file will have to be added to the file name.

24. Add the following code segment to read the output file created by `mapper.py` from the standard input. As records are read, split the records on the tab character, thereby creating two variables: the key and a list containing the provider name and the payment. Then, split the list into the provider name and the payment amount:

```
for record in sys.stdin:
    zipcode, data = record.replace('\n', '').split('\t')
    provider, paymentString = data.split('|')
```

25. Test the code by adding a statement to print the name of the provider. Execute the code from the command line:

```
cat ../Data/mapperOut.txt | ./reducer.py
```

    If the script prints the provider names correctly, then comment out or remove the print statement.

26. Convert `paymentString` to a floating point variable named `payment`. Add a statement to print `payment` and test the script by running it.
27. Store the `payment` in `zipcodeDict` according to the three-digit zip code using the `setdefault` function. The instruction is

```
zipcodeDict.setdefault(zipcode,[]).append(payment)
```

    The argument pair `(zipcode,[])` passed to `setdefault` instructs the `Python` interpreter to initialize the value associated with the key `zipcode` to be an empty list if there is no entry in the dictionary for `zipcode`. The float `payment` is appended to the list after testing for the key and perhaps adding the key and an empty list value to `zipcodeDict`.

28. Add a variable `n` to count the number of records read from the standard input.
29. Add the instructions to print the length of `zipcodeDict` whenever `n` mod 10,000 is equal to zero. Test the code by executing the script. The length of `zipcodeDict` should be approximately 900.
30. At this point in the program execution, all of the data has been read from the standard input. The next stage is to reduce the data contained in `zipcodeDict`.

    First, create a set, call it `shortSet` containing the three-digit zip codes in Table 4.1 and any others that you are curious about.

31. Create a list of values corresponding to the percentile values of interest and a list containing colors, one for each of the selected three-digit zip codes:

    ```
    p = [5, 10, 25, 50, 70, 75, 85, 90, 95]
    colorList = ['black', 'red', 'blue', 'yellow', 'green', 'purple']
    ```

    Put this code segment near the beginning of the script (and definitely not in a `for` loop).
32. We'll iterate over the zip codes in `shortSet` and extract the list of payments for each of the three-digit zip codes. The list is passed to the `Numpy` function that extracts percentiles.[7]

    ```
    for i, zipC in enumerate(shortSet):
        payment = zipcodeDict[zipC]
        print(i, zipC, len(payment))
        percentiles = np.percentile(payment, p)
    ```

    The `enumerate` function generates an index `i` as the `for` loop iterates. When the first item in `zipc` is extracted, `i` $= 0$, when the second item in `zipc` is extracted, `i` $= 1$, and so on. This code segment executes after creating `colorList` (instruction 31). The returned array `percentiles` contains the percentiles $x_5, \ldots, x_{95}$.
33. Print the percentiles and construct a plot that resembles Fig. 4.2. Therefore, plot x_p against p for each of the three-digit zip code areas on a single figure by inserting the function call

    ```
    plt.plot(percentiles, p, color = colorList[i])
    ```

    in the `for` loop shown in instruction 32. This instruction executes immediately after the percentiles are computed.
34. Create a file containing the plot using the instruction

    ```
    plt.savefig('percentiles.pdf')
    ```

    This instruction executes once after all the zip codes in `shortSet` have been processed, in other words *after* the `for` loop as completed.
35. The mapper and reducer programs will be moved to a Hadoop environment in the next tutorial. In that environment, the plot cannot be

[7] The `percentile` function will compute percentiles from any object that can be converted to an array by `Numpy`.

created and instead we write the array of percentiles to a file that has been assigned to the standard output. These data would then be read into Python or some other program or language that can be used to create the plot. Add the instructions to write the percentiles to the standard output:

```
pList = [str(pc) for pc in percentiles]
print(zipC + ',' + ','.join(pList))
```

36. Check that the only print statement remaining in the reducer program is writing the percentiles to the standard output. Remove any other remaining print statements. Remove or comment out the two plotting function calls.
37. The output from the mapper can be streamed directly into the reducer program thereby eliminating the need for creating the intermediate output file. The instruction, assuming that Medicare provider data from years 2012 and 2013 are to be analyzed, is

```
cat ../Data/Medicare_Provider_Util_Payment_PUF_CY2013.txt
    ../Data/Medicare_Provider_Util_Payment_PUF_CY2012.txt
    | ./mapper.py | ./reducer.py > output.txt
```

Test the mapper and reducer combination by submitting this instruction from the command line. Check that the output file contains the percentiles.

4.6.3 Synopsis

We've used a local host environment for the purpose of creating the MapReduce algorithm. The mapper and reducer are written in Python. Both the environment and the language were chosen for convenience. Other scripting languages, for instance, Java and Ruby, may be used to write the mapper and reducer programs. The Hadoop utility Streaming will accommodate almost all scripting languages that have the facility of using the standard input and output for reading and writing.

Execution time and computational resources might be strained by the Medicare data sets depending on the configuration of the local host. The computational load can be reduced by processing only a fraction of the 18,441,152 records with complete data.[8] Instead of simply programming the mapper to

[8] A conditional statement such as `if m % 3 ==0` may be used to select every third record for processing.

produce key-value pairs and the reducer to aggregate according to key, another approach is to run the mapper and reducer programs under a local host installation of Hadoop. A local host installation of Hadoop resides on a single computer. Since the data and computational load are not distributed across a network, there's no computational benefit to the local host Hadoop environment. The reason for using the local host Hadoop is that the environment is very similar to the distributed network environment, thereby achieving a greater level of realism in the first stage of development. We do not take this approach because the process of installing Hadoop differs among operating systems and is technically challenging.

The next tutorial provides the reader with the opportunity to use Hadoop in a distributed computing environment hosted by Amazon Web Services.

4.7 Tutorial: Using Amazon Web Services

Amazon Web Services is a relatively painless way to carry out distributed computing. In this tutorial, we'll use two services available from Amazon Web Services to gain access to distributed computing and the Hadoop ecosystem. These are S3 (Simple Storage Service), a cloud-based data storage system, and Elastic MapReduce which in essence provides a front-end to Hadoop. S3 stores the input data, log files, programs, and the output files. The Elastic MapReduce service provides the `Hadoop` ecosystem and true distributed computing. Furthermore, the graphical user interface of Elastic MapReduce streamlines the process of setting up a Hadoop cluster. In this context, a cluster is a network of computers connected for the purpose of executing a MapReduce algorithm. Through the Elastic MapReduce interface, the user will issue instructions to create a cluster and perhaps select options related to the cluster. Elastic MapReduce sets up the NameNode and DataNodes that form the `Hadoop` cluster. The NameNode and DataNodes are not physical computers, though we've referred to them as such, but instead are virtual machines, or *instances.* A virtual machine is a computer within a computer. It has its own operating system, dedicated memory, and processing units. Virtual machines can be quickly created and terminated. Once the instances are available, the two programs that constitute the core of Hadoop, HDFS and YARN (discussed in Sects. 4.2 and 4.2.1) distribute of data blocks and mapper and reducer programs across the instances, start, and control the execution of the mappers and reducers. Amazon Web Services provides several webpages that track the progress of the Hadoop cluster and allows the user to determine when the cluster has completed and whether the cluster has completed without errors. Non-fatal errors within the mapper and reducer are of course undetectable by Hadoop.

Amazon does not provide the service for free and you should expect to compensate Amazon for the use of their services.[9] You will need an account with Amazon.com; if you have ever purchased something from Amazon, you probably have an account. It's strongly recommended that the mapper and reducer programs are developed to the greatest possible extent in a local host environment to save time and money.

The first steps are to upload programs and data to S3. Begin by creating buckets (directories) to store the input data, log files, and mapper and reducer programs.

1. Navigate to `https://aws.amazon.com/console/` and sign in to the Amazon Web Services console or create an account.
2. Find the heading `Storage & Content Delivery` and click on the `S3` icon. Click on `Create Bucket`, enter a name, and select a region. Remember the region name since it ought to be used when launching the Hadoop cluster. Click on `Create`. Navigate to the newly-created bucket and create three directories in the bucket.

   a. Log files generated by EMR are to be stored in a directory called `logfiles`.
   b. Create a directory named `Data`. Move into the directory. Using the `Actions` drop-down menu, upload the two data files
      `Medicare_Provider_Util_Payment_PUF_CY2012.txt`
      `Medicare_Provider_Util_Payment_PUF_CY2013.txt`
      into the directory. Choose `Upload`, select a file, and click on `Upload now`.
   c. Create a directory named `programs` and upload your mapper and reducer `Python` programs into the directory.

   Do not create an output directory for the output of the cluster.
3. Set up the cluster by returning to the `Services` page and selecting `Analytics` and then `EMR`. Then select `Create Cluster`.
4. Under the top section named `General Configuration`, specify a cluster name or accept the default.
5. In the `Software configuration` panel, select Amazon as the `vendor` and the current release of Elastic MapReduce (`emr-4.6.0` at the time of this writing). Select `Core Hadoop` as the application. Accept the default version of Hadoop.[10] Accept the default hardware configuration and the default security and access options.
6. At the top of the page is a link to more options titled `go to advanced options`. Several options must be set to form the cluster, so follow the link.

[9] It's possible that the reader may be able to obtain an academic discount or free trial.

[10] The default version was 2.7.2 at the time of this writing.

In the `Software Configuration` field, turn off all of the software options except for `Hadoop`.[11] Don't edit the `software setting`.

7. In the `Add Steps` panel, select the step type to be `Streaming program`. This selection allows the mapper and reducer programming language to be `Python`.
8. Go to the `Add Steps` webpage by clicking on `Configure`. In the `Configure` panel, specify where the program and data files are located. Also specify the name of a non-existent output directory. EMR will create the output directory with the name and path that you specify. The output of the reducer programs will appear in this directory. If it already exists, the cluster will terminate with errors.
9. Identify the mapper and reducer programs and their location as follows. To select the mapper program, click on the folder icon and navigate to the directory containing the file and select the file. Repeat to select the reducer program. Do the same with the data file except that the selection should be the name of the directory and not the data file name(s). HDFS will process each file in the data directory.
10. Identify the output directory as follows. Select the top-level directory. Supposing that you have named the `S3` bucket containing the programs and data as `Medicare`, then the top-level directory ought to be `s3://Medicare/`. Next, append a name for the output directory to the name of the top-level directory. For example, we set the output directory to be `s3://Medicare/output/`. Note: if the output directory already exists, then the cluster will terminate with errors. Therefore, if you execute the MapReduce configuration more than once, you must delete the output directory after each execution or specify a new name for the output directory. This new directory must not exist in the `s3://Medicare` directory.
11. The last setting to modify on the `Add Steps` page is the `Action on failure` setting. Select `Terminate cluster`. If the cluster is not terminated, then the resources allocated to the cluster are not released and you will have to pay for the resources even if they are not in use. As the charge depends on how long the resources are held, it's important to terminate the cluster as soon as possible.
12. Review the `Add steps` arguments. Make sure that the `Auto-terminate cluster after the last step is completed` option has been selected.

    If the selections appear to be correct, then you may step through the remaining advanced options without modifying the default settings until you reach the last page. On the last page there is a `Create cluster` button that will create the cluster. Alternatively, you can skip the remaining advanced options pages by clicking on the button titled `Go to quick options` which leads to the `Create Cluster-Quick Options` page. At

[11] At the time of this writing, `Hue`, `Pig` and `Hive` are included in the default configuration. These programs are not necessary.

the bottom of this page, there is a Create cluster button that will create the cluster.

13. Create the cluster. Depending on the time of day (and hence, the activity load from other users), the execution time will be between 5 and 30 min. If you pass the 1 h mark, terminate the cluster. Look for errors.
14. To diagnose a termination error, locate the Cluster Details page and expand the Steps item. Two steps are shown: Streaming program and Setup hadoop debugging. Expanding the Streaming program step reveals the arguments sent to HDFS. In particular, the names of programs and the data and output directories are identified. The specific names of mapper and reducer programs are listed. The format is

```
Main class:None
   hadoop-streaming -files (path to mapper), (path to reducer),
     -mapper (name of mapper)
     -reducer (name of reducer)
     -input (path to data directory)
     -output
Arguments (output path)
```

The paths and names in parentheses above will depend on your cluster. Check that the names and paths match the structure in S3. Whatever you entered when setting up the cluster will appear where indicated by the parentheses. The output path listed after the Arguments keyword will match what was specified for the output directory in item 10.

Information about the execution of the cluster is shown under the heading Log files. The syslog provides information about failures and has been most useful to us.

15. If the cluster has completed successfully, then return to S3 and examine the contents of the output directory. The output of the reducer is contained in files named part-0000x where x$\in \{0, 1, \ldots, 6\}$. (Your output may differ). Several of the files may be empty but several will contain records listing the percentiles from one or a few zip codes. For example, our cluster generated the following content for file part00001:

```
708,2.68,6.44,16.17,40.6,68.78,76.76,94.43,114.86,143.77,212.23,348.7
127,2.67,6.7,14.52,34.13,64.35,75.42,83.99,104.8,130.41,171.29,245.23
```

The shuffle directed all records with the keys 708 and 127 to the same DataNode. The reducer program processed these records and computed the percentiles for each. The 5th percentile of Medicare payments made to providers in the 708 zip code region is $2.68 and the 95th percentile is $348.70. The percentiles have been computed using all of the records that originated from these zip codes since no other DataNode received records originating from these zip codes.

4.7.1 Closing Remarks

Hadoop rarely is an efficient solution when the data may reside on a single computer without consuming nearly all of the available storage capacity and execution time is not prohibitively long. When the data are not too massive, the standalone computer is preferred to the cluster because of the expense of distributing data blocks across the cluster and building in redundancies. For many infrequent Hadoop users, working in the Hadoop ecosystem requires more of the user's time than the more familiar standalone environment. Most of the time, the first stage of building and testing the mapper and reducer is roughly equivalent to the standalone solution with respect to time expenditure. Therefore, Hadoop is a resource to be utilized when data sets are too massive or computational tasks too time-consuming for a single computer to manage. From a mechanical standpoint, it's not practical to provide the reader with a problem that truly needs a Hadoop environment for analysis. To gain a brief exposure to true distributed computing, this last tutorial has guided the reader through a commercial Hadoop environment, Amazon Elastic MapReduce. But even in this instance, the analysis can be carried out locally and with substantially less human and computational effort.

4.8 Exercises

4.8.1 Conceptual

4.1. Suppose that n is the number of DataNodes of a cluster and each DataNode will fail with probability $p = .001$. Also, assume that failures occur independently. Show that fault tolerance is essential because the likelihood of one or more DataNode failures increases exponentially with the number of DataNodes. Specifically, compute the probability of the event that one or more DataNodes fails if the cluster consists of 1000 DataNodes.

4.8.2 Computational

4.2. In Sect. 4.6.2, we use a `defaultdict` dictionary to build the dictionary of payments. Replace the `defaultdict` dictionary with a conventional dictionary. You'll have to test whether `zipcode` is a key in the dictionary before appending the payment and creating a key-value pair from the zip code and the payment.

4.3. This problem investigates variation in Medicare payments by provider types.

a. For a selection of zip codes, identify the top five provider types with respect to the average paid charge (column 26).

b. For a selection of zip codes, identify the top five provider types with respect to the total paid charge. To compute the total paid charge, compute the weighted sum

$$\sum_i n_i x_i, \tag{4.4}$$

where n_i is the number of distinct Medicare beneficiaries receiving the service (column 20) from the healthcare provider in the ith zip code, and x_i is the estimated average Medicare payment rendered to the provider by CMS (column 26). For a selection of zip codes, identify the top five provider types with respect to the total paid charge.

4.4. Write a MapReduce algorithm for computing the correlation matrix using the variables payment (column 26), amount submitted (column 24), and allowed (column 22). In the mapper, use n to count the number of records processed and define the key to be n mod 100. The value is to be the three values of payment, submitted, and allowed. The reducer is to compute the augmented moment matrix (Eq. (3.23)). Download the reducer output, say, $\mathbf{A}_1, \ldots, \mathbf{A}_r$, and aggregate as $\mathbf{A}$. Then compute the correlation matrix from $\mathbf{A}$. Section 3.7.4 provides an outline of how to proceed.

Part II
Extracting Information from Data

Chapter 5
Data Visualization

The drawing shows me at one glance what might be spread over ten pages in a book.
— Ivan Turgenev, *Fathers and Sons*

Abstract A visual is successful when the information encoded in the data is efficiently transmitted to an audience. Data visualization is the discipline dedicated to the principles and methods of translating data to visual form. In this chapter we discuss the principles that produce successful visualizations. The second section illustrates the principles through examples of best and worst practices. In the final section, we navigate through the construction of our best-example graphics.

5.1 Introduction

Humans are visual animals. We absorb sensory information most efficiently through vision. It's no surprise that data visualization is very effective for extracting information from data. As we see throughout this volume, gathering large amounts of data has never been easier. Even 40 years ago, displaying that information was difficult to do well, requiring specialized tools or a steady hand. The democratization of creating figures from data allows us to create more visualizations than ever before. And with this profusion comes the ability to develop best practices. Figures *encode* information from a data set, displaying those data as ink on a page or, more commonly, as pixels on a screen. This encoding makes use of a vernacular that has developed over the last several centuries. The typical audience will understand the Cartesian plane and its axes. We understand how to determine values of points in a scatterplot and easily manage the color-coding of groups.

B. Steele et al., *Algorithms for Data Science*,
DOI 10.1007/978-3-319-45797-0_5

A visualization is effective when it can be quickly and accurately *decoded* by the audience—the salient points should be almost immediately apparent. Edward Tufte, a vocal evangelist for better graphics, calls this the "Interocular Trauma Test": does the visualization hit one immediately between the eyes? A common expression about scripting languages, like `Python` and `Perl`, is that they should make easy things easy and difficult things possible. The same can be said of good visualizations: the key features of the narrative should emerge immediately and the more subtle relationships should be visible whenever possible.

Data visualization combines several different threads and we will cover each in a section. The first is an understanding of the guiding principles of graphics. In this section we lean heavily on the pioneering work of William Cleveland, the researcher who truly brought graphics into the modern era. Most of the excellent graphics one sees in data science journalism[1] are built using the ideas he introduced. The second is a basic understanding of the paradigms of graphics that are often employed for data of different types. Knowing these will allow us to create an abstract version of the graphic. This ability, to sketch a version of the graphic you wish to create, is critical. It is impossible to follow a map if you don't know where you're going. Finally, we must be able to tell a software package how to render the image that is in our minds. The best tool for creating data visualizations in the context of an analysis[2] is Hadley Wickham's `ggplot2` available in `R` [65, 66]. The "gg" in the package name stands for "The Grammar of Graphics". You can think of this grammar as being a semi-secret language that, for better or worse, you must be fluent in to realize the potential of graphics from data. Recognizing this reality, we will introduce the grammar and illustrate its implementation in `R` syntax.

The data used in this chapter and Chap. 10 originates from the largest cooperative (co-op) grocery store in the United States. We received approximately 20 gigabytes of transaction-level data covering 6 years of store activity. The data are essentially cash register or point-of-sales receipts. As is common with automatically recorded point-of-sales information, a considerable amount of associated meta-data is attached to the receipt. We work a great deal with two variables: the department classification of the item (e.g., produce) and whether the shopper is a member of the co-op. The co-op is member-owned and approximately three-quarters of the transaction records

[1] Three great examples:

- The Upshot from the New York Times: `http://www.nytimes.com/section/upshot`.
- Five Thirty Eight, Nate Silver's organization that has largely invented the field of data science journalism. `http://fivethirtyeight.com`.
- Flowing Data, a site created by Nathan Yau dedicated to creating beautiful and informative data visualizations. `http://flowingdata.com`.

[2] If you are building interactive graphics or large-scale graphics via the web, there are better tools. Check out `bootstrap`, `D3`, and `crossfilter`.

originated from members. The remainder originated from non-members. If the shopper is a member, then a unique, anonymous identifier is attached to receipt that allows us to analyze data at the shopper level.

We use the co-op data in this chapter to illustrate a variety of data visualizations. In Chap. 10, we develop a prediction function that classifies non-member shoppers to customer segment with the ultimate goal of better understanding the non-member shoppers.

5.2 Principles of Data Visualization

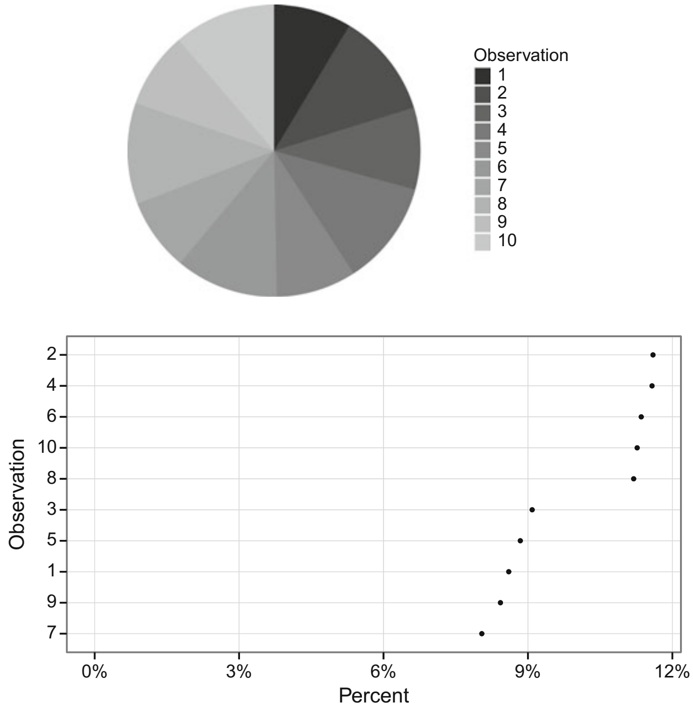
When describing what makes good data visualization, there are two paths to follow: that of simplicity and that of exhaustiveness. There are entire books of the latter variety and thus we opt for a treatment that will give the reader minimal guidelines and that points them in the direction of the more thorough treatments. Our goal, after all, is to be able to make good visualizations and improve on those found in the wild.

A data visualization is effective when the information that the analyst is trying to convey to the audience is transmitted. Complicated information sometimes requires complicated visualizations, but often we can organize our thoughts in terms of principles.

Let the Data Speak. If there is one overarching goal, then this is it: the data should be allowed to speak for itself. As Strunk and White say, "vigorous writing is concise." Similarly, good data visualization allows the data to tell its story. Tufte coined a term for elements of a visualization that do not add to understanding: *chartjunk*. A good, revised definition of chartjunk from Robert Kosara at Tableau is this: any element of a chart that does not contribute to clarifying the intended message. Historically, the most egregious violations of this principle came from Excel. The default settings in Excel now avoid the worst examples of chartjunk, but options to add them abound, particularly with the addition of mysterious third dimensions to one- or two-dimensional data sets, color coding for no reason, and patterns that impede understanding. For each element of a figure, we must ask if that element is serving our principal goal of letting the data speak.

Let the Data Speak Clearly. A subtle addition to our previous point is to let the data tell their story clearly and quickly. As we shall see, the relationship between two variables, often plotted against one another in a scatterplot, can be illuminated by the addition of a smoothed or fitted line. At the other end of the spectrum, Fig. 5.1 illustrates how certain visualizations, in this case the much- and rightly-maligned pie chart can obfuscate the story. In the pie chart version, it's extremely difficult to identify the predominant pattern—the presence of two sets of observations with different means. The lower panel in Fig. 5.1 shows a dotchart which lets the data speak for itself and uses a sensible ordering of observations. This visualization conveys much more information and allows for it to be immediately decoded.

Fig. 5.1 A pie chart makes patterns in the data difficult to decode; the dotchart is an improvement



We introduce the dotchart in Sect. 5.3 and it is an excellent choice for displaying univariate data. There are other types of graphs besides the pie chart that inhibit understanding, notably stacked bar charts and area charts. The stacked bar chart makes it difficult to understand the behavior of the individual elements, as we see in Fig. 5.2. This figure has two views of average sales data by month for a grocery store for four large departments. In the upper panel, we can see that May is the month of maximum overall sales (by looking at the heights of bars and ignoring the colors). We can see that packaged grocery appears to have lower sales around July and that meat and bulk appear to be smaller departments, although it is difficult to gauge the magnitude of the difference. In the lower panel we have replaced the stacked bars with points connected by lines. The points allow precise estimation of individual observations and the lines (and the color) help us group the departments together across time. Now we see that packaged grocery and produce are larger than the other two departments, by a factor of just more than two. Note that the lower panel figure is less than perfect because the vertical axis does not extend to 0. We also see that produce has an annual cycle out of phase with the other departments, peaking in the North American summer months. Area charts such as the pie chart typically arise when circles are scaled based on some variable that would not otherwise be plotted. There are cases where this is useful, for instance, adding sample size to a chart via area scaling. Research indicates that people are able to decode area to perceive order but magnitude is difficult accurately translate from area.

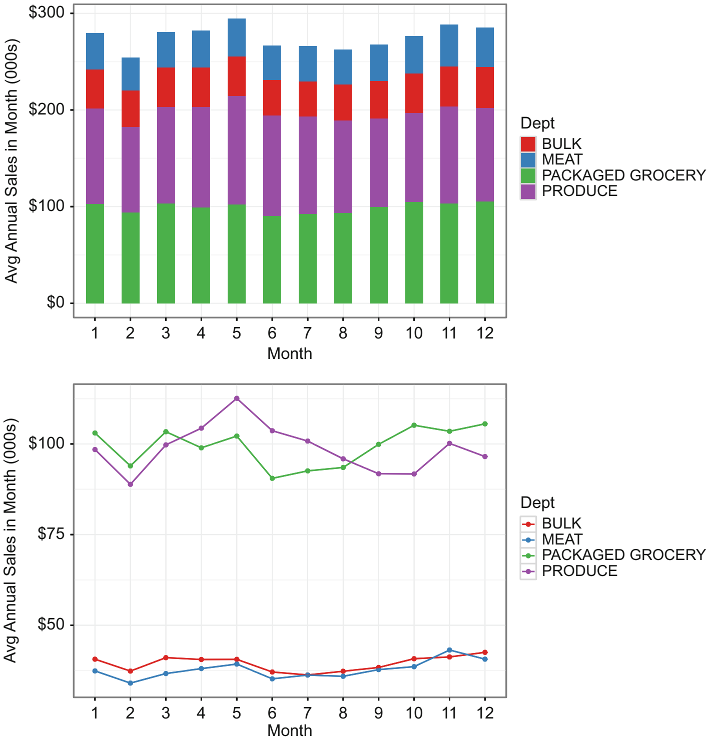


Fig. 5.2 Two views of monthly sales for four departments. The stacked bar chart obfuscates much information that the line chart makes clear

Choose Graphical Elements Judiciously. As one builds graphics, there are many choices: colors, shading, line types, line widths, plotting characters, axes, tick marks, legends, etc. Make these choices thoughtfully. Color is often used well when colors indicate membership according to a categorical variable. Color is often used poorly when practitioners get bored and add color haphazardly. Axes can be used intelligently to highlight certain observations or the range of the data. Smoothing lines can illustrate trends in bivariate data or mistakenly cover up observations. With a good graphics package, like the `R` package `ggplot2` which we introduce in Sect. 5.4, every element of a figure can be manipulated. Take advantage of the opportunities.

Help Your Audience. Wherever possible, make adjustments to your figure that helps your audience better understand the data. A choice had to be made in the lower panel of Fig. 5.1. The default behavior sorts the observations by name. This would be the desired order if our goal was to allow the reader to quickly find a given observation in the list and look up the value.[3] But if this is the goal, a table might be a better choice. By sorting the observations by value we can instantly see the minimum and maximum and the observations that define the break between the two groups. By manipulating the data using the `R` function `reorder`, we help the audience in several ways. Another example of this principle would be to highlight important observations by labeling them.

Limit Your Scope. Most interesting projects generate a profusion of data and, as our tools to visualize that data grow, so does the temptation to try to tell the entirety of a story with a single graphic. A well-written paragraph has a topic sentence and a unifying idea. Applying this concept to your graphics requires discipline and attention to detail. There will be times when you are tempted to add additional elements to a chart that already tells the story. Take care that the new elements do not detract from the central message. The classic example of overreaching is a line chart with two different axes for the lines. If you find yourself building such a chart, you are unlikely to be telling the story with clarity and power.

Armed with these general principles, we are now well positioned in the next section to delve into the types of data that are likely to be encountered. We'll describe the elements that make useful visualizations for univariate, bivariate, and multivariate data. In the subsequent section we learn how to produce these graphics in `R`.

5.3 Making Good Choices

Many of our visualization tasks will be defined by the number and type of variables that we will be plotting. The first and most important distinction is between quantitative and qualitative data. A qualitative, or categorical, variable is coded as a `factor` in `R`. In terms of the number of variables to plot, there are relatively clear approaches when we have one or two variables. Once we move beyond that, there are some general principles in effect but creativity plays a larger and larger role. One must be mindful of the edict to not try to do too much with one chart.

[3] When ordering is a problem, it is often referred to as the "Alabama First!" problem, given how often Alabama ends up at the top of lists that are thoughtlessly put, or left, in alphabetical order. Arrange your lists, like your factors, in an order that makes sense.

5.3.1 Univariate Data

Our goals with univariate quantitative data typically are to understand the distribution of the data. We typically begin describing the center and spread of the distribution and identifying unusual observations, often called *outliers*. More in-depth analyses describe the shape of the distribution, a task that provides more information and requires more effort. The classical starting point for investigating shape is either the histogram (described at length in Chap. 3, Sect. 3.4.2), the empirical density function, or one of several variations on the boxplot. In this first section we work with a grocery-store data set, specifically, sales and items by month and department for 6 years. We begin by looking at the distribution of sales, in units of thousands of dollars.

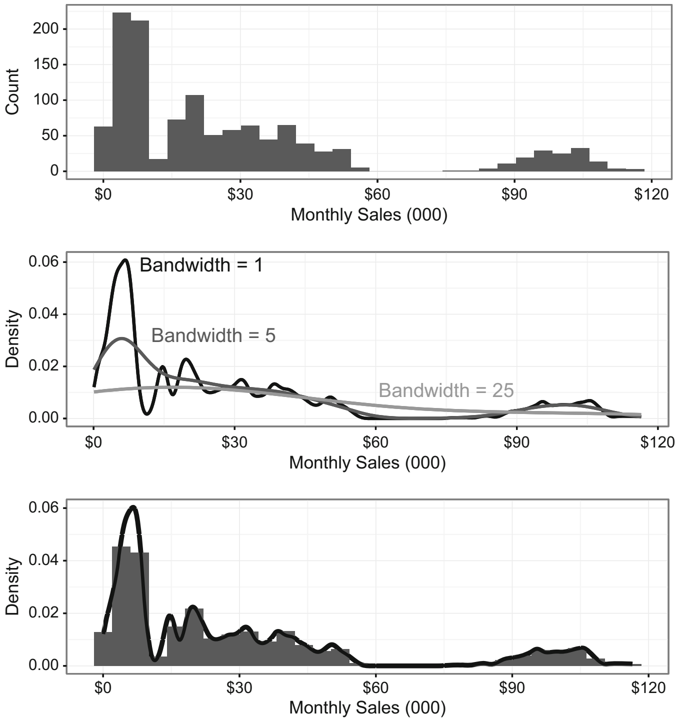


Fig. 5.3 Three different ways of looking at monthly sales by department in the grocery store data: a histogram, several empirical densities illustrating the variations possible from the bandwidth parameter, and a density superimposed on a histogram

The top panel of Fig. 5.3 shows a histogram depicting the numbers of observations falling in each interval, or *bin*, by the height of a vertical bars. We see sales by month by grocery department with bar height representing the count of observations that fall into that “bin”, as the intervals on the x-axis are referred to.

The second panel illustrates a more powerful and complex way to visualize the distribution of a quantitative variable—empirical density functions. An empirical density function is a data-driven estimate of the probability distribution from which the data originated. More simply, they are smooth histograms. The formula that generates the empirical density function

$$\widehat{f}_b(x) = \frac{1}{n \cdot b} \sum_{i=1}^{n} K\left(\frac{x - x_i}{b}\right), \tag{5.1}$$

where n is the number of observations, b is the bandwidth, and K is the kernel.[4] The bandwidth is usually used to control the smoothness of the function by determining the influence of individual differences $x - x_i$ on the function. Setting the bandwidth to a small value, say 1, results in a very bumpy distribution. Setting it to a large value arguably makes the distribution overly smooth and removes the some of the secondary modes visible with $b = 1$. Much like the bin width choice for histograms, some trial and error may be necessary to capture the interesting features of the distribution. The default bandwidth in `R` which chosen by `bw.nrd0` and remains the default for historical reasons. The most recommended choice is `bw.SJ`, based on the method of Sheather and Jones [54]. We show how to set the bandwidth in Sect. 5.4.2.

The middle panel of Fig. 5.3 shows three different bandwidths. The bottom panel shows the histogram and density on the same plot, which necessitated changing the units on the histogram to relative frequency. Note that all depictions show some department-months with low sales, a larger group from $20K to $55K, and a group at $100K.

Another useful way to display distributions uses boxplots or violin plots. These plots, illustrated in Fig. 5.4, are more compact displays of the distribution of monthly sales. Boxplots, first developed by John Tukey in the 1970s, summarize the distribution using five numbers. The box is defined by the first and third quartiles, Q_1 and Q_3, and is split by the median. The interquartile range is $\text{IQR} = Q_3 - Q_1$. The IQR is a useful nonparametric measure of spread. The whiskers, the length of which can be customized in `R`, are by default set to $Q_1 - 1.5 \times \text{IQR}$ and $Q_3 + 1.5 \times \text{IQR}$. Points beyond the whiskers are plotted individually and identified as outliers.

The definition of an outlier may seem arbitrary, but it captures data features in a predictable way if the data are normally distributed. Given a normal distribution, the 25th percentile of the data is $x_{25} = \mu - .67 \times \sigma$ and the IQR

[4] Kernels are an interesting side area of statistics and we will encounter them later in the chapter when we discuss `loess` smoothers. In order for a function to be a kernel, it must integrate to 1 and be symmetric about 0. The kernel is used to average the points in a neighborhood of a given value x. A simple average corresponds to a uniform kernel (all points get the same weight). Most high-performing kernels uses weights that diminish to 0 as you move further from a given x. The Epanechnikov kernel, which drops off with the square of distance and goes to zero outside a neighborhood, can be shown to be optimal with respect to mean square error. Most practitioners use Gaussian kernels, the default in `R`.

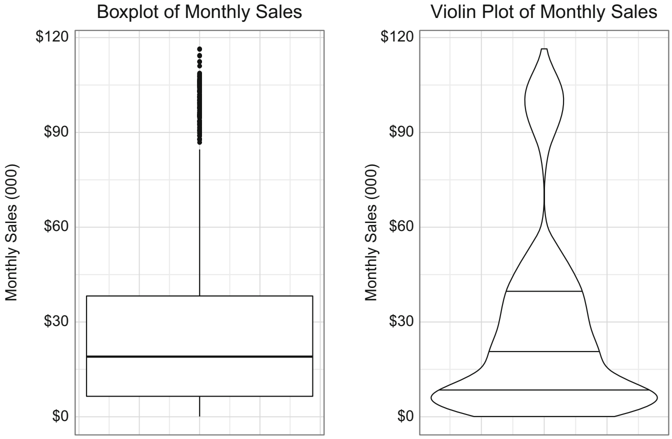


Fig. 5.4 Two different ways of visualizing the distribution of monthly sales numbers, the boxplot and the violin plot

has length 1.34 × σ. Thus, 1.5× IQR $\approx 2\sigma$. The edge of the whiskers is placed at $\pm 2.67\sigma$ units from the mean so we would expect less than .8% of the data to fall outside the whiskers. Note that in our example, many points are plotted individually, indicating that the normal distribution is not an appropriate approximation of the monthly sales distribution.

The violin plot, by contrast, makes use of many more features of the data and can be seen as a boxplot replacement. The plot shows a smooth representation of the distribution turned on its side and gives us a more detailed visualization of the distribution. The width of the figure reflects the density of observations. With the addition of the horizontal lines at the first, second, and third quartiles, there is no loss of information compared to the boxplot. Moreover, the most interesting feature of this distribution, the lack of department-months with sales between $60K and $80K, is obscured by the boxplot but easy to see with the violin plot.

When we have univariate data that is labeled, we have already seen one of the best visualizations: dotcharts. These charts allow us to clearly depict single values and convey the uncertainty around those estimates (where appropriate). In Fig. 5.1 we saw how a dotchart was superior to a pie chart. In Fig. 5.5 we see another example of a dotchart, this time displaying summary statistics. This figure shows monthly spend by department with bars representing the range of values. Note that we have avoided the “Alabama First!” problem by sorting the departments from highest spend to lowest. In this depiction we can clearly see the two largest departments (produce and packaged grocery), the range of mid-sized departments (refrigerated grocery down to cheese), and the smaller departments. We have translated the x-axis variable to the $\log_{10}$ scale. This choice gives us a better view of the monthly sales for the small departments, but can make interpretation a bit trickier for the gray bars, which represent the range. At a first glance, it appears that bakehouse and flowers have by far the widest range in monthly sales. This is true as a percentage; bakehouse ranges over almost an order of magnitude, from a minimum of $1000 to almost $10,000. By contrast, produce has a range of approximately $40,000 and a sample mean of nearly $100,000. An important note: the x-axis displays the values of monthly sales after transforming the values to the $\log_{10}$ scale. The labels, however, show the original, untransformed monthly sales in thousands of dollars. By retaining the labels in dollars, the reader is better able to interpret the variable. The best practice recommended by Cleveland is to place this axis at the bottom of the graphic and the corresponding log-based scale at the top. Unfortunately, `ggplot2` makes this difficult to do, so our labels show five values on the scale of thousands of dollars.

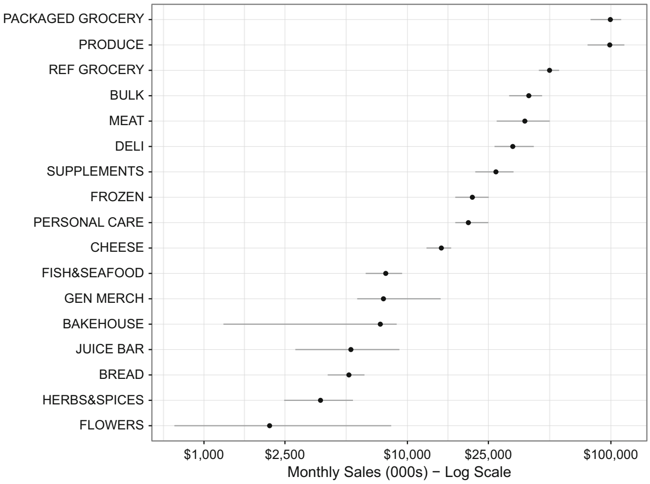


Fig. 5.5 A dotchart of spend by month by department, with bars indicating the range of the data. Monthly sales have been transformed to the $\log_{10}$ scale

The final type of data we might wish to visualize is univariate categorical data. If the number of categories is small, then a simple bar chart is an excellent choice to compare the proportions in each category. If the number of categories is very large, then summarizing with a dotchart as in Fig. 5.5 is often the best practice.

5.3.2 Bivariate and Multivariate Data

Bivariate data is usually straightforward to visualize. There are really three main possibilities: two categorical variables, a quantitative and categorical variable, and two quantitative variables.

Data consisting of all categorical variables typically are summarized by contingency tables and there is no reason to deviate from this practice. A contingency table is a cross-tabulation of observations according to the values of the categorical variables. For instance, if there are a levels of variable A and b levels of variable B, then the table contains $a \times b$ cells and the content of a cell is the number of observations with a particular combination of levels. Table 5.1 is a contingency table cross-classifying receipts according to customer segment and department. We can see general trends by examining the table but specific and fine differences are not immediately discernible.

Table 5.1 Number of receipts cross-classified by department and the three largest customer segments, light, secondary, and primary

	Customer segment		
Department	Light	Secondary	Primary
Supplements	439	55,657	90,017
Cheese	859	96,987	147,679
Frozen	647	97,808	152,940
Meat	653	107,350	149,251
Deli	5830	138,722	155,086
Bulk	2713	144,862	178,520
Refrigerated grocery	3491	194,758	197,463
Produce	3971	211,226	199,978
Packaged grocery	6980	223,815	200,737

There exists, however, a useful visualization that rapidly conveys the relationships between categorical variables, the *mosaic plot*. Figure 5.6 shows a mosaic plot in which the area of the tiles represents the relative number of observations that fall into each cell of the contingency table. We can see, for instance, that there are many more primary shoppers than light shoppers. Packaged grocery is over-represented among light shoppers, whereas primary shoppers make up larger portions of the less popular departments.

This figure allows us to quickly absorb some of the features of the contingency table:

- Primary and secondary shoppers make up the majority of the observations.
- The four largest departments—produce, packaged grocery, refrigerated grocery, and bulk—represent about half the activity of the primary shoppers. Secondary shoppers used those departments to greater extent than primary shoppers and light shoppers used those departments to an even greater extent.
- Supplements represents a small fraction of the purchases of all segments.
- Primary shoppers tend to shop more of the store's departments, generally.

A virtue of the mosaic plot is that it allows estimation of the strength of the relationship between the categorical variables. A downside is that statistical features, such as confidence intervals for the difference in sizes, cannot be displayed. Moreover, the display becomes unwieldy beyond two dimensions. Figure 5.7 shows a mosaic plot built from data generated from two independent random variables with discrete uniform distributions. The plot shows no evidence of association. Conditioning on one variable shows approximately equal-sized rectangles as you travel across a row or down a column.

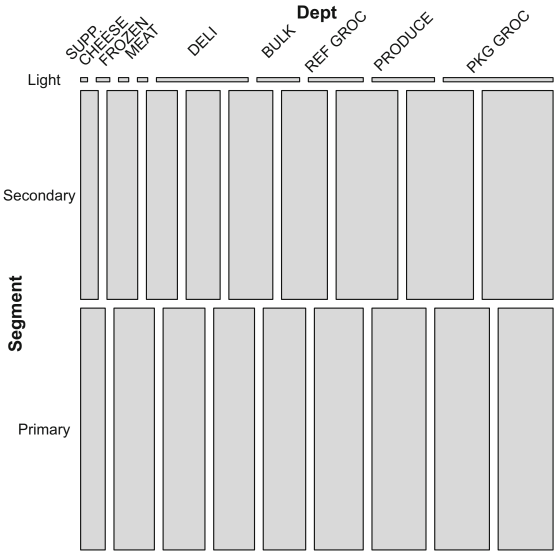


Fig. 5.6 A mosaic plot showing the relationship between customer segments and departments shopped

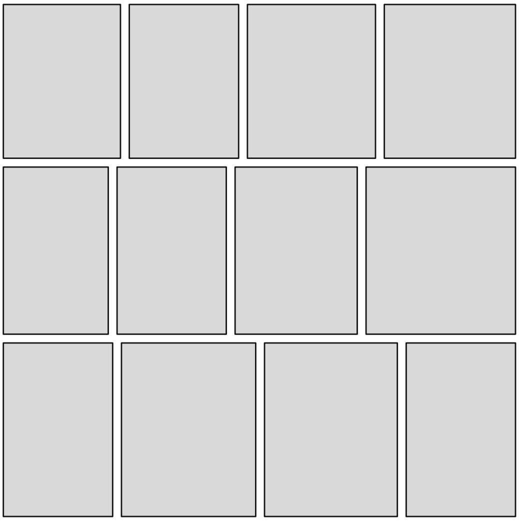


Fig. 5.7 A mosaic plot showing no relationship between two categorical variables. Reference plots like this are good to keep in mind when looking at mosaic plots

Most data that we need to visualize is not categorical, however. In the case of a quantitative variable and a categorical variable, we have already seen several good methods of showcasing relationships. Figure 5.5 shows several numeric results (the minimum, mean, and maximum spend by month) split by a categorical variable (the grocery store department). In Fig. 5.8 we see a great deal more information in a similar format. The previous chart showed spend at the department level. This chart shows spend at the individual shopper level for a sample of 10,000 shopper-months.

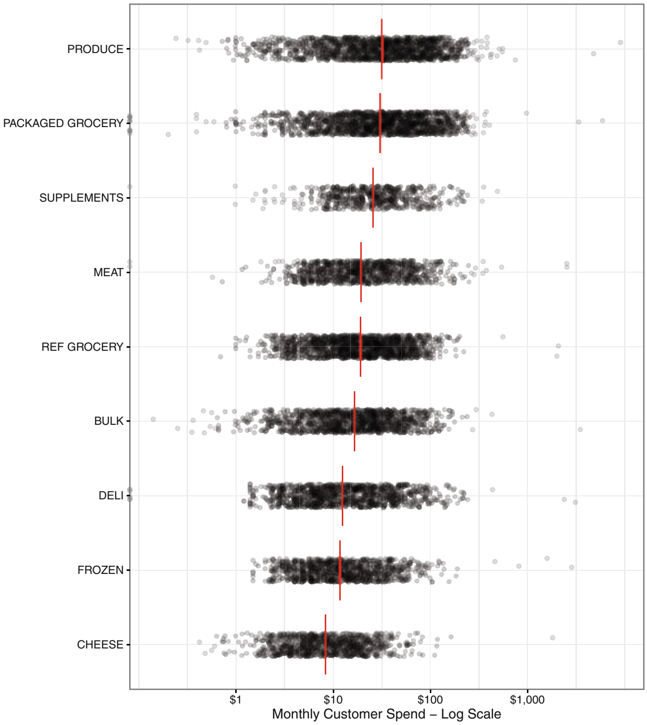


Fig. 5.8 A second example of a dotchart showing spending by department at the individual shopper level for 10,000 shoppers

Figure 5.8 displays spend on the $\log_{10}$ scale by individuals in a month across nine departments in the grocery store. The departments are ordered by the median monthly spend, shown as a vertical red line. The horizontal axis corresponds to a log scale to allow additional detail to be seen at the lower end of the scale. It also appears that the $\log_{10}$ scale reveals a much more symmetrical distribution—it is not uncommon for retail data to be approximately log-normal in distribution.

Several techniques we have not yet seen are illustrated in Fig. 5.8. The y-axis shows the levels of department. The values plotted at each level have been jittered so that more of them can be seen. Jittering adds a small random value, say between $-\varepsilon$ and ε, to the vertical coordinate of each plotted pair. Without the jittering all points would collapse onto their nearest horizontal grid line. We have also used transparency, set at the `ggplot2` value $\alpha = .1$. The interpretation of this value is that no fewer than $1/\alpha$ points plotted in the same location will appear completely opaque. Supplements are evidently shopped much less than, say, bulk, but the median spend is higher, presumably because each item is more expensive in this department.

With bivariate numerical data, the natural plotting technique is the scatterplot. Figure 5.9 illustrates this technique.

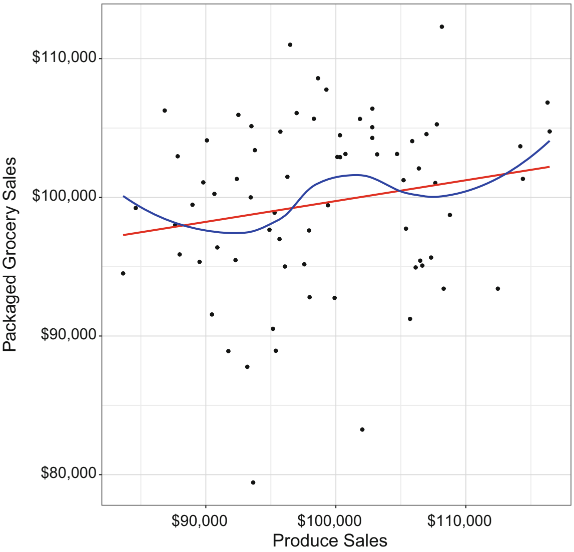


Fig. 5.9 A basic scatterplot, showing monthly sales for our two largest departments across 6 years. A linear regression line and loess smoother have been added to the plot to aid in interpretation of the relationship

Scatterplots are straightforward and adhering to the basic tenets laid out in Sect. 5.2 will keep a practitioner on the right track. The data speak for themselves, in this case illustrating the variability between produce and packaged grocery spend when viewed by month. Overall, the department sales move together, with a considerable amount of variation in the relationship.

When illustrating a relationship, one should strongly consider adding a fitted line to the graph to aid the viewer in decoding the relationship. In Fig. 5.9 we add two: a line built by linear regression and a curve created by locally weighted regression. The former needs little elaboration here, as linear regression is covered in Chap. 6. Locally-weighted regression, a technique introduced by Cleveland in 1979 and refined in 1988 [13], is a powerful technique for uncovering the relationship between two numerical variables. There are two competing terms, lowess and loess, and it seems the latter has become preeminent. They are closely related, both setting up a weighted neighborhood around an x value and fitting a regression line primarily influenced by points in vicinity of x. Much like our kernel smoothers discussed above, a bandwidth parameter, α, is specified. In each neighborhood a polynomial of degree d will be fit,[5] and we require a choice of $\alpha \in [\frac{d+1}{n}, 1]$. The default settings in R are $\alpha = .75$ and $d = 2$. For each subset of size $n\alpha$, a polynomial is fit. This fit is based on weighting the points and the typical weight function is the tricube weight. Let us assume that we have an observation x_i, a neighborhood around x_i denoted by $N(x_i)$, the width of which is r_i. Then the weight function is

[5] In practice d is almost always 1 or 2.

$$w_i(x) = \begin{cases} \left(1 - \left|\frac{x-x_i}{r_i}\right|^3\right)^3, & x \in N(x_i), \\ 0, & x \notin N(x_i). \end{cases}$$

The tricube kernel is nearly as effective as the Epanechnikov kernel. In practice, the choice of kernels is not nearly as important as the bandwidth choice.

We now turn our attention to multivariate data. Multivariate data is defined as data comprising at least three covariates per observation and, as mentioned earlier, most solutions require a thoughtful process. We can apply the general principles mentioned earlier and enjoy the profusion of options that are available. If there is one principal that stands above all others it is to avoid doing too much. This temptation is nearly irresistible. One may violate the rule profitably when a graphic will do the heavy lifting for several pages of text or when the graphic can be displayed on a slide for several minutes of explanation. This is not the norm and data scientists would be wise to split their story into multiple graphics if the audience is pressed for time.

Figure 5.10 is a reprisal of Fig. 5.8, made multivariate with the addition of the customer segmentation of the shopping data seen in Chap. 10. A number of interesting features emerge, notably the separation between secondary and primary shoppers for all departments except for supplements. The addition of median lines for each segment aids comprehension, particularly for the light shoppers, who make up small fraction of the shoppers. Interestingly, we can now see the compression of the deli spend—this department is one of the most popular with light shoppers.

Figure 5.10 is, essentially, a two-dimensional data visualization with a third dimension (segment), layered on top. Using two-dimensional plots with additional information to encode other variables is a common technique. Another variation, illustrated by Fig. 5.11 splits a two-dimensional plot into small multiples of pairs according to a third variable. The term “small multiples”, coined by Tufte, captures the idea that readers, once they have been oriented to an individual plot, can quickly discern similarities and differences across a family of plots.

Each small panel, known as a “facet”, is a scatterplot of spend versus items within a given department. A loess smoother is added to each panel. The advantages of faceting versus simply repeating scatterplots are several-fold: parsimonious code, an efficient use of space on the layout, the ability to order the facets in a sensible way, and common axes that facilitate comparison. With this treatment we can quickly identify interesting patterns within the data:

- The large volume of sales in produce, packaged grocery, and refrigerated grocery stand out relative to the other departments.
- Steeper curves indicate departments with cheaper items (produce, deli) while flatter curves show the expensive items (supplements, meat).
- Certain departments do not have large spends (cheese, frozen, and bulk).

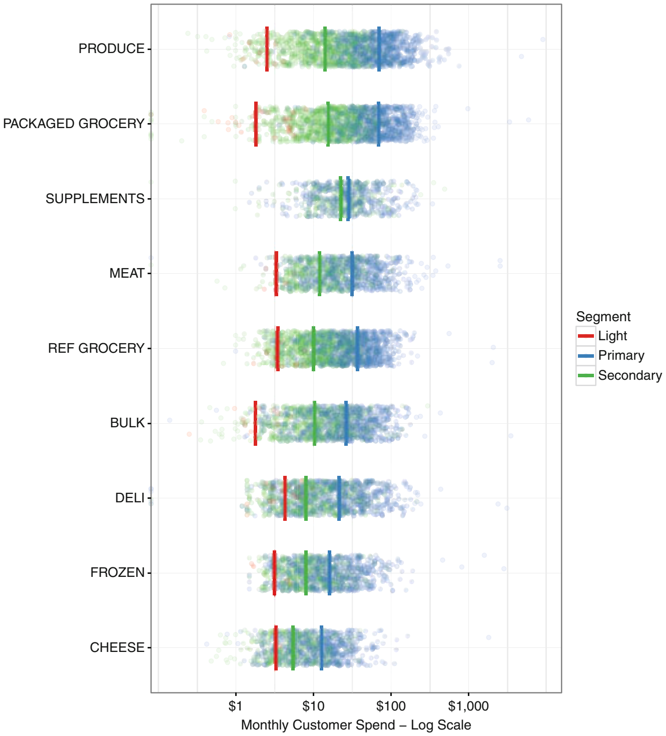


Fig. 5.10 An example of multivariate data. The spend-department dotchart is now colored based on the segments of the shoppers (either primary, secondary, or light). The medians are shown for each segment as a color-coded line

This section has served as a travelogue without a map. We have seen the destinations, but now we must learn how to get there. Building these plots requires a surprisingly small amount of code using the package that we will learn, `ggplot2`, partially because a great deal of complexity is hidden from us. Graphical software is a leaky abstraction, unfortunately, and so we will illustrate how to build up these charts from base elements.

5.4 Harnessing the Machine

The plots in this chapter were made with the `R` package `ggplot2`. This package is our strong recommendation for your personal data visualization tasks. There are three widely-used graphics packages associated with `R`: `base`, `lattice` [53], and `ggplot`. The `base` package is useful for quick plotting and for learning the basic techniques. It is possible to control many aspects of the plot with the `base` package but constructing publication-quality graphics is not easy. The second package, `lattice`, is based very closely on the ideas of Cleveland, but is not developed on the framework of a formal model. The `lattice` framework has limited its extensibility. The package `ggplot` is more flexible and easier to use.

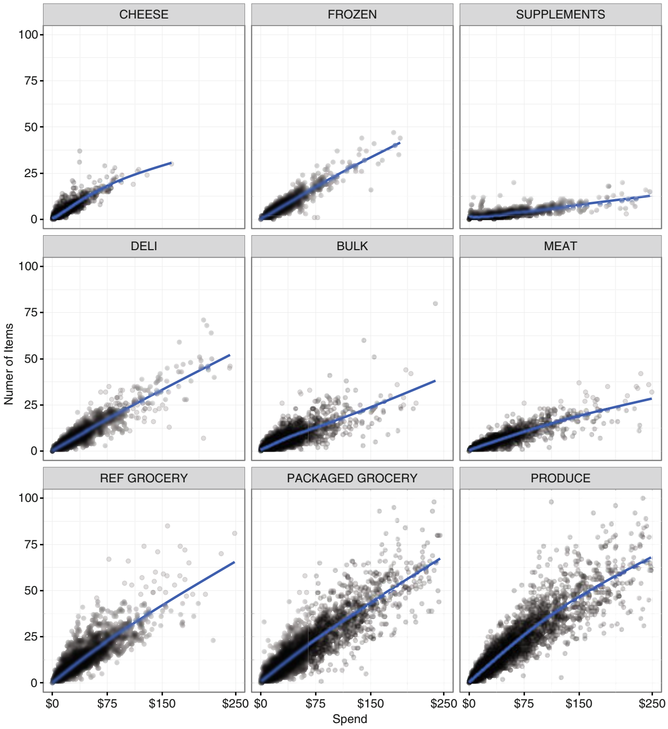


Fig. 5.11 A scatterplot, faceted by department, of spend versus items, with a loess smoother added to each panel

The package name, `ggplot2`, reflects both the version number, two, and the heritage, Wilkinson's Grammar of Graphics [69]. To quote Hadley Wickham, the principal author of `ggplot2`, "Wilkinson created the grammar of graphics to describe the deep features that underlie all statistical graphics." [66] Wickhams' books covers the grammar in some detail. Our treatment draws extensively from Wickham's texts.

The `ggplot` grammar requires work on the part of the reader. Once it is mastered, however, `ggplot2` provides the user with lots of flexibility and power. We encourage the reader to learn the grammar.

The grammar of graphics consists of a number of components that merge to form the grammar. They are

1. **data** The data to be rendered as a visual. Using `ggplot`, the data must be an `R` data frame. A `R` data frame is rectangular arrangement of the data with somewhat more specificity and overhead than a simple matrix. For example, a data frame may contain variables of several types, say, quantitative (numeric in the `R` lexicon) and categorical (a factor in the `R` lexicon).
2. **aes** *aes* is shorthand for *aesthetic mapping*. Aesthetic mappings tell `ggplot2` how to translate the data into graphic elements. The aesthetic mapping identifies the variables to be plotted on the x- and y-axes.

More may be added to the aesthetic mapping. For example, a categorical variable that determines the color of the points or lines is identified in the aesthetic mapping.

3. **geoms** *geoms* is shorthand for *geometric objects*. These are the elements that visually portray the data. Examples are points, lines, line segments, error bars, and polygons. All geometric object specifications take a form such as `geom_points()`.
4. **stats** These are statistical transformations that are used to reduce and summarize the data. The smoothers we saw above are examples of statistical transformations since the data was reduced in some manner to produce the smooth lines.
5. **scales** Scales provide the mapping between the raw data and the figure. Scales are also used to create legends and specialized axes.
6. **coord** The coordinate system takes us from the data to the plane of the graphic. This component is used, notably, to change axis scales from the original units of a variable to different units, say, logarithms.
7. **facet** As we saw in Fig. 5.11, facetting divides the graphic into subgraphics according to a categorical variable. This graphical component defines how the facets are arranged.

The process of building a visualization as a series of *layers* in `ggplot` is a remarkable improvement on the traditional process. Figures in `ggplot` are built by first creating a base consisting of a rudimentary plot and then adding more information and data to the base in the form of layers. Each layer may contain specific information from the attribute list above. Usually, layers inherit most of the attribute values from the initially created plot. The specification of the layer may be very simple since we only need to change a few items in each layer. Consequently, layers make the task of building a complex graphic much easier. We tend to build graphics one layer at a time. We can see the effect of each layer on the graphic and more easily correct coding errors. A highly readable tutorial on the subject is available at https://rpubs.com/hadley/ggplot2-layers.

Our intent in this chapter is not to systematically review `ggplot2` techniques but to provide a general understanding of how visualizations are constructed in `ggplot2`. In the remainder of the chapter, we explain how the graphics discussed above were built. To go further, utilize internet resources and, in particular, Stack Overflow (`https://www.stackoverflow.com`). If you can't solve a problem or remember an instruction, then search the internet for information. Adding *ggplot* to the search string makes one much more likely to find results related to `R` and to the `ggplot2` plotting package in particular. Using the vocabulary of the package in the search string is important to efficient searching.

We now turn to the code that created this chapter's figures. We first start by reading in the data and loading the necessary libraries.

```
library(ggplot2)
library(scales)
library(RColorBrewer)
```

The first two libraries, ggplot2 and scales are directly related to plotting. The final one, RColorBrewer, is based on the pioneering work on color and perception of Brewer et al. [7], is indispensable and we recommend it highly.

5.4.1 Building Fig. 5.2

Figure 5.2 is built from a summary table named month.summary. This data frame holds the sum of monthly sales for four departments at the co-op. The first five rows are displayed in Table 5.2.

Table 5.2 The first few rows of the data frame month.summary

Row	Month	Department	Sales
1	6	Packaged grocery	90,516.72
2	6	Produce	103,650.8
3	6	Bulk	37,105.20
4	6	Meat	35,221.97
5	5	Packaged grocery	102,178.30

The code follows.

```
month.summary <- read.delim("../data/month_summary.txt")

ggplot(data=month.summary,
       aes(x=factor(month), y=sales/1000, group=Dept, col=Dept) )
  + scale_color_brewer(palette="Set1")
  + geom_point()
  + geom_line()
  + theme_bw()
  + scale_y_continuous(label=dollar)
  + ylab("Avg Annual Sales in Month (000s)")
  + xlab("Month")
  + theme(legend.key.size=unit(0.5, "cm") )
```

After reading in the data, we build the plot in ggplot2, layer by layer.

1. **data** We use the data shown in Table 5.2.
2. **aes** We assign month to the x-axis and sales-divided-by-one-thousand to the y-axis. Department is used as a grouping variable. The consequence of

setting `group=Dept` is that points belonging to the same department will be joined by lines. Departments will identified by color. When building graphics that use grouping and color, it is common to forget to include the grouping variable in the aesthetic. As an exercise, we encourage the reader to build the plot *without* declaring the grouping variable in `aes`. The appearance of the resulting plot, with the telltale diagonal lines, are the mark of a missing grouping variable.

3. **geoms** There are two geometric objects used in the figure: points and lines. They are added as separate layers and inherit the grouping and color from the aesthetic.
4. **scales** We use a continuous scale and, for the axis labeling, take advantage of the library `scales` to reduce the effort of the reader to understand what has been plotted. The units are dollars so we pass the keyword `dollars` into the `label` argument. This parameter will ensure that the axis labels are formatted with dollar signs, commas, and cents. Other options include `percent` and `comma`, all of which promote readability. We also use the colors provided by `RColorBrewer` for the same reason.
5. **theme** Although not part of our original list, themes help you adjust the appearance of your plots. We invoke the simple and clean `theme_bw`, label the axes, and shrink boxes in the legend a bit. These features are almost always included in our graphics.

5.4.2 Building Fig. 5.3

Our next code segment builds our histogram and empirical density function.

```
# Read in grocery store summary data.
working.dir <- "../data/"
gd <- read.delim(paste0(working.dir, "grocery_data.txt"))
ggplot(gd, aes(x = ownerSales/1000))
    + geom_histogram(aes(y=..density..))
    + geom_line(stat="density",col="gray50",size=1.5,
              bw="SJ") + # Changing the bandwidth algorithm to 'SJ'
    + theme_bw()
    + scale_x_continuous(label=dollar)
    + ylab("Density")
    +  xlab("Monthly Sales (000)")
```

The key features used to construct the figure are as follows.

1. **data** Histograms summarize distributions as they are built. We don't have to carry out data reduction before building the figure. It's done by `ggplot` in the construction of the histogram.

2. **aes** Histograms are defined by a single variable which we assign to the x-axis. `ggplot2` determines the y-values using our guidance.
3. **geoms** The first geometric object is a histogram. The `aes` argument sets the scale for the y-axis. Our choice of units for the histogram are proportions. The y-axis will show the proportion of observations in each interval and it's specified by setting `y=..density..`. The syntax of `..density..` looks odd. The pattern of periods (two at the beginning and two at the end) indicates a statistical computation is necessary. The other commonly chosen option shows the counts of observations in each interval. Entering `?stat_bin` from the console will provide more information.

   A second `geom`, `line` is also specified. The `stat` argument specifies that the line to be drawn is the graph of a empirical density function (Sect. 5.3.1). Several other attributes are specified: the color, the line width (by adjusting `size`), and that the method of computing the bandwidth is to be Sheather and Jones' method [54].

5.4.3 Building Fig. 5.4

The code for the violin plot from Fig. 5.4 is next.

```
ggplot(gd, aes(x=1, y=ownerSales/1000))
    + geom_violin(draw_quantiles = c(0.25,0.5,0.75))
    + theme_bw()
    + scale_y_continuous(label=dollar)
    + xlab("")
    + ylab("Monthly Sales (000)")
    + theme(axis.text.x=element_blank(),axis.ticks=element_blank())
    + labs(title="Violin Plot of Monthly Sales")
```

A couple of new techniques emerge.

1. **aes** The x-axis is set to be a constant. The result is that the violin plot will be drawn in the center of the graphic.
2. **geoms** The `geom_violinplot` accepts an argument specifying that quantiles are to be drawn. We specify that these are to be the first, second, and third *quartiles*.
3. **theme** Since the x-axis does not have any meaning, we would like to avoid showing ticks marks and labels. The arguments to `theme` eliminate those features. A internet search for something similar to "ggplot blank x axis" will provide details.

5.4.4 Building Fig. 5.5

The dotchart in Fig. 5.5 requires a prepared data frame `dept.summary` containing the sample minimum, mean, and maximum sales the 17 departments of the grocery store co-op. The first five rows are shown in Table 5.3.

Table 5.3 The first five rows of the `dept.summary` data.frame

		Sample		
Row	Department	Minimum	Mean	Maximum
1	Packaged grocery	79,434.24	99,348.55	112,303.60
2	Produce	76,799.20	98,711.50	116,442.92
3	Bulk	31,610.05	39,482.60	45,908.04
4	Refrigerated grocery	44,239.77	49,940.93	55,656.07
5	Cheese	12,407.99	14,674.32	16,404.14

After reading the summary data into a data frame, it is re-ordered by the sample mean.

```
dept.summary <- read.delim("../data/dept_summary.txt")
dept.summary$dept <- reorder(dept.summary$dept, dept.summary$mean.val)

ggplot(dept.summary, aes(x=mean.val, y=dept))
    + theme_bw()
    + geom_errorbarh(aes(x=mean.val, xmax=max.val, xmin=min.val, y=dept),
                       height=0, color="gray60")
    + geom_point(col="black")
    + ylab("")
    + xlab("Monthly Sales (000s) - Log Scale")
    + scale_x_continuous(label=dollar,trans="log10",
                     breaks=c(1000,2500,10000,25000,100000))
```

1. **aes** When a factor is used as an x- or y-variable in an aesthetic, the factor levels are mapped to sequential integers, say $1, 2, \ldots, g$, where g is the number of levels. Knowing that the levels occur at integer positions on the x-axis will be important when we add features like jittering.
2. **geoms** Another new `geom`, `geom_errorbarh`, is used. The "h" stands for horizontal—the vertical variety needs no suffix. The aesthetic mapping for `geom_errorbarh` requires us to supply x- or y-variables and the starting and ending positions for the error bars. The parameter `height` specifies the width of the whiskers on the bars. We've suppressed the whiskers as they add no information. Note that points are desired at the median and receive their own `geom`.
3. **scale_x_continuous** We repeat our label trick and instruct `ggplot` to transform the variable plotted on x-axis according to the transformation $x \rightarrow \log_{10}(x)$. Also, we set the label positions using the `breaks` argument.

5.4.5 Building Fig. 5.8

We build the plots for Figs. 5.8 and 5.10 in the next series of code segments. We begin using `sample.int` to draw a random sample of manageable size from the shopper data set. Setting the seed of the random number generator with the instruction `set.seed` ensures that our results are the same on repeated runs. We also order our departments based on the median total sales by department.

```
set.seed(3939394)
# Read in grocery store detail data.
gdd <- read.delim(paste0(working.dir,"shopper_dept_segment.txt"))
this.data <- gdd[sample.int(nrow(gdd),size=10000,replace=F),]
this.data$DepartmentName <- reorder(this.data$DepartmentName,
                                    this.data$TotalSales,
                                    FUN=median)
```

Drawing the vertical lines that depict the medians requires those values to be calculated and it is clearest if we assign them to a data frame of their own. We call that data frame `medians`.

```
medians <- aggregate(this.data$TotalSales,
                     list(this.data$DepartmentName),
                     FUN=median)
names(medians) <- c("DepartmentName","TotalSales")
medians$y.val <- as.numeric(medians$DepartmentName)
```

We are now ready to build Fig. 5.8.

```
ggplot(this.data, aes(x = TotalSales, y = DepartmentName))
  + theme_bw()
  + geom_point(position=position_jitter(h = .4),
             alpha=0.1)
  + ylab("")
  + xlab("Monthly Customer Spend - Log Scale")
  + scale_x_continuous(label=dollar, trans="log10",
                     breaks=c(1,10,100,1000))
  + geom_segment(data = medians,
             aes(x = TotalSales, xend = TotalSales,
                 y = y.val -.4, yend = y.val+ .4),
             col="red")
```

1. **data** Two data sets are used to construct the figure. The principal data set is the sample of individual shopper grocery data. A second data set called `medians` is used for the vertical lines and is passed directly into the necessary `geom_segment`.

2. **aes** This plot uses an aesthetic that is similar to what we have seen before, with the categorical variable `departmentName` assigned to the y-axis. Individual shopper sales are assigned to the x-axis. There is a separate set of aesthetic mappings used with `geom_segment`.
3. **geoms** There are two `geoms` used in this plot, one for the points and one for the line segments. The `geom_point` has been used before, but now we show how to add jittering to a plot. Recall that jittering perturbs points so that there is less over-plotting and individual points are easier to see. The command, `position=position_jitter(h=.4)` instructs `ggplot2` to place the points at the original x-axis coordinate and to perturb the y-axis coordinate. The argument $h = .4$ controls the magnitude of the random perturbations.

   The other `geom` layer, `geom_segment` uses the second data set. We specify the starting and ending line segment coordinates on the x- and y-axes.
4. **scale** Once again we use the *dollar* labeling from the library `scales`. As we have been doing in other plots, we instruct `ggplot` to transform the variable plotted on x-axis according to the mapping $x \rightarrow \log_{10}(x)$ and specify where we would like the labels to appear.

5.4.6 Building Fig. 5.10

The code used to construct Fig. 5.10 is a relatively simple modification of the code used to construct Fig. 5.8. The key difference is that we wish to color the points according to segment, as well as have separate median lines for the segments. First, we must calculate the medians for each department and segment.

```
medians <- aggregate(this.data$TotalSales,
                     list(Segment=this.data$Segment,
                          DepartmentName=this.data$DepartmentName),
                     FUN=median)
names(medians)[3] <- "TotalSales"
jit.val <- 0.6 # More jittering with segments
```

The function `aggregate` is convenient for summarizing one variable based on one or more other variables. We rename the column that holds the calculated medians to match the aesthetic we'll use in our plot.

As we shall see, it is quick to add a grouping variable that allows coloring of the points and the technique is analogous to what we did in Fig. 5.2. There we used grouping and color aesthetics to ensure that the department sales by month were joined by a line and common color. Here the variable `Segment` will fill that role.

```
ggplot(this.data,
      aes(x=TotalSales, y=DepartmentName, group=Segment, col=Segment))
  + theme_bw()
  + geom_point(position=position_jitter(h=jit.val),
             alpha=0.1)
  + ylab("")
  + scale_color_brewer(palette = "Set1")
  + xlab("Monthly Customer Spend - Log Scale")
  + scale_x_continuous(label=dollar,trans="log10", breaks=c(1,10,100,1000))
  + geom_segment(data = medians,
      aes(x = TotalSales,xend=TotalSales, group=Segment, col=Segment,
          y = y.val-0.5*jit.val, yend=y.val+0.5*jit.val), size=1.25)
```

What are the differences between this code and the previous code? Just the addition of the argument `group=Segment` and `col=Segment` in the `ggplot` aesthetic and `geom_segment` aesthetics, respectively. Other aspects of our plots remain the same.

5.4.7 Building Fig. 5.11

Our final example builds Fig. 5.11. The code is reproduced here, though there are few features we have not seen before.

```
set.seed(3939394)
this.data <- gdd[sample.int(nrow(gdd),size=25000,replace=F),]
this.data$DepartmentName <- reorder(this.data$DepartmentName,
                    this.data$TotalSales,
                    FUN=sum)
ggplot(this.data, aes( x = TotalSales, y = Items))
    + geom_point(alpha=0.1)
    + stat_smooth(se=F)
    + facet_wrap(~DepartmentName)
    + scale_x_continuous(label=dollar, limits=c(0,250),
         breaks=c(0,75,150,250))
    + scale_y_continuous(limits=c(0,100),label=comma)
    + theme_bw()
    + xlab("Spend")
    + ylab("Number of Items")
```

The first new feature is a statistic that stands on its own (as opposed to being hidden inside `geom_line` as we saw above). The function, `stat_smooth`, is quite useful, adding a line to a plot according to the specified method. In the code that generated Fig. 5.9, we added two smooths in two layers. One layer added the default smoothing method, loess, and the second added a fitted linear regression line using the instruction `stat_smooth(method="lm")`. The default is to include standard error envelopes. We have suppressed the envelopes by passing the argument `se=F`.

The other new feature is `facet_wrap`, the function that controls the creation of the panels. The *wrap* version creates a rectangular array of panels. One panel is created for each level of the factor `DepartmentName`. Only observations associated with a particular level of the factor are graphed in the panel. Rows and columns can be controlled. Multiple variables can be set on the right-hand side of the tilde (∼) to partition the data by combinations of levels of the variables. The analog of `facet_wrap` is `facet_grid`, which is designed to set up the panels in a tabular array, specifying row factors and column factors using the same tilde operator. One important parameter, not changed in this code, is the `scales` value, which can be set to `free`, `free_x`, `free_y`, and `fixed`. The default is the last one and it forces a single set of axes for every panel. The `free` varieties allow the axes to differ and generates the best visual fit for each panel. The `free` option is often a mistake if the purpose of the panels is to compare and contrast groups because the reader will have to account for different scales in different panels. The best visualization depends on the goals of the data scientist.

A final word of caution and encouragement. Plotting with `ggplot2` imposes an initially steep learning curve. The payoff is well worth the effort—do not underestimate the importance of communicating clearly and efficiently. Visualization is the best way to do it. The key to learning `ggplot` is to start simple and add complexity one layer at a time. Build a simple scatterplot with `geom_point`, then add a layer that identifies groups by color. Add a layer showing smooths. Label the axes clearly and adjust the font sizes. Every successful plot that we created for the first 6 months of working with `ggplot2` was made by incremental steps.

5.5 Exercises

5.1. Use the data set `grocery_data.txt` containing sales by month and build a dotchart of monthly sales by department. Put department on the y-axis and sales (in thousands) on the x-axis. Include the following features:

a. Sort the departments based on average sales.
b. Jitter the points vertically and set `alpha` to help with over-plotting.
c. Label the x-axis using dollars.
d. Suppress the y-axis label and use a sensible label for the x-axis.

What patterns emerge from the departments? How does the spread of sales vary by department? Why might this be?

5.2. Add vertical red lines representing the median monthly sales by department to the figure of Problem 5.1.

5.3. Build the middle panel from Fig. 5.3 with the bandwidth parameter set to the values 1, 5, and 25. What is the bandwidth returned by the "SJ" method? How does it compare to the `bw.nrd0` method[6]?

5.4. Continuing to use the data set `grocery_data.txt`, build a faceted graphic showing the empirical density of sales per month by department. Allow the x- and y-axes to vary within the panels. Use the SheatherJones bandwidth selection algorithm. Compare the densities.

5.5. Now we fine-tune the solution to Problem 5.4 to make it publication worthy. Add the following features:

a. Angle the x-axis labels to 45°. (Hint: `?theme` and search for `axis.text.x`.)
b. Use the black and white theme.
c. Remove the beer & wine segment from the plot since there are very few observations in the segment.
d. Shrink the font size of the facet titles (called `strip.text`) to make the longer names fit.

5.6. Repeat Problem 5.1 with violin plots. Add quartile lines and draw a red vertical line at the overall average monthly sales across all months and all departments. Use the parameter `scale="width"` in the violin plot to avoid the default behavior of the width being proportional to sample size.

Does this view of the data add to any insights beyond what you learned from Problem 5.1?

5.7. Build the mosaic plot shown in Fig. 5.6. This plot is *not* made by ggplot, but is made with the `mosaic` function in the package `vcd`. Controlling the orientation of the axis labels is tricky; we recommend replacing the department and segment names with abbreviations.

5.8. Use the `gdd` data to build the sum of sales by year and month, and segment. Display these in a line chart in chronological order, with each segment being its own line.

5.9. Use the `gdd` data to build the sum of sales by year and month and segment. Plot sum of sales against year and month, with each segment shown by a separate line. Include the following features in your plot:

a. Vertically align the year-month labels.
b. Plot the y-axis on a log scale and choose sensible breakpoints. Label the axis with the "dollar" formatting.
c. Label the x-axis every 12 months starting at 2010-1.
d. Add a linear trend line, by segment, to the plot. What information can you infer about the light segment from this graph?

[6] The purpose of these last two questions is to learn how to extract the bandwidth information from `R` directly.

Chapter 6
Linear Regression Methods

Abstract Linear regression is a broad and well-developed area of statistics. If there is a core to statistical methodology, then linear regression is it. The ubiquity of linear regression methods in statistics and data analytics stems from the ease with which one may fit tractable models that describe the primary features of a process or population. Not only is linear regression useful for description, it's also very useful for prediction since the models often provide good approximations of complex relationships. In the field of statistics, hypothesis testing and confidence intervals are routinely used in linear regression analyses. The extension of these methods to data science is often unsuccessful because of the prevalence of opportunistically collected data. Most of the time, opportunistically collected data cannot support inferential methods because the quality of the inferences produced by the methods is unknown. We discuss inference herein so that the reader may understand the potential for success and for failure of these methods. However, the focus is on the essential and most useful aspects of the subject matter for data analytics—the fitted models. The topic of linear regression provides an avenue to gain experience with the statistical package R, one of the most popular software packages used by data scientists.

6.1 Introduction

The subject matter of this chapter is applicable a wide collection of data analytic problems. These problems share a common feature: one of the variables stands out in importance and the aim is either to gain a greater understanding of the process that generated the variable or the aim is to predict unobserved, often future, values of the variable. For example, if there is a resource of limited availability and variable demand, it's important to understand how and why demand varies so that the demand may be met. If the analyst is able to arrive at a reasonably good understanding of the process

B. Steele et al., *Algorithms for Data Science*,
DOI 10.1007/978-3-319-45797-0_6

through linear regression analysis, then a logical next step is to develop a predictive model for forecasting demand in the future. Predictive applications of linear regression generally are narrower and better defined than the problem of understanding the relationship between one variable and a set of explanatory variables. As a substantial portion of data science involves prediction and predictive analytics, linear regression is a very important component of data analytics. It should not be forgotten that linear regression may be successfully used for both learning about the origins of the data and for predicting future and unknown values of a process.

The linear regression model is set up before going on to an example.

6.2 The Linear Regression Model

In the linear regression setting, the data may be viewed as a set of n pairs. We'll use the notation $D = \{(y_1, \mathbf{x}_1), \ldots, (y_n, \mathbf{x}_n)\}$ to denote the data. The statistical view of the situation supposes that $y_i, i = 1, 2, \ldots, n$ is realization of a random variable, henceforth identified as the *response* variable Y_i, and that the concomitant *explanatory* vector $\mathbf{x}_i$ paired with y_i is not random and can be used to explain y_i in a sense that will be made concrete momentarily. Occasionally, the statistical objective is to predict an unobserved realized value y_0 from the observed vector $\mathbf{x}_0$ using a predictive function constructed from the data. In statistics, we *predict* realizations of a random variable and *estimate* parameters that describe a distribution of the random variable.

The variables y_i and $\mathbf{x}_i$ are inherently different—the aim is to describe, or model, the expected value of the random variable Y_i whereas $\mathbf{x}_i$ is a vector of explanatory variables of secondary interest. Routinely, in statistical applications, it's assumed that the explanatory vectors do not have a distribution but rather are fixed and measured without error, or in complete control of the researcher. Though this assumption is often specious, linear regression is a valuable method of extracting information regarding the process or population from which the data originated. In predictive problems, the linear regression model often provides a good and relatively simple approximation of complex relationships between the expected value of Y_0 and the *predictor* vector $\mathbf{x}_0$. The vector $\mathbf{x}$ may be referred to as either an explanatory or a predictor vector depending on the objective of analysis. For simplicity though, we'll refer to the vectors $\mathbf{x}_i$ and $\mathbf{x}_0$ as predictor vectors with the understanding that the analyst may only be interested in explaining the relationship between the expected value of the response variable and the concomitant predictor vector.

The linear regression model specifies that the expected value of Y_i is a linear function of $\mathbf{x}_i$ given by

$$\mathrm{E}(Y_i|\mathbf{x}_i) = \beta_0 + \beta_1 x_{i,1} + \cdots + \beta_p x_{i,p}. \tag{6.1}$$

The model may be expressed more compactly using vectors:

$$\mathrm{E}(Y_i|\mathbf{x}_i) = \mathbf{x}_i^T\boldsymbol{\beta},$$

where $\boldsymbol{\beta} = [\beta_0 \;\; \beta_1 \;\; \cdots \;\; \beta_p]^T$ is a vector of unknown constants. The vector $\mathbf{x}_i = [1 \;\; x_{i,1} \cdots \;\; x_{i,p}]^T$ contains the observations on the p predictor variables. For convenience, we set $q = p + 1$ so that $\boldsymbol{\beta}$ and $\mathbf{x}_i$ are of length q.

To illustrate a typical problem that is amenable to regression analysis, suppose that Y_i is the price of a particular stock at time step i. A linear regression model may describe the expected value of Y_i as a function of other stock prices measured at some previous time step. It's clear that the realized prices will not coincide exactly with a linear model (there are far too many factors that affect price), but it is not unreasonable to suppose that the average of many observations made when the predictor vector equals $\mathbf{x}_i$ can be approximated by a linear combination of the terms in $\mathbf{x}_i$. Now with the hope of predicting the stock price in the future, data can be collected automatically for some time span,[1] and using the collected data, an estimate of $\boldsymbol{\beta}$ computed. The next step is evaluating the predictive model. If the observed stock prices are, on average, close to the model predictions of the stock prices, then the model is promising for prediction.

The prediction application is simple in the sense that the details of the predictive model are not particularly important provided that it produces accurate predictions of Y. Often, more is desired, and we are interested in understanding the influence of each predictor variable on Y, or more precisely on the expected value of Y. The model of the expected value of Y_i as a linear function of $\mathbf{x}_i$ might be expanded to specify a form for the distribution of Y_i. An extended set of inferences are possible when the distribution of Y_i meets certain conditions such as normality.

Extended inferences depend on a set of conditions that are described herein as the *inferential* model. The inferential model states that

$$\begin{aligned} Y &= \mathrm{E}(Y|\mathbf{x}) + \varepsilon, \\ \mathrm{E}(Y|\mathbf{x}) &= \mathbf{x}^T\boldsymbol{\beta}, \\ \text{and } \varepsilon &\sim N(0, \sigma_\varepsilon^2). \end{aligned} \tag{6.2}$$

The third statement specifies that the residual random variable ε is distributed as a normal random variable with expectation 0 and variance σ_ε^2. The variance of ε is unrelated to $\mathbf{x}$ and is the same for every value of $\mathrm{E}(Y|\mathbf{x})$. Constant variance implies that the differences between the realized values of Y and model estimates of $\mathrm{E}(Y|\mathbf{x})$ should be roughly equal in magnitude for any choice of $\mathbf{x}$. A final condition is necessary for accurate hypothesis testing and unbiased estimation of σ_ε^2. The condition states that the data values $y_1, \ldots, y_n$ were generated by independent random variables $Y_1, \ldots, Y_n$, all of which follow model (6.2).

[1] A `Python` script for this purpose will be written in Chap. 11, Sect. 11.10.

Occasionally, the intercept β_0 is omitted from the model by not augmenting $\mathbf{x}$ with the constant 1, in which case $q = p$. An intercept would be omitted if $\mathbf{x} = \mathbf{0}$ implies that Y must take on the value zero, for example, if Y is the rate of a reaction and $\mathbf{x}$ consists of measurements on the concentration of reactants.

6.2.1 Example: Depression, Fatalism, and Simplicity

The type of situation for which the inferential linear model (6.2) is useful is illustrated by the Ginzberg data set [19]. To treat depression, a sometimes incapacitating mental illness, the causes should be understood. It's been postulated that depression is associated with a couple of less complex conditions, fatalism and simplicity [51]. Fatalism refers to a perceived inability to control the world in which a person lives. The extent to which a person views events and circumstances as either positive or negative, without gradation and complexity, is described by the term *simplicity*. The degree to which these dispositions are associated with depression might illuminate the value of treating depression by addressing a person's fatalistic and simplistic perceptions. Quantitatively measuring the extent to which fatalism and simplicity are related to depression is an important step toward understanding the interrelationships between depression and these conditions.

The Ginzberg data set contains 82 observations made on patients hospitalized for depression. Patients were scored on depression severity, simplicity, and fatalism. The variables were scaled so that large values reflect stronger manifestations of simplicity and fatalism and more severe depression. The strength of linear association between fatalism and depression as measured by Pearson's correlation coefficient is $r = .657$, and the correlation between simplicity and depression is $r = .643$. The correlation coefficients indicate a moderately strong degree of positive linear association between depression and each variable individually and some assurance that there is a joint relationship between depression and fatalism and simplicity. The next task is to quantify the strength of joint association and describe the relationship, or more precisely, produce an objective and tractable approximation of what must be a complicated and shifting relationship at the level of individual patients.

Let us consider the following linear model

$$\mathrm{E}(Y|\mathbf{x}) = \beta_0 + \beta_1 x_{\text{fatalism}} + \beta_2 x_{\text{simplicity}} = \mathbf{x}^T\boldsymbol{\beta}, \tag{6.3}$$

where Y is depression score and $\mathbf{x} = [1 \quad x_{\text{fatalism}} \quad x_{\text{simplicity}}]^T$. Realistically, model 6.3 is at best a rough approximation of the actual relationship.

We proceed in the hopes of gleaning some information from the approximation. A computational algorithm was introduced in Chap. 3 for computing the estimate of the parameter vector $\boldsymbol{\beta}$ for a linear model such as the right-hand side of Eq. (6.3). A similar algorithm produced the parameter estimates shown in Table 6.1, but it remains to explain the meaning of the confidence intervals shown in the table. We can, however, interpret the parameter estimates mathematically. Since the fatalism and simplicity scores have been scaled to have the common mean value of 1 and standard deviation of .5, the parameter estimates can be directly compared. In particular, a one unit increase in fatalism when simplicity is held fixed is estimated to induce an increase of .418 units on the depression scale, and a one unit increase in simplicity when fatalism is held fixed is estimated to increase depression score by .380 units, and so we may argue that fatalism and simplicity are nearly equal in determining depression score, but that fatalism is more important than simplicity. The argument is made in an artificial context, because attitudes and beliefs, and specifically, fatalism and simplicity scores are impossible to manipulate by increasing one with the other held fixed. In fact, fatalism and simplicity are correlated ($r = .631$). Generally, individuals with larger fatalism scores will also manifest larger simplicity scores compared to individuals with lesser fatalism scores.

Table 6.1 Parameter estimates, standard errors, and approximate 95% confidence intervals for the parameters of model (6.3)

	Parameter	Standard	95% confidence interval	
Variable	estimate	error	Lower bound	Upper bound
Constant	.2027	.0947	.0133	.3921
Fatalism	.4178	.1006	.2166	.6190
Simplicity	.3795	.1006	.1783	.5807

A fitted linear model involving a single variable, say $\widehat{\mathrm{E}}(Y|x_1) = \widehat{\beta}_0 + \widehat{\beta}_1 x_1$ describes a line in the sense that every fitted value that may be calculated using a value of x_1 is a point on the same line.[2] With two variables, the fitted model describes a plane in the sense that every fitted value obtained from the model and a predictor vector $\mathbf{x} = [x_1 \;\; x_2]^T$ is a point on the same plane.[3] Figure 6.1 shows the fitted model of depression. Note that the intersection of the plane with the vertical sides of the box at opposing sides form lines with the same slope though the vertical position of the lines are different. This observation agrees with the interpretation that when fatalism is held constant, a one unit increase in simplicity produces an increase in depression score of .380 units regardless of the fatalism score. These statements about the change in the response variable (depression) as a function of the explanatory variables are examples of statistical inference. The statements are based on a sample of data but are being generalized to a larger population of individuals suffering from depression. The inference may well be wrong, perhaps grossly

[2] This is to be expected since the fitted model is the equation of a line.

[3] The fitted model is an equation describing a plane.

wrong to the extent that there is *no* relationship between depression and the two explanatory variables.

The possibility of incorrectly describing the population stems from the estimates having been computed from a sample and the model being an approximation. Sampling has introduced error into the parameter estimates. There would be no sampling error if every individual in the population were measured and the data set amounted to a census of the population.

The degree of error associated with one particular estimate is quantified by the standard error of the estimator. This statistic may be loosely interpreted as the average absolute difference between the true, unknown parameter and a parameter estimate computed from a sample of independent observations drawn from the population or process. Table 6.1 shows the standard errors $\widehat{\sigma}(\widehat{\beta}_1)$ and $\widehat{\sigma}(\widehat{\beta}_2)$ associated with each of the parameter estimates. The standard errors do not immediately provide useful information about the extent of error in the estimates. Confidence intervals provide an alternative interpretation and sometimes, a better approach to dealing with imprecision in the estimates related to sampling. Before expanding on confidence intervals, we review least squares estimation.

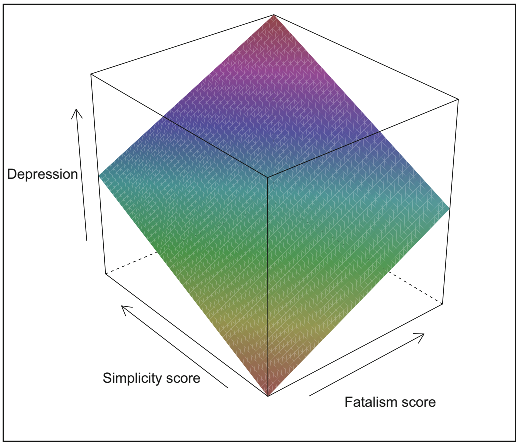


Fig. 6.1 The fitted model of depression showing the estimated expected value of depression score given fatalism and simplicity scores. The plane shows the triples $(x_{\text{fatalism}}, x_{\text{simplicity}}, \widehat{y})$ where $\widehat{y} = .203 + .418 \times x_{\text{fatalism}} + .378 \times x_{\text{simplicity}}$

6.2.2 Least Squares

The first computational task of linear regression analysis is to compute an estimate of the parameter vector $\boldsymbol{\beta}$. In statistics and data science, the least squares estimator is nearly always the first choice of estimators because it's easy to understand and compute. Furthermore, across a very wide breadth of problems, it's accuracy rivals that of more complex methods. Confidence intervals and hypothesis tests regarding the parameters β_0, β_1, ..., β_p are straightforward. The accuracy of the inferences drawn from model (6.2) depend on the appropriateness of the inferential model for the situation at hand. All of the computational properties originate from the least squares method of estimation.

The objective of least squares is to minimize the sum of the squared residuals. The residuals are the differences between the observed values y_i and the fitted values $\widehat{y}_i = \mathbf{x}_i^T \widehat{\boldsymbol{\beta}}, i = 1, \ldots, n$ where $\widehat{\boldsymbol{\beta}}$ is a q-length vector of real numbers. The sum of the squared residuals is

$$\begin{aligned} S(\widehat{\boldsymbol{\beta}}) &= \textstyle\sum_{i=1}^n (y_i - \widehat{y}_i)^2 \\ &= \textstyle\sum_{i=1}^n \left(y_i - \mathbf{x}_i^T \widehat{\boldsymbol{\beta}}\right)^2 . \end{aligned}$$

The sum of squared residuals is minimized by the choice of $\boldsymbol{\beta}$. Of course, $\boldsymbol{\beta}$ is unknown—we need to generate a value to do anything further. The vector $\widehat{\boldsymbol{\beta}}$ that minimizes $S(\cdot)$ is called the least squares estimator, and, by definition, every other vector will yield a sum of squared errors at least as large as the least squares estimator.[4] A computational form for the least squares estimator can be determined using multivariable calculus. Rather than proceeding through the derivation,[5] we limit the discussion to stating that the least squares estimator of $\boldsymbol{\beta}$ is the solution to the *normal equations*

$$\mathbf{X}^T \mathbf{X} \boldsymbol{\beta} = \mathbf{X}^T \mathbf{y}.$$

The details of building $\underset{n \times q}{\mathbf{X}}$ were developed in Sect. 3.9.2. Rencher and Schaalje [50] provide a lucid and in-depth exposition on the theory of linear models. In brief, $\mathbf{X}$ is formed by stacking the predictor vectors $\mathbf{x}_1^T, \ldots, \mathbf{x}_n^T$. The vector $\mathbf{y} = [y_1 \cdots y_n]^T$ consists of the n realizations of the random variables $Y_1, \ldots, Y_n$. If $\mathbf{X}^T \mathbf{X}$ is invertible, then the solution to the normal equations is

$$\widehat{\boldsymbol{\beta}} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}. \tag{6.4}$$

If $\mathbf{X}^T \mathbf{X}$ is not invertible, then $(\mathbf{X}^T \mathbf{X})^{-1}$ does not exist and solving the normal equations is more difficult. An algorithm for computing $\widehat{\boldsymbol{\beta}}$ was the subject of Sect. 3.10.

If the inferential model (Eq. (6.2)) correctly or approximately describes the relationship between $\mathrm{E}(Y)$ and $\mathbf{x}$ and distribution of ε, then the variances of the following estimators are simple in form. Let's assume that the inferential model is correct. Then, the variance of Y about its conditional expectation $\mathrm{E}(Y|\mathbf{x}) = \mathbf{x}^T \boldsymbol{\beta}$ is estimated by

$$\widehat{\sigma}_\varepsilon^2 = \frac{\sum_{i=1}^n (y_i - \widehat{y}_i)^2}{n - q},$$

[4] If $\mathbf{X}$ is not full rank, then the optimality statement needs to be modified.

[5] Exercise 3.3.7 guides the reader through the derivation.

where $\widehat{y}_i = \mathbf{x}_i^T\widehat{\boldsymbol{\beta}}$. The variance of the parameter estimator $\widehat{\boldsymbol{\beta}}$ is the $q \times q$ matrix

$$\mathrm{var}(\widehat{\boldsymbol{\beta}}) = \sigma_\varepsilon^2(\mathbf{X}^T\mathbf{X})^{-1}.$$

The diagonal elements of $\mathrm{var}(\widehat{\boldsymbol{\beta}})$ are the variances of the individual estimators $\widehat{\beta}_0, \ldots, \widehat{\beta}_p$ and the off-diagonal elements are the covariances between the individual estimators. The usual estimator of the variance matrix is $\widehat{\mathrm{var}}(\widehat{\boldsymbol{\beta}}) = \widehat{\sigma}_\varepsilon^2(\mathbf{X}^T\mathbf{X})^{-1}$. The estimated variance of an individual estimator $\widehat{\beta}_i$ is extracted from the diagonal of $\widehat{\mathrm{var}}(\widehat{\boldsymbol{\beta}})$. The square root of the estimated variance usually is referred to as the standard error of $\widehat{\beta}_i$ and is

$$\widehat{\sigma}(\widehat{\beta}_i) = \sqrt{\widehat{\sigma}^2(\widehat{\beta}_i)}.$$

Standard errors are used extensively for confidence interval building and hypothesis testing. As noted above, the appropriateness of these estimators is contingent in the realism of the inferential model.

The *point* estimates $\widehat{\beta}_i$, $i = 0, 1, \ldots, p$, may be interpreted as the single, best possible estimate of β_i. Despite being best, it must be admitted that $\widehat{\beta}_i$ almost certainly will not be exactly equal to β_i. There are other values besides the point estimate but near the point estimate that are consistent with the data in the sense that the data set supports the possibility that each may well be the true value of β_i. It's often desirable to present an interval of data-consistent values in addition to the point estimate. If so, a confidence interval is appropriate.

6.2.3 Confidence Intervals

A confidence interval for a parameter is a set of possible values for the parameter that are consistent with the data. For example, Table 6.1 shows an approximate 95% confidence interval for β_{fatalism}. It was calculated according to

$$[\widehat{\beta}_{\text{fatalism}} - 2\widehat{\sigma}(\widehat{\beta}_{\text{fatalism}}), \widehat{\beta}_{\text{fatalism}} + 2\widehat{\sigma}(\widehat{\beta}_{\text{fatalism}})] = [.217, .619].$$

We say that we are 95% confident that β_{fatalism} is between .217 and .619. All values not contained in the interval are considered to be inconsistent with the data and so we are confident that the true value is not *outside* of the interval. The term $2\widehat{\sigma}(\widehat{\beta})$ is referred is the margin of error. The width of the interval is the difference between the upper and lower bounds, and hence is approximately $4\widehat{\sigma}(\widehat{\beta})$, or twice the margin of error. Narrow confidence intervals are preferable to wide intervals, and an interval of negligible width is optimal since we would be 95% confident that the true value is essentially the estimated value.

The *confidence level* is the probability that the *procedure* will yield an interval containing the parameter. Mathematically, the confidence level is $1 - \alpha$ where $0 < \alpha < 1$. Standard choices for α are .01, .05 and .1. The confidence level can be interpreted as follows. If the exercise of collecting the sample and computing the confidence interval were repeated many times, then $100(1 - \alpha)\%$ of all confidence intervals will contain the parameter (provided that the inferential model (Eq. (6.2)) is correct). It should be noted that the parameter is either in a particular computed interval, say $[.217, .619]$, or it is not, but it's impossible for us to know which case is true. Therefore it's not correct to say that there is a .95 probability that β_{fatalism} is between .217 and .619 in value. The correct interpretation is that the probability of the *procedure* generating an interval that contains the true parameter value is $1-\alpha$. We cannot directly address whether or not β_{fatalism} is between .217 and .619 using probability. Hence, we settle for a heuristic, somewhat ambiguous statement and say that we are $100(1 - \alpha)\%$ confident that the parameter is between .217 and .619.

Whether or not the confidence interval captures the parameter with probability $1 - \alpha$ depends on two events. First, the interval must be centered on the true value β, which in turn depends on the adopted linear model being correct. If the relationship between $\mathrm{E}(Y|\mathbf{x})$ and $\mathbf{x}$ is not approximately linear, then the interval may not be centered on the true value. The accuracy of a linear approximation of a nonlinear relationship does not depend the sample size, so the failures of a poor model cannot be escaped by increasing the number of observations. Secondly, the standard error estimate $\widehat{\sigma}(\widehat{\beta}_i)$ must be an accurate estimate of the true variability in parameter estimates from sample to sample, and the accuracy of $\widehat{\sigma}(\widehat{\beta}_i)$ depends on several distributional conditions describing the population or process which will be discussed momentarily.

Parameter estimates and confidence intervals obtained from the Ginzberg data set are summarized in Table 6.1. Table 6.1 gives us some confidence in the conclusion that there is a relationship between depression and simplicity given fatalism since the 95% confidence interval for $\beta_{\text{simplicity}}$ is $[.178, .581]$. The term *given* expresses the idea that we have accounted for the effect of fatalism by including it in the model. The possibility that $\beta_{\text{simplicity}}$ is zero is inconsistent with the data since the smallest possible value for $\beta_{\text{simplicity}}$ that is consistent with the data is the lower bound, .178, nowhere near zero.

The discussion of confidence intervals up to this point assumed that the sample size n is not small. If n is small (say, less than 100), then an adjustment is made to accommodate the lack of precision stemming from small n. For values of $n-q$ less than 80, a more accurate $100(1-\alpha)\%$ confidence interval for β is

$$\widehat{\beta} \pm t^*_{n-q,\alpha/2}\widehat{\sigma}(\widehat{\beta}),$$

where $t_{n-q,\alpha/2}$ is the $100\alpha/2$ percentile from the (central) $\mathcal{T}$-distribution with $n-q$ degrees of freedom. The $\mathcal{T}$-distribution is very similar to the standard normal distribution though with somewhat longer tails when $n-q$ is less than 80. When $n > 80$, we routinely compute an approximate 95% confidence interval for β according to $\widehat{\beta} \pm 2\widehat{\sigma}(\widehat{\beta})$. The justification of the choice of 2 as a multiplier of $\widehat{\sigma}(\widehat{\beta}_i)$ is that the 2.5 percentile of the standard normal distribution is -1.96, slightly larger than -2.

6.2.4 Distributional Conditions

The confidence level claimed by a confidence interval is approximate since the accuracy depends a set of conditions being met that in reality cannot be met exactly. We've succinctly described these conditions as the inferential model (Eq. (6.2)). Given the importance of the conditions for inference, some further discussion is worthwhile. If the conditions are approximately met, then the stated confidence level is likely to be close to truth. The conditions are often referred to as assumptions, a term which is sometimes misinterpreted to mean that the analyst need only assume or pretend that the conditions hold for the problem and data under consideration. In fact, the conditions must be met at least approximately to obtain reliable results. Three of the conditions describe the distribution of the response variable and the fourth describes the observations. They are:

1. *linearity*: the relationship between expectation of Y and $\mathbf{x}$ is linear. In other words, the model $\mathrm{E}(Y|\mathbf{x}) = \mathbf{x}^T\boldsymbol{\beta}$ is correct.
2. *Constant variance*: for every value of $\mathbf{x}$, the distribution of Y about $\mathrm{E}(Y|\mathbf{x})$ has the same variance σ^2_ε.
3. *Normality*: for every value of $\mathbf{x}$, the distribution of Y is normal. Equivalently, $\varepsilon \sim N(0, \sigma^2_\varepsilon)$.

Conditions 1 and 2 specify the mean and variance of Y, and so the normal distribution condition can be combined with the mean and variance conditions. In brief, $Y \sim N(\mathrm{E}(Y|\mathbf{x}), \sigma^2_\varepsilon)$ for every value of $\mathbf{x}$.

4. *Independence*: The observations are realizations of n independent random variables. Specifically, y_i is a realization of a random variable Y_i, for $i = 1, \ldots, n$, and $Y_1, Y_2, \ldots, Y_n$ are independent random variables.

Collecting all the conditions as one leads to the condition that Y_i, for $i = 1, \ldots, n$, are independently distributed as $N(\mathbf{x}_i^T\boldsymbol{\beta}, \sigma^2_\varepsilon)$ random variables.

The importance of these conditions depends strongly on the intended purpose of a fitted model. If the linearity condition is violated, then the fitted regression model will be locally biased and will under- or over-estimate $\mathrm{E}(Y|\mathbf{x})$ for some values of $\mathbf{x}$. However, when the relationship between $\mathrm{E}(Y|\mathbf{x})$ and

$\mathbf{x}$ is not linear, then the linear model may still provide a reasonably accurate approximation to the unknown true relationship. If prediction is the sole objective, then nonlinearity in the relationship between $\mathrm{E}(Y|\mathbf{x})$ and $\mathbf{x}$ is tolerated provided that the fitted model is sufficiently accurate for its intended purpose. Investigating the extent to which the model fails is part of residual analysis, the subject of Sect. 6.8.

Violations of the constant variance condition have little effect on the parameter estimates and hence on the predictive utility of a fitted model. Nonconstant variance may bias the standard errors $\widehat{\sigma}(\widehat{\beta}_i)$. The consequence of biased standard errors is that confidence intervals are either too wide or too narrow and the probability that the procedure captures the parameter is not $1-\alpha$ as claimed.

The normality condition has relatively little bearing on the parameter estimates and the predictive utility of a fitted model. The condition is critical for confidence intervals only if the sample size is small (say, $n-q<80$) because the Central Limit Theorem insures that linear combinations of random variables will be approximately normal in distribution for large values of n. Certain conditions must hold for the Central Limit Theorem to apply but these conditions do not often fail to hold.

A common error is to investigate the normality of the responses $y_1,\ldots,y_n$. This is a mistake since the responses have different expectations, a situation that muddies the analysis of normality. Instead, the normality condition should be investigated using the residuals $\widehat{\varepsilon}_1=y_1-\widehat{y}_1,\ldots,\widehat{\varepsilon}_n=y_n-\widehat{y}_n$, or standardized versions of the residuals. Then, the task is to determine if the distribution of residuals is consistent with a normal distribution with mean 0. If the sample size is small and the residuals clearly are not normal in distribution, then confidence intervals may not be consistent with the claimed confidence level.

6.2.5 Hypothesis Testing

Behind each data set there lurks a population or process from which the data were collected. It's certainly not always true, but let's suppose that the realized data are a sample taken from the population or a set of realizations generated by a process. In this context, the random variables $Y_1,\ldots,Y_n$ represent a random mechanism that creates realizations $y_1,\ldots,y_n$. Our purpose for analyzing a set of realizations ultimately is not to learn about the realizations. Instead, our purpose is to learn about the underlying population or process that generated the realizations. We often want to answer the question of whether there really is a relationship between the response variable Y and one or more predictor variables. The question of a relationship between variables is a recurrent question in applied statistics, and hypothesis testing is routinely used to assess the strength of evidence supporting the contention that there is a relationship. In data science applications of linear regression, hypothesis testing may be applicable and useful, but often hypothesis testing is unsuited for the situation at hand. Some data scientists would argue that hypothesis tests and p-values should not be used at all. We ignore the protests of the purists and delve into the topic of hypothesis testing to understand the method and when it is suitable for data analytics.

Let us consider the linear regression model (formula (6.1)) and a test of whether a linear relationship exists between Y and X_i, the ith predictor variable. A hypothesis test begins with two competing and contradictory hypotheses,

$$\begin{aligned} &H_a : \beta_i \neq 0, \ \text{the alternative, and} \\ &H_0 : \beta_i = 0, \ \text{the null.} \end{aligned} \tag{6.5}$$

If $\beta_i = 0$ (hence, H_0 is true), then a linear relationship between the expected value of Y and the ith predictor variable does not exist and the variable has no explanatory or predictive value. On the other hand, if the data support H_a, then there is a justification for including the ith variable in the model of $\mathrm{E}(Y|\mathbf{x})$. The traditional statistical approach to model fitting is to retain only those variables for which the data support the alternative hypothesis. If the analyst adopts this approach, then for the most part, model fitting is a series of hypothesis tests conducted on different predictor variables or combinations of predictor variables.

The hypotheses stated in formulae (6.5) may be enlarged to test whether there's a linear relationship between Y and the predictors $X_j, \ldots, X_k$. In this case, the hypotheses are

$$\begin{aligned} &H_a : \ \text{at least one of } \beta_j, \ldots, \beta_k \text{ is not zero,} \\ &H_0 : \beta_j = \cdots = \beta_k = 0, \end{aligned} \tag{6.6}$$

where $0 \leq j < k \leq q$. If the test does not support H_a over H_0, then the entire set of variables should not be included in the model. Hypotheses 6.5 and 6.6 are tested using a sample. The term *test* is often a misnomer since we often stop short of arriving an unambiguous conclusion such as H_a *is true* and instead report the strength of evidence supporting H_a. In most applications, the p-value is reported along with some interpretation such as *there is strong evidence that X is linearly related to the expected value of Y*. Small values of the p-value support H_a. The p-value is an estimated probability and therefore is bounded by 0 and 1.

Occasionally, an accept or reject decision is made. The two possibilities are: reject H_0 in favor of H_a (thereby concluding that H_a is true), or fail to reject H_0 (and accept H_0). In the second case, H_0 is usually not concluded to be true; instead the conclusion is that there is insufficient evidence to rule out H_0 on the basis of the data. Its often true that more data will show H_0 to be incorrect.

Jumping ahead a little, Table 6.2 shows the results of two hypothesis tests that help answer the question of whether a linear relationship exists between expected depression score and the explanatory variables. The column headed by *t-statistic* contains test statistics measuring the strength of evidence supporting $H_a : \beta_i > 0$ and contradicting $H_0 : \beta_i \leq 0$. The *p-value* column shows the strength of evidence. Table 6.2 shows strong evidence of a linear relationship between depression and fatalism since the p-value is quite small. The same statement applies to depression and simplicity. A test involving β_0 is omitted since it is not germane to the question of association between variables.

Table 6.2 Parameter estimates and standard errors obtained from the linear regression of depression score on fatalism and simplicity. Tests of $H_0 : \beta_i \leq 0$ versus $H_a : \beta_i > 0$ are summarized by the t-statistic and the p-value

Variable	Estimate	Std. Error	t-statistic	P-value
Constant	.2027	.0947		
Fatalism	.4178	.1006	4.15	$< .0001$
Simplicity	.3795	.1006	3.77	.0002

Let us systematically develop the tests shown in Table 6.2. A determination of whether there is *any* association between Y and X_i requires a test of the hypotheses $H_0 : \beta_i = 0$ and $H_a : \beta_i \neq 0$. A test of this form is logical because $\beta_i = 0$ implies that there is no association between Y and X_i. Tests regarding β_0 are not often of much interest though occasionally a test of whether the intercept is a specific value is of interest. In any case, we will let β denote any particular parameter in $\{\beta_0, \beta_1, \ldots, \beta_p\}$.

Occasionally, there's a specific value for β different from zero which is to be tested. Let's call the value β_{null}. Any real number including zero may be used for β_{null}. Three pairs of hypotheses are routinely tested. They are

1. $H_a : \beta > \beta_{\text{null}}$ versus $H_0 : \beta \leq \beta_{\text{null}}$,
2. $H_a : \beta < \beta_{\text{null}}$ versus $H_0 : \beta \geq \beta_{\text{null}}$,
3. $H_a : \beta \neq \beta_{\text{null}}$ versus $H_0 : \beta = \beta_{\text{null}}$.

Table 6.2 uses the first form because our hypothesis (H_a) is that association exists between depression and, say fatalism, *and* the association is positive. We do not believe that the association could be negative, and so H_a differs from simply saying that an association exists.

For all three hypothesis pairs, the strength of evidence supporting H_a and against H_0 is determined by a test statistic. In this situation involving a single regression coefficient, the test statistic is a t-statistic given by

$$T = \frac{\widehat{\beta} - \beta_{\text{null}}}{\widehat{\sigma}(\widehat{\beta})},$$

where $\widehat{\sigma}(\widehat{\beta})$ is the standard error of $\widehat{\beta}$.

If the null hypothesis is true, then $\widehat{\beta}$ will be near β_{null} and the test statistic is expected to be near 0 because the numerator will be near 0. If H_a is true, and it states that $\beta > \beta_{\text{null}}$, then the realized value of T is expected to be supportive of H_a and therefore relatively large in magnitude and positive. Translating a realized value of T to a measure of support is accomplished by computing a p-value. Formally, the p-value is the probability of obtaining any value of T as or more unlikely than the realization t and, at the same time, supportive of H_a given that H_0 is true. So, if $H_a : \beta > \beta_{\text{null}}$ is the alternative, then large values of T are supportive of H_a, and

$$p\text{-value} = \Pr(T \geq t | H_0 \text{ is true}).$$

If the p-value is small, then an improbable outcome has been observed, casting doubt on the premise that H_0 is true. For example, it's usually said that there is evidence of a positive linear association between X_i and Y if the p-value is less than .05. The logic behind the statement is that an event has been observed that occurs less than once in twenty times when H_0 is true, and so either a fluke has occurred or H_0 is false and H_a is true.

Despite the analyst's beliefs,[6] the p-value is computed assuming H_0 to be true to convince a skeptical but objective disputant that believes H_0 to be true. In our effort to convince her that H_a is true, we accept her position for the sake of argument and carry out the test (therefore assuming that H_0 is true). If the p-value is very small, then an improbable event has occurred. Since the disputant is objective, she must admit that the premise upon which the p-value was computed may well be false. That premise is the null hypothesis. The objective disputant may not give up their belief in H_0, but will concede that we have provided legitimate evidence in favor of H_a and contradicting H_0.

To expand on the details of the p-value calculation, suppose that the observed value of T is t. Then, p-values are computed according to one of the following three scenarios.

1. If $H_a : \beta > \beta_{\text{null}}$, then p-value $= \Pr(T \geq t | H_0)$.
2. If $H_a : \beta < \beta_{\text{null}}$, then p-value $= \Pr(T \leq t | H_0)$.
3. If $H_a : \beta \neq \beta_{\text{null}}$, then p-value $= 2\Pr(T \geq |t| | H_0)$.

The p-values are computed assuming that $T \sim \mathcal{T}_{n-q}$. The third calculation doubles the right tail area of the $\mathcal{T}$-distribution. The logic in doubling the tail area is that values of T that contradict $H_0 : \beta = \beta_{\text{null}}$ and support $H_a : \beta \neq \beta_{\text{null}}$ may be either positive or negative and a measurement of the probability of observing a test statistic as or more contradictory to H_0 and in favor of H_a allows for values that are larger than $|t|$ and smaller than $-|t|$. A two-sided p-value is computed according to

$$\begin{aligned} p\text{-value} &= \Pr(T < -|t| | H_0) + \Pr(T > |t| | H_0) \\ &= 2\Pr(T > |t| | H_0), \end{aligned}$$

[6] The analyst often believes H_a to be correct.

since the $\mathcal{T}$-distributions are symmetric. Two final remarks: if n is sufficiently large (e.g., $n \geq 80$) and H_0 is true, then T is approximately standard normal in distribution and the standard normal distribution may be used to compute the p-value instead of the $\mathcal{T}_{n-q}$ distribution. Secondly, the procedure of specifying hypotheses, computing the test statistic and p-value, and making a statement regarding the strength of evidence is generally called a *test of significance*. As a rule of thumb, a p-value less than .01 may be described as strong evidence; a p-value between .01 and .05 is equivalent to evidence (no modifier); a p-value between .05 and .1 may be described as weak evidence; and a p-value greater than .1 provides little or no evidence in support of H_a and in contradiction of H_0.

To illustrate, there is no reason to believe that depression score and fatalism would be negatively associated even before seeing the data, but it is logical to postulate a positive association. Then, to measure the strength of evidence supporting the hypothesis of a (positive) association, the appropriate alternative hypothesis is $H_a : \beta > 0$ and the counter-hypothesis is $H_0 : \beta \leq 0$.

There's strong evidence of a linear association between depression score and fatalism, and likewise, strong evidence of a linear association between depression score and simplicity (Table 6.2). The adjusted coefficient of determination is .507 and so together, fatalism and simplicity explain almost 51% of the variation in depression score.

6.2.6 Cautionary Remarks

Much has been made of the inferential model and the necessity for it to be realistic. If the inferential model fails to be realistic, then the confidence intervals and p-values are not accurate. Most of the data used in this text were collected without benefit of random sampling as is the usual situation in data science. Hypothesis testing is not applicable in the vast majority of these cases.

Even when testing is appropriate, hypothesis testing is not very informative. We are often defeated by the alternative hypothesis statement that the parameter is simply larger than zero. If we look at it another way, H_a above states that $\beta_i \in (0, \infty)$. The alternative hypothesis is correct if β_i is infinitesimally larger than zero, in which case, we may well favor the null hypothesis and we *don't* want be in the position of reporting that there is an abundance of evidence favoring H_a even though it is true.

Let's look at the test statistic when there is only one predictor variable. The test statistic is

$$\begin{aligned} T &= \frac{\widehat{\beta}_1 - \beta_{\text{null}}}{\widehat{\sigma}(\widehat{\beta}_1)} \\ &\approx \sqrt{n}\frac{\widehat{\sigma}_x(\widehat{\beta}_1 - \beta_{\text{null}})}{\widehat{\sigma}_\varepsilon}, \end{aligned} \tag{6.7}$$

where $\widehat{\sigma}_x$ is the estimated standard deviation of the predictor variable. Both $\widehat{\sigma}_x$ and $\widehat{\sigma}_\varepsilon$ are largely unaffected by n, and the same holds for the difference $\widehat{\beta}_1 - \beta_{\text{null}}$. Thus, if data volume is massively large, T will be very large in magnitude because of the multiplier $\sqrt{n}$. For example, the combined Medicare data sets of Sect. 4.7 consisted of over 18 million records. There's no point in testing hypotheses about the variables—even the smallest of differences between $\widehat{\beta}$ and β_{null} lead to a large in magnitude T-statistic and a very small p-value. What is really needed is a test of the hypotheses $H_a : \beta_i > \eta$ versus $H_0 : \beta_i \leq \eta$, but determining η before the test is carried out (and never afterward) is hard, and tests of this form are rarely undertaken in practice.[7] Setting η after the model has been fit is poor practice since there is a temptation to arrange the test to support one's preconceptions, and even when the analyst is objectively testing after analysis, many observers are not inclined to trust the results.

You may ask: how did it come to be that hypothesis testing, the bread-and-butter of linear regression, is mostly unhelpful in data analytics? The reason is that in the past century, manual data collection meant that data was expensive and therefore, it was collected carefully, using designs that generated representative samples. Sample sizes were usually small and spurious results were possible. A method of measuring strength of evidence was needed to account for the uncertainty stemming from small sample sizes. Hypothesis testing was born from that need. Now, the needs are different, and the utility of hypothesis testing is greatly diminished.

6.3 Introduction to R

Statistical methods are routinely used in data analytics despite limitations associated with their application. Linear regression is probably the most heavily used of the many statistical methods useful for data analytics. Not only is linear regression an essential competency for data scientists, but so is the ability to use a mature and sophisticated statistical language.[8] Thus, we leave Python in this chapter and turn to a different environment for carrying out data analytics. The environment is the open-source statistical package R. Carrying out statistical calculations in R is usually easy. The R scripting language is mature (meaning old) and particularly easy to use for matrix-based calcu-

[7] Computationally, the test is easy to execute.

[8] The stipulation that the language is sophisticated eliminates Excel as a platform for statistical analysis.

lations. For the practicing data scientist, R is a good for analytics in which the emphasis is on statistical procedures but it's not a good choice for data processing.

The goal of the next series of tutorials is to gain competency in working with R, and in particular, working with data objects, `for` and `while` loops, conditional statements, and linear regression. An essential skill of data scientists is data visualization and so the reader will learn to construct a few widely used plots. The first tutorial assumes that the reader is familiar with the R interface and is able to create a "hello world"-type script and execute the code contained within. Two online tutorials (among many available for free) that will help the reader gain familiarity with the R environment are described in Chap. 1. Our tutorial assumes that R is being used in the integrated development environment RStudio though there are no actions described in the tutorial that are substantially different in the base R environment.

6.4 Tutorial: R

The Australian athletes data set [60] consists of measurements on hematology and morphology measurements collected from national-caliber athletes participating in 12 sports. The data set is available from the DAAG library [38] and provide a smooth introduction to regression analysis using R. The objective of the tutorial is to determine the relationship between skinfold thickness, percent body fat, and gender. Percent body fat is measured by submerging an individual in a water tank to determine the individual's displacement[9] and is believed to be the most accurate method of body fat measurement. Skinfold thickness is another method of measuring body fat that uses a caliper to measure the thickness of the subcutaneous fat underlying the skin. It's usually computed as an average of measurements at several sites. Measuring skinfold thickness is less time-consuming and expensive to determine than tank measurements of percent body fat, but concerns have been expressed regarding its accuracy for measuring body fat in children [49]. We set out to investigate the relationship between the two measurements using the Australian athletes data set.

1. Start R. We're assuming that R has been installed and that the user is familiar with the R environment.
2. Determine if the library DAAG is available by submitting the instruction `library(DAAG)` in the R console. If the response is *Error in library (DAAG) : there is no package called 'DAAG'*, install the library using the instruction

[9] Body weight is also used in a calculation that incorporates differences in the density of muscle, bone, and fat.

```
install.packages('DAAG')
```

Alternatively, packages may be installed through a drop-down menu. You may have to select a repository before installing the package.

3. Create a script file by opening a new file from the drop-down menu.
4. The first instruction of the script will load the functions and data sets from the DAAG library. The instruction is `library(DAAG)`. Run the instruction by clicking on the `Run` button.
5. Add a call to the function `head` to view the first six records of the data file. Add a call to the function `str` to determine the structure of each of the variables in the data set. The script should appear as so:

```
library(DAAG)
head(ais)
str(ais)
```

If you're using `RStudio`, then click on the `Source` button (above the edit window) to execute the script. Alternatively, highlight a code segment and press `Ctrl-Enter` to execute the code segment.

6. Run the command `?ais` at the console prompt to learn a little more about the variables. Note that three variables related to body fat are recorded for each athlete: percent body fat, skinfold thickness, and body mass index.
7. Add the instruction

```
plot(ais$ssf,ais$pcBfat)
```

to the script to plot percent body fat against skinfold thickness. The syntax `ais$ssf` extracts the variable `ssf` from the dataframe `ais` as a column vector. A dataframe is an object with several useful attributes such as column and row names. A dataframe may contain variables of different types such as quantitative and qualitative. In contrast, an `R` matrix can consist of only one type of variable, say numeric or factor. Create the plot.

8. There appears to be a strong linear relationship between the two variables. Add the fitted least squares line to the plot using the instruction

```
abline(lm(ais$pcBfat ~ ais$ssf))
```

The outer function, `abline`, adds a line to an existing plot. The inner function, `lm`, fits a linear model by regressing y on x if the argument is y~x and returns an object. The function abline extracts the slope and intercept from the object to create the line.

9. In the console, execute the instruction str(lm(ais$pcBfat ~ ais$ssf)). You'll see that the object returned from the call to lm has a number of attributes that are accessible using the $ operator. Most important of these are the parameter estimates. You may save the vector of parameter estimates as a vector b using the instruction

```
b = lm(ais$pcBfat ~ ais$ssf)$coefficients
```

Type b in the console to see the contents of b and press the Enter key.

10. Identify the male and female athletes by adding colored points to the plot:

```
males = which(ais$sex == 'm')
points(ais$ssf[males], ais$pcBfat[males], col = 'blue', pch = 16)
points(ais$ssf[-males], ais$pcBfat[-males], col = 'red', pch = 16)
```

The function which creates a vector of indexes of those rows for which the variable sex is equal to 'm'. The result is that males contains the indexes of the male athletes. The function points adds solid circles to the plot because the plotting character pch has been set to 16. The object -males removes the males from the ais$ssf data vector and so the third instruction identifies the females using red solid circles.

An examination of the plot reveals that for a particular value of skinfold thickness, females tend to have a greater percent body fat. The implication is that females have relatively more visceral fat than males with a comparable amount of subcutaneous fat.[10]

11. To view a statistical summary of the fitted regression model, add the following instructions to your script and execute the script:

```
lm.obj = lm(ais$pcBfat~ais$ssf)
summary(lm.obj)
confint(lm.obj)
```

The third line of the code segment computes and displays 95% confidence intervals for β_0 and β_1. The analyst should beware of assuming that the confidence interval procedure just invoked will capture the true parameter 95% of the time. The actual coverage rate depends on the extent to which these data conform to the conditions described in Sect. 6.2.4. Checking the conditions are discussed in Sect. 6.9.

[10] Visceral fat is located in the abdominal cavity.

12. Since the relationship is different between males and females, add a variable that identifies females to the model:

```
females = as.integer(ais$sex == 'f')
print(females)
```

The variable `females` is called an *indicator* and is defined mathematically case-wise:

$$x_{i,\text{female}} = \begin{cases} 1, \text{ if the } i\text{th athlete is female,} \\ 0, \text{ if the } i\text{th athlete is male.} \end{cases} \tag{6.8}$$

We incorporate the indicator variable into a model of expected percent body mass:

$$\mathrm{E}(Y_i|\mathbf{x}_i) = \beta_0 + \beta_1 x_{i,\text{ssf}} + \beta_2 x_{i,\text{female}}. \tag{6.9}$$

Fit this model using the function call `lm(ais$pcBfat~ais$ssf+females)`. The resulting fitted model for the ith athlete can be expressed as

$$\begin{aligned} \widehat{y}_i &= 1.131 + .158 x_{i,\text{ssf}} + 2.984 x_{i,\text{female}} \\ &= \begin{cases} 1.131 + .158 x_{i,\text{ssf}}, \text{ if the } i\text{th athlete is male,} \\ 4.115 + .158 x_{i,\text{ssf}} \text{ if the } i\text{th athlete is female,} \end{cases} \end{aligned}$$

since $4.115 = 2.984 + 1.131$. Female Australian athletes are estimated to carry 2.98% more body fat than male Australian athletes if we compare male and female athletes with the same skinfold thickness. Verify that a 95% confidence interval for the true difference is $[2.62\%, 3.35\%]$. Since zero is not included in the interval, we conclude that there truly is a difference between female and male athletes with respect to the distribution of body fat (subcutaneous versus visceral).

13. The summary table produced by passing the object `lm.obj` to the `summary` function contains a measure of the proportion of variation in percent body fat explained by the regression model. This statistic is the adjusted coefficient of determination, R^2_{adjusted} (formula (3.36)). The values of R^2_{adjusted} for the models with and without the female indicator variable are .986 and .927, respectively. Both predictive models are accurate, but the model accounting for differences between gender is more accurate. The *unexplained* variation is reduced by $100(.986 - .927)/(1 - .927) = 80.8\%$ by adding the female indicator variable to the first model.

14. Add a blue line to the plot showing the estimated expected percent body fat as a function of skinfold thickness for the males:

```
abline(a = 1.131, b = .158, col='blue')
```

15. Add a red line to the plot showing the fitted model for females.

16. Investigate whether there are differences in percent body fat among athletes grouped by sport. Use a boxplot[11] to visualize the distributions by sport:

```
boxplot(ais$pcBfat ~ ais$sport, cex.axis=.8, ylab = 'Percent body
       fat')
```

    The axis labels have been reduced in size by setting the argument `cex.axis` so that all of the names are visible.[12]

6.4.1 Remark

The tutorial showed that skinfold thickness is a good proxy for percent body fat. We infer that analyses of body fat can be conducted using skinfold thickness rather than percent bodyfat with little loss of accuracy particularly if the analyses are within-gender (that is, by conducting separate analyses for males and females).

6.5 Tutorial: Large Data Sets and R

The objective of this tutorial is to develop some techniques for processing large data sets with `R`. `R` has been built for statistical analysis, not data analytics, and so some additional time and effort usually is needed to process larger data sets compared to using `Python`.

The Consumer Financial Protection Bureau is a federal agency with the responsibility of enforcing federal consumer financial laws and protecting consumers from malfeasance on the part of the providers of financial services and products. The Consumer Financial Protection Bureau maintains a database of consumer complaints. The database is periodically updated and so the contents change over time. The objective of this tutorial is to determine which commercial product is most often on the receiving end of a complaint to the Consumer Financial Protection Bureau.

1. Download the data file from https://catalog.data.gov/dataset/consumer-complaint-database.
2. Open a Linux terminal window or a Windows Command Prompt window (see Chap. 4.6, instruction 1a). Write the first few lines of the file to the console using the Linux command `cat` or the Windows command `type`. The first record of the data file is a list of attributes.

[11] Boxplots were discussed briefly in Chap. 3.6.1.

[12] Netball is played only by women.

3. The usual method of consuming data using R is to read the entire set into memory.[13] We will have to read the data file line-by-line to handle missing data. Open a new script file. Put the following commands in the script and revise the file path to match the location of the file on your computer:

```
fileName = ".../Consumer_Complaints.csv"
x = read.table(fileName, sep=’,’, header=TRUE)
```

   The file has a csv (comma separated values) extension which identifies the structure as using commas to delimit variables. There’s no guarantee that every record will have the same number of commas though.

   Run each line separately (highlight the line and press Ctrl+Enter to execute the line). The first line should not generate an error. Run the second line. The read.table function attempts to read the data file into a rectangular (row by column) data frame. It’s likely that the command will fail with the message that one particular row did not have the same number of elements as column names in the data file header. Traditionally, an analyst would open the data file in an editor and attempt to locate and correct the error. An effort of this type is impractical with a data set of this size as there may be too many errors to manually correct all of them. The 2014 edition of the data file is approximately 175 megabytes. A different strategy must be pursued.
4. Two techniques will be used to handle the file. First, blocks of one or a few (say, 1000) lines will be read at once so that memory limitations are not encountered. Secondly, each block will be read as a single character string so that missing or mildly corrupted data will not generate an error during the read operation. Add the following code segment to your script. By executing the code, you will open the file and read one line. The argument n = 1 instructs the interpreter to read one line.

```
f  = file(fileName, open = "r")   # Open the file for reading.
names = readLines(f, n = 1)       # Read the first line. It
                                  # contains a comma-delimited
                                  # list of attributes.
print(names)
names = strsplit(names, ’,’)      # Split the string into a
                                  # vector of names.
print(names)
```

5. There are nine categories of commercial products. We will determine the relative frequency of occurrence for each product and the proportion of

[13] The tutorial of Sect. 6.4 used a data set that was loaded into the object ais when the DAAG library was invoked with the library command.

times that a complaint about the product was settled in a timely manner. Set up a vector containing the category of products:

```
products = c(’Bank account or service’,’Credit card’,’Credit
              reporting’,
  ’Debt collection’,’Money transfers’,’Mortgage’,’Payday loan’,
  ’Prepaid card’,’Student loan’)
```

6. Initialize a vector to store the frequencies of occurrence for each product and a matrix to store the evaluation of whether the complaint was settled in a timely manner.

```
countVector = rep(0,length(products))
timelyMatrix = matrix(0, length(products), 2)
```

The matrix `timelyMatrix` will contain the number of *yes* responses and *no* responses. Each row of `timelyMatrix` corresponds to a product and the two columns contain the counts of *yes* and *no* responses, respectively.

7. In the console, execute the command `block <- readLines(f, n=10)`. The file object `f` must be open. You may have to execute the command `f = file(fileName, open = "r")` discussed in instruction 4. A block of $n = 10$ strings will have been read and stored in the array named `block`.[14] Print the contents: `print(block)`. Verify that `block` is a vector of length n in which each element is a character string by typing `str(block)` in the console. The function `str()` displays the structure of the argument.
8. Verify that the second entry of `block` is a character string by submitting the instruction `block[2]` at the console. Unlike `Python`, `R` uses one-indexing; hence, the first element of a vector is indexed by 1.
9. Split the string apart at the commas, display the result, and extract the second element:

```
lst = strsplit(block[2],’,’)
print(lst)
print(lst[[1]][2])
```

Splitting the block produced a list. You may verify the length of the list is one using the instruction `length(lst)`. But, the *item* in the list is a character vector of 18 or more items (type `length(lst[[1]])`). The items in the character vector are the data we're interested in. The character vector can be extracted using the instruction `unlist(lst)`.

The records contain variable numbers of items. This explains why `read.table()` failed. The function `read.table()` stores the data in a

[14] Strings are delimited by the end-of-line character at the end of each record.

table structure consisting of rows and columns by splitting the string at the commas. When a string was encountered containing more or less than the expected number of items, `read.table()` terminated with an error message.

10. In your script, build a `while` loop for processing the file. The loop processes the file by reading blocks and processing each line in the block. Here is the basic structure:

```
f  = file(fileName, open = "r")   # Open the file for reading.
counter = 0
test = TRUE
while (test){
    block <- readLines(f, n = 1000)
    counter = counter + length(block)
    print(c(counter, length(block)))
    if (length(block) == 0) break
}
close(f)    # Destroy the file object and close the file.
```

The instructions enclosed by the braces (`{` and `}`) will be executed in sequence and repeatedly until `block` is empty and the length of `block` is zero. Then, the `break` command instructs the `R` interpreter to terminate the `while` loop.

The `while` loop consumes blocks of 1000 lines. A column vector consisting of `counter` and `length(block)` is constructed by the statement `c(counter, length(block))`. It's necessary to create a vector because the `print` statement will only accept one argument.

11. Returning to the script, insert a `for` loop within the `while` loop to extract the product from each record in the block.

```
for (i in 1:length(block)){
    record = unlist(strsplit(block[i],',','))
    product = record[2]
    print(product)
}
```

We're converting `record` from a list to a vector using the command `unlist` so that sub-scripting is simpler. Execute the code. You may interrupt execution after noting that the second entry in `record` may not be a product.[15] Therefore, it's necessary to test that `product` is an element in the set of products.

[15] With our particular version of the complaint file, there are a variety of other attributes in the second position of `record`.

12. We will attempt to find a match between the extracted string and one of the products in the vector products. If there's a match, then the count for the product is incremented.

```
w = which(products == product)
if (length(w) > 0) {
      countVector[w] = countVector[w] + 1
}
```

Insert this code segment in the for loop that processes each string in block (instruction 11).

The function which is very useful—when passed a boolean vector, it creates a vector of indexes that identifies the boolean elements that are true. The statement products == product creates the boolean vector with the same length as products. If there are no matches, meaning that product is not a product but some other string, then w has length zero.

13. Process the entire file. After completion, print out the contents of productVector and countVector. Order the products from smallest to largest count and print the product names and frequencies of each product:

```
index = order(countVector)
orderedProducts = products[index]
orderedCounts = countVector[index]
```

The vector index contains the indexes that will put countVector in order from smallest to largest value. When index is used as a vector of indexes for products, it arranges the elements in orderedProducts to correspond with the ordered counts in the vector orderedCounts.

14. Create a dataframe containing the product names, counts of occurrences, and the percentages of each product (in order from smallest to largest):

```
df = data.frame(orderedProducts, orderedCounts,
    round(100*orderedCounts/sum(orderedCounts), 2))
print(df)
```

The dataframe df consists of a column of characters containing the product names and two numeric columns.

15. Should you have problems with program flow, the critical instructions are

```
while (test){
    block <- readLines(f, n=10000)
    if (length(block) == 0) break
    for (i in 1:length(block)){
        if (length(w) > 0) {
            countVector[w] = countVector[w] + 1
        }
    }
}
```

16. The `names` vector contains the name of the attribute recording the complainants response to the question of whether the company responded to the complaint in a timely manner. The index of the timely response variable varies with year, but let’s assume that the correct index is 17 (as it was in May, 2016).

    Check that the answer is either yes or no, and if it is, then increment the counter for the product. There’s a counter for each response (yes or no) for each product. Add the code segment to the `for` loop over blocks (instruction 11).

```
if (record[17] %in% c(’Yes’,’No’)) {
    timelyMatrix[w,1] = timelyMatrix[w,1] + (record[17] ==’Yes’)
    timelyMatrix[w,2] = timelyMatrix[w,2] + (record[17] ==’No’)
}
```

    The code `(record[17] %in% c(’Yes’,’No’)` returns a boolean value depending on whether or not the timely response string is in the vector `c(’Yes’,’No’)`. If a boolean variable is added to a numeric variable, the boolean is automatically converted to 0 if the boolean is false. The value is converted to 1 if the boolean is true. In this code segment, the result of the evaluation `record[17] ==’Yes’` is boolean (true or false), so we may add the result to `timelyMatrix[w,1]`.

17. Print the results in order of least to most complaints after all of the records have been processed and the file closed. Do this by creating a vector named `index` that orders the vector of counts. The ordering vector is used to create a new dataframe from `df` containing the product names, the counts, percentages, and the proportion of timely responses in the desired order.

```
tv = round(timelyMatrix[,1]/rowSums(timelyMatrix), 2)
cv = round(100*countVector/sum(countVector), 2)
df = data.frame(products,  countVector, cv, tv)
index = order(countVector)
print(df[index, ])
```

The rowSums function computes the sum of each row of a matrix and stores the sums in a vector with the same number of rows as the matrix. The syntax df[index,] creates a data frame ordered by index. The missing column index in the reference df[index,] instructs the R interpreter to put all columns of df in the data frame. The end result will be a table similar to (but different from) Table 6.3.

Table 6.3 Distribution of consumer complaint types obtained from $n = 269{,}064$ complaints lodged with the Consumer Financial Protection Bureau between January 2012 and July 2014

Type	Relative frequency	Count	Fraction of timely responses
Payday loan	.00	1012	.476
Money transfers	.00	1231	.981
Consumer loan	.03	7696	.925
Student loan	.03	8183	.959
Credit reporting	.12	32,547	.983
Debt collection	.12	33,478	.376
Bank account or service	.13	33,877	.482
Credit card	.14	37,554	.984
Mortgage	.42	113,486	.072

6.6 Factors

A factor is a qualitative explanatory variable. The consumer complaint type variable of the tutorial of Sect. 6.5 is an example. The values of a factor cannot be unambiguously ordered and arithmetic operations cannot be carried out using the values. Gender is another example. Arithmetic operations with the values *male* and *female* are impossible to define in a manner that is not open to debate. To distinguish factors from quantitative variables, the term *levels* is sometimes is used in place of values when discussing factors (hence, female is a level of gender). In a regression analysis, factors often identify subpopulations that potentially differ with respect to the relationship between the response variable and another predictor variable. For example, the analysis of the Australian athletes data set investigated the relationship between percent body fat and skinfold thickness. Figure 6.2 strongly suggests that the relationship between skinfold thickness and percent body fat differs between females and males. Specifically, it appears that percent body fat of females is, on average greater than males with equal skinfold thickness. In other words, the intercept of the female regression line is greater than the intercept of the male regression line in Fig. 6.2.

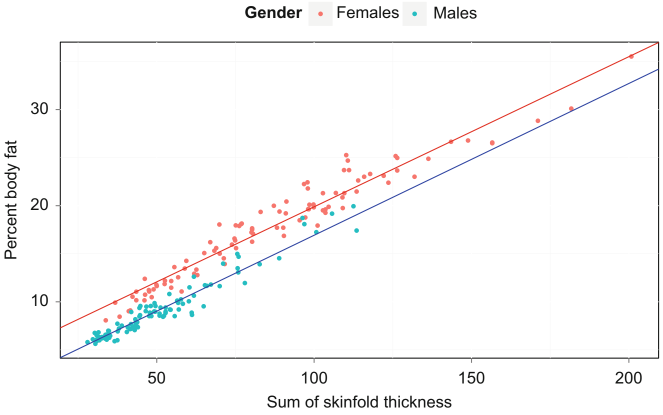


Fig. 6.2 Percent body fat plotted against skinfold thickness for 202 Australian athletes. Also shown are least squares regression lines that approximate the relationship between percent body fat and skinfold thickness for each gender

Other analyses of these data may benefit from introducing gender as an explanatory variable. Furthermore, it may be useful to introduce sport as a factor in regression analyses because the morphology of the athletes and physiological demands differ among sports.

Factors are accommodated in linear regression analysis by using indicator variables. We set up an indicator variable in Sect. 6.4, Eq. (6.8) and used it to create Fig. 6.2. Generically, an indicator variable for level A of a factor is defined according to

$$x_{i,\mathrm{A}} = \begin{cases} 1, & \text{if the } i\text{th observation was made at level } A, \\ 0, & \text{if the } i\text{th observation was not made at level } A. \end{cases}$$

Let's examine the model of expected percent body fat as a function of skinfold thickness (ssf) defined by Eq. (6.9). The model implies that each gender has a possibly different intercept. If the ith athlete is a male, then $x_{i,\text{female}} = 0$ and the model is

$$\mathrm{E}(Y_i|\mathbf{x}_i) = \beta_0 + \beta_1 x_{i,\text{ssf}}.$$

On the other hand, if the ith athlete is a female, then the model is

$$\begin{aligned} \mathrm{E}(Y_i|\mathbf{x}_i) &= \beta_0 + \beta_1 x_{i,\text{ssf}} + \beta_2 \times 1 \\ &= (\beta_0 + \beta_2) + \beta_1 x_{i,\text{ssf}} \\ &= \beta_0^* + \beta_1 x_{i,\text{ssf}}, \end{aligned}$$

where $\beta_0^* = \beta_0 + \beta_2$. If $\beta_2 = 0$, then $\beta_0^* = \beta_0$ and skinfold thickness is unrelated to gender. The model term $\beta_2 x_{\text{female}}$ accounts for the difference in mean or expected percent body fat between females and males given that skinfold thickness is the same. In summary, the relationship between percent body fat and skinfold thickness is modeled by parallel lines offset by the quantity $\widehat{\beta}_2$. Parameter estimates and tests of significance for the model

Table 6.4 Parameter estimates and standard errors obtained from the linear regression of skinfold thickness on percent body fat. The p-value associated with skinfold thickness was computed for $H_0 : \beta_1 = 0$ versus $H_a : \beta_1 > 0$. The p-value associated with β_2 is for the two-sided alternative $H_a : \beta_2 \neq 0$

Variable	Parameter estimate	Std. Error	t-statistic	p-value
Constant	1.131	.184	·	·
Skinfold thickness	.1579	.0029	55.11	$< .0001$
Gender (female)[a]	2.984	.186	16.03	$< .0001$

[a] Gender is an indicator variable taking on the value 1 if the athlete is female

are shown in Table 6.4. The table entry for the gender indicator variable summarizes a test of $H_0 : \beta_2 = 0$ versus $H_a : \beta_2 \neq 0$. The summary shows strong evidence of a difference in skinfold thickness between females and males ($t = 16.03$, p-value $< .0001$), a result that was uncovered in the Sect. 6.4 tutorial. The alternative hypothesis regarding skinfold thickness (line 2) is *one-sided* since the only logical alternative to the hypothesis that there is no relationship between skinfold thickness is that there is a positive linear relationship between skinfold thickness and percent body fat. Therefore, the p-value is $\Pr(T \geq 55.11)$ or half of $\Pr(T \geq 55.11) + \Pr(T \leq -55.11)$, the value presented in the `R` summary of the fitted regression model. There's no need to do anything though, since $\Pr(T \geq 55.11) + \Pr(T \leq -55.11)$ is less than .0001. With 202 observations, presenting a p-value with more than three or four significant digits conveys an artificial sense of accuracy.

If there are $r + 1$ levels of a factor, then r indicator variables are needed to identify the level of an observation. With gender, $r = 1$. If $x_{i,\text{female}} = 1$, then observation i was obtained from a female, and if $x_{i,\text{female}} = 0$, then observation i was obtained from a male. We do not need another indicator variable to identify males. If $r > 1$, and all r indicator variables are zero for observation i, then the observation must have been obtained at level $r + 1$. An indicator variable for the $r + 1$st level of a factor in the regression is not included as it will cause varying degrees of problems depending on the sophistication of the model fitting algorithm. The source of the fitting problem is that the $r + 1$ indicator variables are not linearly independent. Then, $\mathbf{X}^T\mathbf{X}$ will not be full rank and not invertible. You may investigate the issue by creating an indicator variable for the males and introducing it into the model containing the indicator variable for females.

6.6.1 Interaction

A question that often arises in problems that involve a quantitative variable and one or more factors is whether completely separate regression models are appropriate for each level of a factor. Table 6.4 presents results for a model in which the regression lines are *not* completely separate because the male and female models have a common slope, $\widehat{\beta}_1 = .158$. Different slopes can be permitted by including an *interaction variable* between the quantitative variable and each indicator variable associated with a factor level. As there are only two levels of gender (male and female), there is only one indicator variable. The *interaction*, or unconstrained model is

$$\mathrm{E}(Y_i|\mathbf{x}_i) = \beta_0 + \beta_1 x_{i,\mathrm{ssf}} + \beta_2 x_{i,\mathrm{female}} + \beta_3 x_{i,\mathrm{interaction}}, \tag{6.10}$$

where

$$\begin{aligned} x_{i,\mathrm{interaction}} &= x_{i,\mathrm{ssf}} \times x_{i,\mathrm{female}} \\ &= \begin{cases} x_{i,\mathrm{ssf}}, & \text{if the } i\text{th individual is female,} \\ 0, & \text{if the } i\text{th individual is male.} \end{cases} \end{aligned}$$

Consequently, if the ith athlete is a male, the model is

$$\begin{aligned} \mathrm{E}(Y_i|\mathbf{x}_i) &= \beta_0 + \beta_1 x_{i,\mathrm{ssf}} + \beta_2 x_{i,\mathrm{female}} + \beta_3 x_{i,\mathrm{ssf}} \times x_{i,\mathrm{female}} \\ &= \beta_0 + \beta_1 x_{i,\mathrm{ssf}} + \beta_2 \times 0 + \beta_3 x_{i,\mathrm{ssf}} \times 0 \\ &= \beta_0 + \beta_1 x_{i,\mathrm{ssf}}. \end{aligned}$$

If the ith athlete is a female, the model is

$$\begin{aligned} \mathrm{E}(Y_i|\mathbf{x}_i) &= \beta_0 + \beta_1 x_{i,\mathrm{ssf}} + \beta_2 x_{i,\mathrm{female}} + \beta_3 x_{i,\mathrm{ssf}} \times x_{i,\mathrm{female}} \\ &= \beta_0 + \beta_1 x_{i,\mathrm{ssf}} + \beta_2 \times 1 + \beta_3 x_{i,\mathrm{ssf}} \times 1 \\ &= (\beta_0 + \beta_2) + (\beta_1 + \beta_3) x_{i,\mathrm{ssf}}. \end{aligned}$$

With this set-up, β_0 is determined entirely by those observations obtained from males and β_2 is the difference between β_0 and the intercept that would be computed using only females in a regression of percent body fat on skinfold thickness. Consequently, $\beta_0 + \beta_2$ is the intercept that would be computed in a female-only regression. A parallel set of interpretations can be formulated for the slope parameters β_1 and β_3. The fitted interaction model is summarized in Table 6.5. There is no evidence that the slope depends on gender ($t = -1.07$, p-value $= .284$), and so we adopt model (6.9) over the interaction model. We prefer no-interaction models as they are simpler to explain and use.

Table 6.5 Parameter estimates and standard errors obtained for the interaction model (formula (6.10)). Significance tests of no interest have been omitted

	Parameter			
Variable	estimate	Std. Error	t-statistic	p-value
Constant	.849	.320	·	·
Skinfold thickness	.163	.0058	·	·
Gender (female)	3.416	.4429	·	·
Interaction	−.0072	.0067	−1.07	.284

Table 6.5 omits the test of significance for the individual variables gender and skinfold thickness for two reasons. Recall that in the absence of an interaction term, a test of $H_a : \beta_1 \neq 0$ tests whether skinfold thickness is linearly associated with percent body fat. With the interaction term in the model, a test of $H_a : \beta_3 \neq 0$ amounts to a test of whether skinfold thickness is linearly related to percent body fat *and* the relationship depends on gender. Rejecting the null hypothesis and concluding that $H_a : \beta_3 \neq 0$ is true implies skinfold thickness is linearly related to percent body fat. There's no need for a second test.

The second reason for not removing x_{female} from the interaction model is that if were removed, then the resulting model allows females and males to have different slopes but constrains the intercept to be the same for both genders. In many cases, the common intercept constraint is not defensible from a scientific standpoint. Problem 6.3 asks the reader to show that the intercept is common to both genders if x_{female} is removed from the model.

To expand on the use of factors, we'll investigate oxygen-carrying capacity of the athletes' blood using red blood cell count as a response variable. Let's consider both gender and sport as factors. The linear model fitting function `lm` will construct indicator variables accounting for the difference in mean red blood cell count between a reference sport and each of the other sports provided that `R` recognizes the variable `sport` as a factor.[16] We set the reference sport to be rowing using the instruction `ais$sport = relevel(ais$sport,'Row')` rather than accept the arbitrary level that would otherwise be designated by `R`. The reference level is the level with no indicator variable. The parameter associated with a non-reference level then measures the average difference between the reference level (rowing) and the non-reference level.

The two-factor model is

$$\mathrm{E}(Y_i|\mathbf{x}_i) = \beta_0 + \beta_{\text{male}} x_{i,\text{male}} + \beta_{\text{gym}} x_{i,\text{gym}} + \cdots + \beta_{\text{wPolo}} x_{i,\text{wPolo}},$$

where, for an arbitrary sport besides rowing,

$$x_{i,\text{sport}} = \begin{cases} 1, & \text{if the } i\text{th athlete participates in the sport,} \\ 0, & \text{if the } i\text{th athlete does not participate in the sport.} \end{cases}$$

Then, β_{sport} is the mean difference in red blood cell count between participants in the named sport and rowing provided that gender is held fixed. The next question to address is whether there are differences among sports with respect to mean red blood cell count. For example, a positive value for $\widehat{\beta}_{\text{T\_Sprnt}}$ implies that the mean red blood cell count of track and field sprinters is greater that the mean red blood cell count of rowers.[17] While

[16] `R` does recognize the variable `sport` as a factor so no action is needed. A variable `x` can be converted to a factor using the function call `x=as.factor(x)`.

[17] The evidence supports the statement.

comparisons of each level to the reference level may be of central interest,[18] it's often the case that a more general hypothesis is preferred—an alternative hypothesis that states that the factor is related to the mean level of the response variable and a null hypothesis that specifies no relationship. The extra-sums-of-squares F-test is used to test these hypotheses.

6.6.2 The Extra Sums-of-Squares F-test

Suppose that a factor is accounted for by a set of r indicator variables and that the parameters associated with the indicator variables are $\beta_{i+1}, \beta_{i+2}, \cdots, \beta_{i+r}$. A test of whether the response depends on the factor is equivalent to a test the hypotheses

$$\begin{aligned} &H_0 : \beta_{i+1} = \beta_{i+2} = \cdots = \beta_{i+r} = 0 \text{ versus} \\ &H_a : \text{ at least one of } \beta_{i+1}, \beta_{i+2}, \ldots, \beta_{i+r} \text{ is not } 0. \end{aligned} \tag{6.11}$$

For example, $\beta_{i+1}, \beta_{i+2}, \ldots, \beta_{i+r}$ may be the $r = 9$ parameters accounting for mean differences in red blood cell count between the reference sport rowing and the other sports. The null hypothesis imposes a constraint on the r parameters stating that each of the parameters is zero in value. Consequently, there are no differences with respect to $\mathrm{E}(Y|\mathbf{x})$ between the reference and every other level. No differences between the reference level and every other level implies that there are no differences at all among factor levels with respect $\mathrm{E}(Y|\mathbf{x})$ and hence, the factor is not related to $\mathrm{E}(Y|\mathbf{x})$. The alternative hypothesis relaxes the equality constraint and allows the parameters to be any value. When any parameter differs from 0, then $\mathrm{E}(Y|\mathbf{x})$ depends on the factor. This last condition is equivalent to the alternative hypothesis statement that at least one level differs from the reference level.

The extra-sums-of-squares F-test provides a test of the hypotheses (6.11) and, therefore, provides a test of whether the factor is related to the mean of the response variable. A more expansive interpretation of the extra-sums-of-squares F-test helps understand when it may and may not be used. The test compares the fit of two competing models when *one model is a constrained version of the other.* The test statistic compares lack-of-fit, or error, associated with the constrained version of the model to the lack-of-fit of the unconstrained version of the model. If there's a big difference in error between the models, then the evidence supports the better-fitting model. Let's develop the idea with more rigor.

Lack of fit for a given model is quantified by the residual sums-of-squares $\sum(y_i - \widehat{y}_i)^2$. The residual sums-of-squares cannot increase if an additional variable is entered into a model since setting the new parameter to 0 recovers the original (and constrained) model. Additional variables always reduce the

[18] For example, the reference level may be a control group in an experiment.

residual sums-of-squares. Thus, the residual sums-of-squares for the unconstrained model, SSR_u, cannot be more than the residual sums-of-squares for the constrained model, SSR_c. Hence, $\mathrm{SSR}_u \leq \mathrm{SSR}_c$.

The estimator of σ_ε^2 is the estimated residual variance associated with the unconstrained model:

$$\widehat{\sigma}_u^2 = \frac{\mathrm{SSR}_u}{n - p_u},$$

where p_u denotes the number of coefficients that parametrize the unconstrained model. We use the residual variance about the unconstrained model since even if the factor is not related to $\mathrm{E}(Y|\mathbf{x})$, $\widehat{\sigma}_u^2$ will be reasonably accurate. On the other hand, if the factor is related to $\mathrm{E}(Y|\mathbf{x})$, then the estimated residual variance associated with the incorrect and constrained model will over-estimate σ_ε^2. The test statistic is the scaled difference in residual sums-of-squares:

$$\begin{aligned} F &= \frac{\mathrm{SSR}_c - \mathrm{SSR}_u}{r\widehat{\sigma}_u^2} \\ &= \frac{\mathrm{MS}_{\text{lack-of-fit}}}{\widehat{\sigma}_u^2}, \end{aligned}$$

where $\mathrm{MS}_{\text{lack-of-fit}} = (\mathrm{SSR}_c - \mathrm{SSR}_u)/r$ is the mean square error attributable to lack of fit. If H_0 is correct, then F has an $\mathcal{F}$ distribution with r numerator and $n - p_u$ denominator degrees of freedom. Large values of F reflect large differences in the residual sums-of-squares. Therefore, large F-values contradict H_0 and support H_a and the p-value is defined to be the upper tail area of the $\mathcal{F}_{r,n-p_u}$ distribution. Mathematically,

$$p\text{-value} = \Pr(F \geq f|H_0),$$

where f is the observed value of the test statistic.

Table 6.6 provides a summary of the extra-sums-of-squares F-test for sport. There is convincing evidence that the unconstrained model fit is better than the fit of the constrained model ($F_{9191} = 5.70$, p-value $< .0001$) and so it is concluded that mean red blood cell counts differ among athletes with respect to participating sport and after accounting for gender differences. Looking closer at Table 6.6, the line labeled *Lack-of-fit* shows $\mathrm{SSR}_c - \mathrm{SSR}_u$,

Table 6.6 Details of the extra-sums-of-squares F-test for sport. The F-statistic shows strong evidence that mean red blood cell counts differs among athletes with respect to participating sport

Source of variation	Residual sum-of-squares	Degrees of freedom	Mean square	F-statistic	P-value
Constrained model	22.618	200			
Lack-of-fit	4.791	9	.5324	5.70	$< .0001$
Unconstrained model	17.826	191	.0933		

the difference in residual sums-of-squares between the model without sport included and the model with sport included as a factor. The mean square error attributable to lack-of-fit is $\text{MS}_{\text{lack-of-fit}} = .5324$. The value of the F-statistic is $F = .5324/.0933 = 5.70$ and p-value $= \Pr(F \geq 5.70) < .0001$.

The R function anova will compute the extra sums-of-squares F-statistic. Two arguments are passed to the function. The left argument is the linear model object obtained from the constrained model and the right argument is the linear model object obtained from the unconstrained model. For example, a test for interaction between gender and sport is produced by code segment

```
lm.constr = lm(rcc~sex+sport,data=ais)
lm.unconstr = lm(rcc~sex*sport,data=ais) # interaction model
anova(lm.constr,lm.unconstr)
```

The output from the function call is

```
Model 1: rcc ~ sex + sport
Model 2: rcc ~ sex * sport
  Res.Df    RSS Df Sum of Sq      F Pr(>F)
1    191 17.826
2    185 17.376  6   0.45023 0.7989 0.5719
```

From the output, we can see that $n - p_c = 191$ and $n - p_u = 185$, and hence the numerator degrees of freedom is $r = 6$. The participants of seven sports are both males and females. For the remainder of the sports, only one gender participates. Therefore, six indicator variables are necessary to allow the effect of each sport to be independent of gender. The residual sums-of-squares for the constrained model is $\text{SSR}_c = 17.826$ and the sums-of-squares for the unconstrained model is $\text{SSR}_u = 17.376$ and the difference is .4502. The realized value of the F-statistic is

$$F = \frac{.4502/6}{17.376/185} = .799.$$

The results of the test are summarized in Table 6.7. The residual sums-of-squares for the constrained model is $\text{SSR}_c = 17.826$ in this last test is also the residual sums-of-squares for the unconstrained model shown in Table 6.6.

Table 6.7 The extra-sums-of-squares F-test for interaction between sport and gender. There's no evidence of interaction between participating sport and gender (p-value = .57)

Source of variation	Residual sum-of-squares	Degrees of freedom	Mean square	F-statistic	P-value
Constrained model	17.826	191			
Lack-of-fit	.450	6	.0750	.7989	.5719
Unconstrained model	17.376	185	.0939		

The tutorial of Sect. 6.7 provides the reader with the opportunity to analyze effect of factors in a bike share program.

6.7 Tutorial: Bike Share

Bike sharing systems allow individuals to rent a bicycle at a particular location within a city and return it at a different location. These systems are a relatively inexpensive way to improve the quality of life in urban areas by reducing traffic load and providing recreational opportunities. One problem that faces bike sharing systems is insuring that bicycles are available at all locations with a high degree of certainty. Human behavior works against widespread and consistent availability, though. On weekday mornings, there's a general movement of bicycles from residential to commercial areas, and a reverse movement in the late afternoon. The result is an unbalanced spatial distribution of bicycles. Anticipating and meeting demand is essential to maximize the value of the system [42]. Before demand can be met, the spatio-temporal distribution of bicycles needs to be understood to some degree. The objectives of this tutorial are more pedestrian than a spatio-temporal analysis. We set out to determine the effect of a few factors on use over an entire system.

The data to be used in this tutorial were used in a Kaggle [32] competition. The reader can learn more about the Capitol bike share system and the competition at their website https://www.kaggle.com/c/bike-sharing-demand. In brief, the objective of the competition was to accurately forecast bicycle rental demand in the Washington, D.C. Capital Bikeshare program as a function of weather and other variables [18]. There are three potential response variables in the data file: the number of bicycles rented by registered users, the number of bicycles rented by casual users, and the number of bicycles rented by both types of users. The data are hourly counts, and hour of the day is likely to be important for predicting use. While hour can be thought of as quantitative variable, it will not be appropriate to treat it as such since usage will not change linearly during the course of a day. If change were linear, then the change between any 2 hours of the day would depend only on the difference between the 2 hours. The change from 7 a.m. to 10 a.m. would be the same as the change between 9 p.m. and 12 p.m. Yet in the morning, the rate of change will be positive whereas in the evening, the change will negative.

An efficient analytic approach is to treat hour of the day as a factor so that each hour will have an effect unrelated to the other hours.

1. The data file is named `bikeShare.csv`. The `csv` extension indicates that columns are comma-delimited. Store the data as an `R` data frame using the following instruction.

```
Data = read.csv('.../bikeShare.csv')
```

2. Assign names to the columns:

```
names = c('datetime', 'season', 'holiday', 'workingday',
    'weather', 'temp', 'atemp', 'humidity', 'windspeed', 'casual',
    'registered', 'sum')
colnames(Data) = names
```

3. Use the `head` and `str` functions to examine the first 25 records of the data frame and to determine the structure of the column variables.
4. The next step is to examine the relationship between usage counts and hour of day. Create a variable that identifies the hour of the day in which a particular record was collected. The hour of the day is contained in the variable `datetime`. The values of `datetime` are character strings and the hour of the day is stored in positions 12 and 13. The function `substr` (sub-string) will extract the hour. Extract the hour from each record and save the hours as a vector named `Hour`. Then produce a summary table showing how many observations were collected during each hour and a boxplot showing the distribution of the number of bicycles rented by both types of users:

```
Hour = as.integer(substr(Data$datetime,12,13))
table(Hour)
boxplot(Data$sum~Hour)
```

5. Compare the distribution of counts by hour for registered and casual users by constructing a `facet` plot using the `ggplot2` library. Install the package using the function call `install.packages('ggplot2')`. Then load the library using the call `library(ggplot2)`.
6. Build an array in which the observations from registered users are stacked on top of the observations from casual users and include an additional variable is included that identifies the type of user. The instructions to build the data frame are

```
n = dim(Data)[1]  # Determine the number of observations.
labels = c(rep('Casual',n),rep('Registered',n))
df = data.frame(as.factor(rep(Hour,2)),c(Data$casual,Data$registered)
    ,labels)
colnames(df) = c('Hour','Count','Rider')
```

The instruction `labels = c(rep('Casual',n), rep('Registered', n))` creates a vector in which the first n values are the string *Casual* and the second n are the string *Registered*. The instruction creating the `dataframe` is of the form `df = data.frame(x,y,z)`. Since three arguments are passed to the `data.frame`, the resulting data frame will consist of three columns. The contents of `Hour` are converted to an `R` factor since `ggplot2` is expecting a factor. The casual user counts and the registered user counts are concatenated as one vector by the instruction `c(Data$casual, Data$registered)`.

7. Construct the plot:

```
ggplot(data = df, aes(x = Hour, y = Count)) + geom_boxplot(fill='red')
    + facet_wrap(~ Rider)
```

The result will be a figure similar to Fig. 6.3.

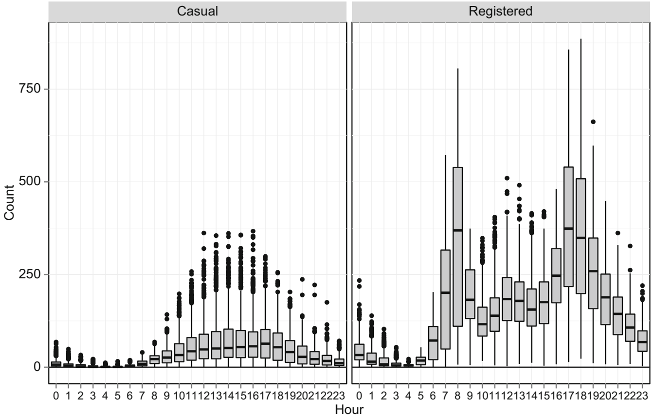


Fig. 6.3 The distribution of counts by hour for registered and casual users

Because there are substantial differences between the pattern of hourly use, it's best to at least consider separate analyses for the registered and the casual users. For simplicity, we'll limit the analysis to registered users.

8. We entertain a simple initial model based on the apparently strong association between hour of the day and counts (Fig. 6.3). The initial model supposes that hour of the day is a factor explaining variation in counts of registered users. The model is

$$\mathrm{E}(Y_i|\mathbf{x}_i) = \sum_{h=0}^{23} \beta_h x_{i,h}, \tag{6.12}$$

where

$$x_{i,h} = \begin{cases} 1, \text{ if the } i\text{th observation was collected during hour } h, \\ 0, \text{ otherwise,} \end{cases}$$

and $h \in \{0, 1, \ldots, 23\}$. The model contains $p = 24$ parameters and omits an intercept. The variables in the model consist of 24 indicator variables.

Fit the model using the `lm` function and specify that the intercept term be omitted from the model by including -1 as a model term:

```
lm.obj = lm(registered ~ -1 + as.factor(Hour),data = Data)
summary(lm.obj)
```

The `Hour` variable was created as a vector of integers with values $0, 1, \ldots,$ 23 in instruction 4. We have changed the type of the variable `Hour` as it was passed into the `lm` function. `Hour` remains a vector of integers outside of the `lm` function call.

9. The coefficient of determination computed by the `lm` function is the relative difference in the mean sums-of-squares between two models: the fitted model and a model without the explanatory variables. Usually, the model without the explanatory variables is the model $\mathrm{E}(Y) = \beta_0$ and so represents the simple null model that uses the mean of the observations, $\overline{y} = \widehat{\beta}_0$ against which to measure the information value of the explanatory variables. But the model that was just fit does not contain an intercept, and so the null model is $\mathrm{E}(Y) = 0$. This model is of little value as a baseline model against which to measure the information value of the explanatory variables. The baseline model ought to be a simple yet reasonable approximation of the target variable. The model $\mathrm{E}(Y) = 0$ is only foolish.

   Verify that `Multiple R-squared` in the output of `summary(lm())` equals
$$\frac{\widehat{\sigma}^2_{\text{null}} - \widehat{\sigma}^2_{\text{reg}}}{\widehat{\sigma}^2_{\text{null}}} = \frac{\sum y_i^2/n - \sum(y_i - \widehat{y}_i)^2/(n-p)}{\sum y_i^2/n}, \tag{6.13}$$
   where $\widehat{\sigma}^2_{\text{null}} = n^{-1}\sum y_i^2$ is the mean square residual about the model $\mathrm{E}(Y) = 0$. You may compute $\widehat{\sigma}^2_{\text{reg}} = \sum(y_i - \widehat{y}_i)^2/(n-p)$ by squaring the residual mean error or by computing `var(lm.obj$resid)`.

10. Compute the proportional reduction in estimated mean square error comparing the fitted model using hour as an explanatory variable to the null model $\mathrm{E}(Y) = \beta_0$. Since the null model estimate of β_0 is $\overline{y}$, the mean square error for this model is the sample variance $\widehat{\sigma}^2$. The adjusted coefficient of determination is the proportional reduction in mean square error:
$$R^2_{\text{adjusted}} = \frac{\widehat{\sigma}^2 - \widehat{\sigma}^2_{\text{reg}}}{\widehat{\sigma}^2}. \tag{6.14}$$
    Notice that the value labeled as the adjusted coefficient of determination by `R` (.771) is substantially larger than the correct adjusted coefficient of determination (.529).

11. With some effort, it can be shown that, in the conventional linear model with an intercept, the coefficient of determination R^2 is the squared sam-

ple correlation between the observed values and the fitted values. Equation (3.20) implies that

$$R^2_{\text{adjusted}} = \frac{\left[\sum(y_i - \overline{y})(\widehat{y}_i - \overline{y})\right]^2/(n-p)}{\widehat{\sigma}^2\, \widehat{\sigma}^2_{\text{reg}}}. \tag{6.15}$$

The numerator of Eq. (6.15) is the covariance between the residuals about the fitted model and the residuals about the model $\mathrm{E}(Y) = \mu$. Equation (6.15) has been derived using the fact that the sample mean of the fitted values equals the sample mean of the observed values when the fitted values are computed using linear regression.[19] Formula (6.15) is not necessary though, since R^2_{adjusted} can be computed according to `cor(lm.obj$fitted,Data$registered)^2`. The relationship between the squared correlation coefficient and the proportional reduction in variance expressed in Eq. (6.14) is very useful because it can be applied for many situations involving models besides those fit by linear regression as a measure of the information value of a set of explanatory variables. An adjusted or pseudo-coefficient of determination can be computed for prediction functions that will produce fitted values or predictions of the observed response variable.

Verify that `cor(lm.obj$fitted,Data$registered)^2` computes the adjusted coefficient of determination.

12. The variable `workingday` identifies those days in the data set that are considered to be conventional working days—days besides holidays and weekends. Include an intercept and add `workingday` to model (6.12). Use the number of registered users as the response variable:

```
r.obj = lm(registered ~ as.factor(Hour) + workingday,data = Data)
summary(r.obj)
confint(r.obj)
```

The `confint(r.obj)` instruction constructs a 95% confidence interval for each of the parameters in the model. Fit the same model to the number of casual users. Notice that the proportion of variation explained by the casual users model is smaller than that for the registered users.

13. Obtain the parameter estimates associated with the `workingday` variable for both models (registered and casual users) and verify that the parameter estimates in Table 6.8 are correct.

[19] If the fitted values are computed using a different prediction function (e.g., k-nearest neighbors regression), then the sample mean may not equal the sample of the fitted values. In that situation, we advocate computing the measure of fit as the squared correlation between fitted and observed values for simplicity.

Table 6.8 Summary statistics from the models of user counts as a function of hour of the day and the working day indicator variable. A 95% confidence interval for the working day parameter is shown for both models

			95% confidence interval	
Model	$\widehat{\sigma}_\varepsilon$	R^2_{adjusted}	Lower bound	Upper bound
Registered users	102.2	.54	34.95	42.19
Casual users	38.16	.42	−35.89	−32.82

6.7.1 An Incongruous Result

A comparison of the coefficients of determination shown in Table 6.8 reveals that the registered users model explains relatively more of the variation in the response variable (number of bicycles checked out by users) than the casual users model. Yet the confidence interval for the working day parameter is much wider for the registered users than the confidence interval obtained from the casual users. The difference lies in the residual standard deviations, 102.2 users versus 38.16 users.[20] The registered users model is actually *less* accurate with respect to parameter estimation (and for prediction) despite the larger adjusted coefficient of determination. This incongruity is present because of much greater variation in the counts of registered users than in the counts of casual users. The registered users model has reduced proportionally more variation in the response variable than the casual users model, but there remains much more unexplained variation. The reader should beware of relying on a single statistic, in this case, the adjusted coefficient of determination, to judge model fit.

6.8 Analysis of Residuals

The purpose of residual analysis is to investigate model adequacy. In some analyses involving small data sets, the analysis of residuals also may seek to investigate the origins of specific unusual observations. This discussion focuses on the different, though related, question of whether the conditions discussed at the beginning of the chapter are appropriate. First and foremost, the analysis investigates the question of whether the model is an adequate approximation of the true relationship between the mean of the response variable and the predictor variables.

When hypothesis testing is conducted, then the constant variance and independence conditions should be investigated, and if the sample size is small, the normality condition also should be examined. The investigation is aimed at determining whether the residuals are realizations of independent and normally distributed random variables with mean zero and constant variance.

[20] The models were fit using the same number of observations: $n = 10{,}886$.

The constant variance condition was described earlier in the statement that all residuals have a common variance σ_ε^2. The normality condition is relatively easy to confirm or refute in most applications. However, the normality condition recedes in importance when n is much larger than p because the Central Limit Theorem implies that the estimator $\widehat{\boldsymbol{\beta}}$ will be nearly normal in distribution when n is relatively large, say, $n > 100p$. Thorough investigations of the constant variance and independence conditions are often difficult when p is large.

Examining individual residuals, specifically, outliers, is fruitful when n is small. When the sample size is small, an analyst may be able to glean information about the population or process by identifying a few individual residuals $r_j = y_j - \mathbf{x}_j^T\widehat{\boldsymbol{\beta}}, j = 1, \ldots, r$ that are large in magnitude and examining the origin of the data pairs $(y_j, \mathbf{x}_j), j = 1, \ldots, r$. It may be that these data pairs possess characteristics related to Y that had not been previously recognized. For example, in an analysis of red blood cell counts using the Australian athletes data set, there may be unusual residuals attributable to the consumption of performance enhancing drugs. In principle, the analyst may be able to identify the athlete by name. Furthermore, individual data pairs may have disproportionate influence on the calculation of $\widehat{\boldsymbol{\beta}}$ and hence, on the fitted model when sample sizes are small. When n is large, undertaking an examination of influence is generally pointless because the influence of one or a few data pairs among many pairs will be negligible. We omit a discussion of influence and focus on residuals originating from large data sets. James et al. [29] and Ramsey and Schafer [48] provide accessible discussions of influence.

As a contrast to the Australian athletes data, consider the bike share data. Identifying the individual residual associated with a particular day and hour as unusual has little practical value. Residual analysis is important, though. We will see soon that a model of registered users containing only hour of the day as a predictor variable produces a preponderance of negative-valued residuals on weekend days. This observation about the residuals suggests that registered users are mostly commuting between home and work when they borrow bikes. Introducing a variable that accounts for weekday and weekend differences may improve the model. In summary, the data analytic orientation is on finding systematic model deficiencies in an effort to improve the model rather than discovering and investigating individual outliers.

6.8.1 Linearity

The linear model $\mathrm{E}(Y|\mathbf{x}) = \mathbf{x}^T\boldsymbol{\beta}$ should be free of local bias. Local bias is a condition in which the model over- or under-estimates the response variable in some sub-region of $\mathbb{R}^p$ encompassed by the predictor vectors $\mathbf{x}_1, \ldots, \mathbf{x}_n$. The objective is to insure that the model does not locally over- or under-predict

$\mathrm{E}(Y|\mathbf{x})$. We use the residuals to identify regions in which the model tends to over- or under-predict the response variable. The sum of the residuals is always zero, so on average, the model will never over- or under-predict $\mathrm{E}(Y|\mathbf{x})$. But if we examine neighborhoods of the space spanned by $\mathbf{x}_1, \ldots, \mathbf{x}_n$, over- or under-prediction is possible. Figure 6.4, from the bike share problem, illustrates local bias since the model consistently under-predicts registered counts when the fitted values are small. The next question to be answered is when are the predictions small. We defer the question now, though.

The investigation of linearity is based on the following maxim. In the absence of local bias, the residuals

$$r_i = y_i - \widehat{y}_i = y_i - \mathbf{x}_i^T \widehat{\boldsymbol{\beta}},$$

$i = 1, \ldots, n$, will not show trend or pattern when plotted against any variable (the variables need not be predictor variables). If there is trend or a pattern exhibited by the residuals, then local bias is present. Linearity is investigated by plotting the residuals against actual and potential predictor variables.

Local bias should be removed by revising the model, perhaps by adding another predictor variable or transforming the response variable and refitting the model. Common transformations are logarithmic, logit, and power functions (e.g., $y^{1/2} = \sqrt{y}$.) Let's continue with the bike share data to gain some insight into linearity and residual analysis.

6.8.2 Example: The Bike Share Problem

Residual analysis consists primarily of the application of graphical techniques that reveal facets of the residual distribution. From this point on, residual analysis is illustrated through a progression of visualizations. We use the bike share data and begin with a relatively simple model of registered counts as a linear function of three factors: hour of the day, holiday, and working day. The holiday factor identifies days as holidays or not, and the working day factor identifies the days Monday through Friday.

The starting point is a plot of the residuals against the fitted values (Fig. 6.4). There are several interesting features. First, the residuals are stacked in vertical bands since there are at most $24 \times 2 \times 2 = 96$ unique fitted values because the hour of the day variable has 24 levels and the working day and holiday variables have two levels each. Secondly, it's difficult to see individual residuals because the number of unique fitted values is small relative to the number of residuals (10,886). Finally, the distribution of the residuals expands in width as the fitted values become larger. This pattern is a classic example of nonconstant variance. The distribution of the residuals is not constant but instead depends on the magnitude of the fitted values. Thus, one of the conditions necessary for accurate hypothesis testing is not met by the model residuals from which the tests are computed. The p-values associated with the tests of significance may not be accurate. However, each test of significance unequivocally argued for the inclusion of the associated variable in the model. We have no doubts about the importance of the variables—we recognize that bicycle use is related to commuting and the number of commuters varies with hour and day of the week. The final observation is that there appears to be local bias associated with the fitted model since all of the residuals associated with the smallest five fitted values are positive. This bias is of minor importance in light of the magnitude of the variability of the residuals when the fitted values are larger than ten users per hour.

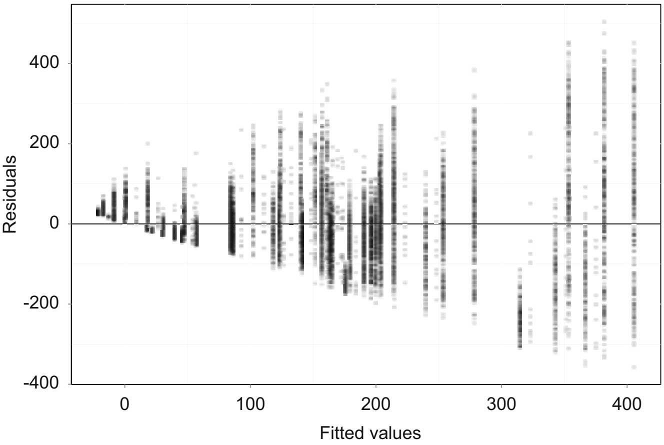


Fig. 6.4 Residuals plotted against the fitted values obtained from the regression of registered counts against hour of the day, holiday, and workingday; n = 10,886, and $p = 23+1+1 = 25$

It's instructive to take a closer look at bias associated with the smallest fitted values. Let's consider the model with only hour of the day and holiday as factors. Specifically, consider

$$\mathrm{E}(Y_i|\mathbf{x}_i) = \beta_0 + \beta_1 x_{i,\text{holiday}} + \textstyle\sum_{j=1}^{23} \beta_{j+1} x_{i,\text{hour } j}, \tag{6.16}$$

where each of the variables in the model is an indicator of a particular level. For instance,

$$x_{i,\text{holiday}} = \begin{cases} 1 & \text{if day } i \text{ is a holiday,} \\ 0 & \text{if day } i \text{ is a not a holiday.} \end{cases}$$

The model can be displayed for every possible combination of hour of day and holiday (there's $48 = 24 \times 2$ combinations). The model is shown explicitly for a handful of combinations in Table 6.9. The specific form of the model is shown for those selected combinations. Note that β_0 is a baseline expected count of registered users because it appears in every cell of the table. All other expected counts are obtained by making an adjustment to this baseline level. The parameter β_1 is the difference between expected counts on holidays versus days that are not holidays *averaged* over hours, since β_1 appears in every row. The parameter β_2 is the average difference in expected counts between hour 0 (12 p.m. to 1 a.m.) and hour 1 (1 a.m. to 2 a.m.) averaged over every day (holiday and not holidays). Since all of the parameters are used to describe multiple combinations of levels, no estimated expected count is determined independently of the counts observed at other levels. The effect is to impose constraints on the parameter estimates as they are fit by least squares. Specifically, the parameter estimates are determined so that sum of the squared residuals is minimized. If a parameter estimate were manipulated to produce residuals distributed about zero for an early morning hour, there will be consequences—the estimates at other levels will not fit as well as possible and the sum of the squared residuals will not be minimized.

We deduce that reducing the magnitude of the residuals associated with larger fitted values improves the sum of squared residuals at the expense of introducing bias into the small fitted values. From the practical standpoint of predicting counts, this local bias is entirely acceptable. The local bias is dwarfed by the magnitude of the error associated with larger fitted values, and so it matters little.

Table 6.9 Model 6.16 for specific combinations of hour of day and holiday. The estimates are $\widehat{\beta}_0 = 9.93$, $\widehat{\beta}_1 = 13.93$, $\widehat{\beta}_2 = -3.80$, and $\widehat{\beta}_3 = -5.50$

Hour	Not holiday	Holiday
0	β_0	$\beta_0 + \beta_1$
1	$\beta_0 + \beta_2$	$\beta_0 + \beta_1 + \beta_2$
2	$\beta_0 + \beta_3$	$\beta_0 + \beta_1 + \beta_3$
$\vdots$	$\vdots$	$\vdots$

6.8.3 Independence

The question of whether the residuals are independent is difficult to exhaustively investigate as there is no single test or plot that immediately provides information about independence or lack of thereof. The analyst must anticipate a reason for lack of independence. Lack of independence is manifested in two ways. First, the value of one residual, say r_i may provide information about another residual. For instance, if knowing the value of r_i provides some information about the value of some other residual, say r_j, then the residuals are not independent. Abstractly, the problem in detecting dependency depends on anticipating a reason for dependency, and thus, determining which pairs (r_i, r_j) to investigate. If the same observational unit (for example, an individual) yields multiple observations, then observations are not independent based on the presumption that observations from the same observational unit will be more alike than observations obtained from different observational units. Detecting lack-of-independence without knowledge of the data generation mechanism may be very difficult.

Two common sources of lack-of-independence that can be detected by analysis are serial and spatial correlation. If the residuals are serially correlated, then residuals that are observed near in time will be more similar than residuals that are well-separated in time. It's unusual, but serially correlated residuals may also be more *dissimilar* than residuals that are well-separated in time. Residuals will be spatially correlated if observations that are spatially close together are more similar, or more dissimilar, than those that are well-separated. Spatial correlation cannot be investigated for the bike share problem as the counts are numbers of bikes in use per hour across the entire region. If a location were attached to the counts, then spatial correlation likely would be manifested by the residuals.

Serial correlation can be investigated for these data because a time stamp is attached to each count. Since the residuals may be organized by time, we may compute the lag correlation coefficients expressing the similarity of residuals that are separated by 1 h, by 2 h, and so on. The lag-r correlation coefficient, ρ_r, measures the correlation between observations that are separated by r time steps (hours in this example). The estimator of the lag-r correlation coefficient is sample correlation coefficient computed from the data pairs $(y_t, y_{t-r}), t = r + 1, \ldots, n$, where t records the number of time steps elapsed since the time of the first observation. If there is no serial correlation, then $\widehat{\rho}_1, \ldots, \widehat{\rho}_k$ for $k > 1$, should vary randomly about 0 and be small in magnitude. Figure 6.5 shows the first $k = 30$ lag-correlation coefficients computed from the residuals about the model using hour of the day, holiday, and working day as factors. This figure was generated from the `R` function `acf` and pointlessly includes the lag-zero autocorrelation coefficient (which is, of course always 1). Notice that $\widehat{\rho}_{10}$, $\widehat{\rho}_{11}$, and $\widehat{\rho}_{12}$ are relatively large, probably because the number of morning and evening commuters is positively correlated. The large values for $\widehat{\rho}_{22}, \ldots, \widehat{\rho}_{26}$ imply that use on 1 day is moderately correlated with use on the next day at the same hour of the day.

Figure 6.5 also obliterates any claim that the observations may be viewed as approximately independent. The p-values associated with the hypothesis tests previously conducted are of unknown accuracy because of substantial autocorrelation in the residuals. While this is unfortunate for hypothesis testing, it does have positive implications for prediction. Specifically, we conclude that the count from day d at hour h contains information useful for predicting the count on day $d + 1$ at hour h.

We presume that the popularity of a bike share system will vary over weeks and months as people become familiar with the operation and location of bike

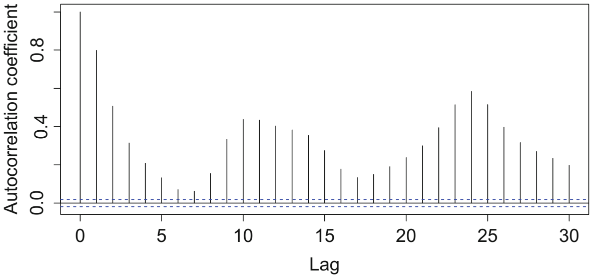


Fig. 6.5 Sample autocorrelation coefficients $\widehat{\rho}_r, r = 0, 1, \ldots, 30$, plotted against lag (r). The data are the residuals from the regression of registered counts against hour, holiday, and working day

stands, or perhaps as the bicycles fall into disrepair. If this presumption is true, then there will be trend over time in the residuals not only on a 24-h cycle but also according to some non-cyclical long-term pattern. Trend is a very important attribute of processes occurring over time, and fortunately, easy to observe. Figure 6.6 plots the residuals from the model against days since January 1, 2011, the earliest day in the data set. A flexible *smooth* summarizing the general trend has been graphed along with the residuals. The regularly occurring gaps in the sequence of days were created as part of the Kaggle competition. These days were intentionally held out. The contest participants were to predict the missing counts.[21] Figure 6.6 reveals trend with time that is neither linear nor cyclical. From the standpoint of prediction, it would be beneficial to model and incorporate the trend into forecasts of future counts. At this point in the analysis of the bike share data, one might entertain the notion of examining the normality condition. However, there's nothing to be gained since the only point in examining the normality condition is related to the appropriateness of hypothesis testing and confidence interval building, neither or which are justifiable given the presence of serial correlation (from two sources!) in the residuals.

Let's proceed with an investigation of normality anyway. A quantile-quantile plot is a convenient device for visualizing the conformity of the sample distribution of the residuals with the normal distribution. The basis of the plot is that sampling n observations from a standard normal distribution is expected to yield a distribution with the smallest observation approximately equal to the $1/n$th quantile, the second-smallest approximately equal to the $2/n$th quantile, and so on. This provides a visual check: the sample quantiles should be in a one-to-one correspondence with the expected quantiles. The kth standard normal quantile is the value $x_{(k)}$ satisfying $k/n = \Pr\left(Z < x_{(k)}\right)$ where Z has a standard normal distribution. A plot of the sample quantiles

[21] The administrators of the contest used the held-data as a test set with which to objectively evaluate the predictions made by the contestants. We think that the hold-out period (10 days) is too long. A better test of predictive accuracy would use shorter periods (3 days or less) randomly interspersed in the time series.

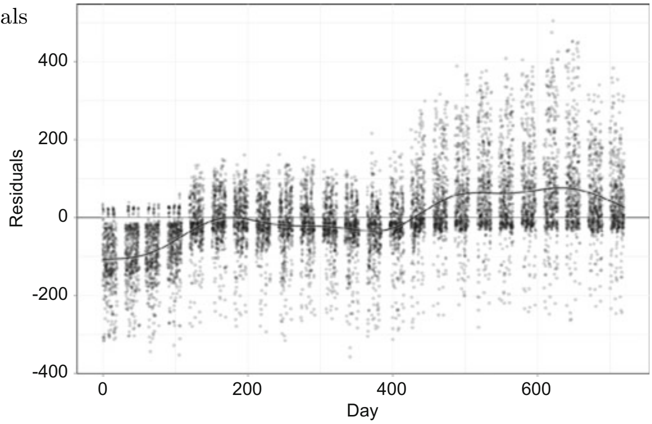


Fig. 6.6 Residuals from the regression of registered counts against hour of the day, holiday, and working day plotted against day since January 1, 2011. A smooth is plotted to summarize trend

against the expected quantile should yield a set points describing a straight line. The degree of conformity to a straight line unfortunately depends on n. The R function call `qqnorm(scale(y))` will produce a quantile-quantile plot from a vector of observations **y**. Figure 6.7 is a quantile-quantile plot constructed from the residuals of the regression of registered user counts against hour of the day, holiday, and working day. The horizontal axis label generated by R is *Theoretical Quantiles.* (We've been using the term expected quantiles instead of theoretical.) Figure 6.7 provides convincing visual evidence that the distribution of the sample residuals is not normal. We observe that the sample quantiles are consistent larger than expected when the standardized residuals are larger 1.5. Residuals less than the median conform to the shape of the normal distribution. This observation and the symmetry of normal distribution implies that distribution of model residuals is right-skewed. The right tail of a right-skewed distribution is longer than the left tail.

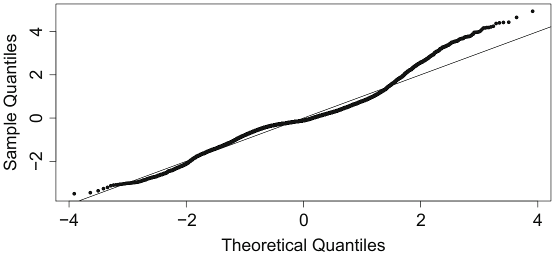


Fig. 6.7 A quantile-quantile plot comparing the distribution of the residuals to the standard normal distribution. Also shown is a diagonal line with slope 1 and intercept 0. If the sample distribution were normal, then the points will fall on or near the diagonal

6.9 Tutorial: Residual Analysis

We continue with the bike share data and use the count of casual users as the response variable.

1. Repeat the first two instructions of the previous tutorial thereby storing the data file `bikeShare.csv` as a data frame named `Data`.
2. Fit the model of expected count of casual users as a function of the factors hour of the day (`Hour`) and `workingday`. Name the fitted model `c.obj`. Verify that the adjusted coefficient of determination is $R^2_{\text{adjusted}} = .416$. Include an intercept in the model.
3. Extract the residuals and fitted values associated with `c.obj` and save as a data frame. Include the hour variable in the data frame:

```
Hour = as.integer(substr(Data$datetime,12,13))
df = data.frame(c.obj$fitted,c.obj$resid,Hour)
colnames(df) = c(’Fitted’,’Residuals’,’Hour’)
```

4. Plot the residuals against hour using the `ggplot2` library functions:

```
library(ggplot2)
plt = ggplot(df, aes(x = Fitted,y = Residuals)) + geom_point(alpha = .2,
    pch = 16)
plt = plt + xlab("Fitted values") + ylab("Residuals")
print(plt)
```

   The argument `alpha = .2` passed to `geom_point()` produces plotting symbols that are semiopaque, a very useful feature when plotting large volumes of data.

   Once again, we see that the variance of the residuals increases with the fitted values and that the model under-predicts the counts when the counts are small. This statement is based on the observation that the residuals associated with the smallest fitted values are all positive. A positive residual, say $r_i = y_i - \widehat{y}_i$, implies that $\widehat{y}_i < y_i$.
5. Allow for interaction[22] between hour of the day and working day by fitting the model

$$\begin{aligned} E(Y_i|\mathbf{x}_i) = \beta_0 + \textstyle\sum_{j=1}^{23} \beta_j x_{i,\text{hour } j} + \beta_{24} x_{i,\text{working day}} \\ + \textstyle\sum_{j=1}^{23} \beta_{j+24} x_{i,\text{interaction } j}, \end{aligned} \tag{6.17}$$

   where

$$x_{i,\text{interaction } j} = x_{i,\text{hour } j} \times x_{i,\text{working day}}.$$

[22] Interaction was discussed in Sect. 6.6.1.

The instruction

```
c.obj = lm(casual ~ as.factor(Hour) * workingday, data = Data)
```

will fit model (6.17). The interaction model has the effect of allowing each combination of hour and working day to have an estimated expected count unconstrained by the observations obtained from other combinations of hour and working day. The consequence is that residuals for any particular combination of hour and working day will be centered on zero. In this context, centered on zero means that the sum of the residuals for any particular combination will be zero. Plot the residuals against the fitted values for this model. The function call `plot(c.obj)` will produce the residual plot and the quantile-quantile plot and two other plots that are not germane to this discussion.

6. Construct a quantile-quantile plot of the residuals:

```
qqnorm(scale(c.obj$resid),main=NULL,pch=16,cex=.8)
abline(a=0,b=1)
```

The observed quantiles are smaller than the normal-distribution quantiles at the lower and upper range of the sample distribution implying a distinct deviation from the normal distribution.

7. Construct a plot showing the lag-correlation coefficients using the residuals:

```
acf(c.obj$resid)
```

8. Plot the residuals against number of days elapsed since January 1, 2011, the first day for which there is data. First, convert the `datetime` variable from a factor to a date-time object. The function `julian` translates `datetime` to days elapsed since January 1, 1960.

```
datetime = strptime(as.character(Data$datetime),
    format="%Y-%m-%d %H:%M:%S")
elapsedDays = as.integer(julian(datetime) - 14975)
```

We subtract 14,975 from all of the julian dates so that the smallest elapsed day is 0.

9. Plot the residuals against `elapsedDays` using `ggplot2` and add a smooth:

```
df = data.frame(elapsedDays,c.obj$resid)
colnames(df) = c('Day','Residuals')
plt = ggplot(df, aes(x=Day,y=Residuals)) + geom_point(alpha=.2,pch=16)
plt = plt + xlab("Day") + ylab("Residuals") + geom_hline(yintercept=0)
plt = plt + geom_smooth()
print(plt)
```

   There is a pattern of local over- and under-estimation that appears to be a manifestation of season.

6.9.1 Final Remarks

Much may be learned about model adequacy by examining the residuals. Anyone that uses a fitted model should be knowledgeable of its failures and almost all models fail in some respect.[23] Some authors, notably Ramsey and Shafer [48], argue that residual analysis should be based on the residuals from a model containing most if not all of the available variables. In other words, their position is that *for residual analysis* (and only for residual analysis), an over-fit model is better than an under-fit model. The logic is that the presence of uninformative variables will not substantially affect the residuals since a non-informative variable does not contribute much to the fit of the model, and hence to the residuals. The exception to this rule occurs when the number of variables is nearly as large as the number of observations. In this case, the fitted values will be partially determined by uninformative variables. On the other hand, a model that is wrong by the omission of informative variables will yield residuals that will be larger in magnitude than the residuals from a properly fit model. After the analyst has examined the residuals from the (perhaps) over-fit model, she may make an informed decision about the conformity of the residuals to the conditions necessary for accurate hypothesis testing and confidence interval construction. There's no need to repeat the residual analysis after model fitting unless the fit of the final model is substantially worse that the first model.

There's a large literature on the subject of model fitting. We've largely ignored the topic as it is something of a digression from data science algorithms. James et al. [29] and Hastie et al. [28] provide modern discussions of the topic more suited for data science applications than the statistically oriented recommendations of Ramsey and Shafer [48]. Given a relatively small pool for candidate predictor variables, we find ourselves most often beginning the process of model fitting with one or a few predictors that, in our

[23] The aphorism *all models are wrong, but some are useful* is due to G.E. Box.

mind, belong in the model and examining residual plots as we try adding additional predictor variables. We tend to ignore hypothesis tests and focus on the numerical measures of fit R^2_{adjusted} and $\widehat{\sigma}_\varepsilon$, especially when data sets are large ($n > 50{,}000$) since even very small improvements in model fit will be statistically significant. We may examine p-values at any stage in the search for a good model to confirm visual evidence of associations. In do so, we do not hold fast to a particular p-value threshold for including or excluding variables. In general, our preference is for fewer predictor variables and limiting the number of interaction terms in the model. With many candidate predictor variables and little knowledge of the variables, we favor the lasso for automatic variable selection [28, 29].

6.10 Exercises

6.10.1 Conceptual

6.1. Show that formula (6.13) is correct.

6.2. The bike share tutorial of Sect. 6.7 used a model (6.12) with one factor and no intercept. Based on this specific model, `R` generates an indicator variable for each level of the factor. The *design* matrix $\mathbf{X}$ used in model fitting can be extracted using the function call `X = model.matrix(lm.obj)`.

a. What is the dimension of $\mathbf{X}$?
b. Recall that the least squares estimator of $\boldsymbol{\beta}$ is computed according to $\widehat{\boldsymbol{\beta}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$. The elements of the matrices $\mathbf{X}^T\mathbf{X}$, $(\mathbf{X}^T\mathbf{X})^{-1}$, and $\mathbf{X}^T\mathbf{y}$ are recognizable—describe the elements.
c. Describe $\widehat{\boldsymbol{\beta}}$ in terms of the counts. Why are the standard errors associated with the individual estimates $\widehat{\beta}_0, \ldots, \widehat{\beta}_{23}$ approximately equal?
d. Consider the model given in Eq. (6.12) and the test statistics for the hypotheses

$$\begin{aligned} H_0 &: \beta_i = 0 \\ H_a &: \beta_i \neq 0, \end{aligned} \tag{6.18}$$

   $i = 0, 1, \ldots, 23$. `R` summarizes these tests in the output table generated by `summary(lm())`. However, H_a is not appropriate for the situation at hand. Restate H_a so that it is consistent with the response variable being counts. Calculate a p-value for the appropriate H_a for β_3 and interpret the result of the hypothesis test.
e. Add the variable `holiday` to the model of expected registered users. Given the hour of the day, determine whether the data support the hypothesis that the expected count of registered users varies with the variable `holiday`. If there is statistical evidence, report the estimated effect and give a 95% confidence interval for the true effect. Repeat the analysis for the casual users. Comment on the accuracy of the inferential methods.

f. Test the significance of `workingday` given hour of the day and `holiday`, but this time, compute the test statistics for each of the response variables: counts of registered users, counts of casual users, and the sum of registered and casual counts (`sum`). Why do the results of the test for `sum` not agree with the results obtained from registered and casual counts?

6.3. Consider the interaction model given by Eq. (6.10). Show that if the variable x_{female} is removed from the model, then the model implies that the linear model for males and the linear model for females have different slopes but the same intercept. In most cases, removing x_{female} or x_{ssf} from the interaction produces a model that is constrained in an illogical manner.

6.10.2 Computational

6.4. Using the bike share data (Sect. 6.7), answer the following questions.

a. What is the estimated effect of one additional degree of ambient air temperature on the count for registered users? Fit a model to the number of registered users using `workingday`, hour of day, and ambient air temperature (`temp`). Report the parameter estimate associated with `temp` and a 95% confidence interval for the parameter associated with `temp`.
b. Does the effect of temperature depend on the hour of the day? Investigate this question by carrying out an extra sums-of-squares F-test that compares the lack-of-fit of a model that does not allow the effect to depend on hour to the lack-of-fit of a model that allows different temperature effects for different hours. Plot the estimated differences in temperature effect against hour and interpret.

6.5. Return to the Australian athletes data set and consider the model in which the response variable is percent body fat and the explanatory variable is skinfold thickness.

a. Fit separate models for females and males by first selecting males only. This may be accomplished using the `subset` argument. For example:

```
plot(ais$ssf,ais$pcBfat)
m.obj = lm(pcBfat~ssf,subset = (sex=='m'),data = ais)
males = which(ais$sex == 'm')
points(ais$ssf[males],ais$pcBfat[males],col='blue',pch=16)
abline(m.obj,col = 'blue')
```

Repeat for females. Show both genders and the separate regression lines on the same plot.

b. Compare the fitted models. Does it appear that the slopes are nearly equal allowing for sampling variability? The expression *sampling variability* implies that a statistical test is necessary to answer the question.
c. Compare the intercepts by retrieving confidence intervals for each fitted intercept. Do the confidence intervals overlap and what can be inferred about the true intercepts from the confidence intervals?

6.6. The researchers' interest in the Australian athletes [60] centered on the relationship between hematology, morphology, and physiological demands related to the athletes' sports.

a. Fit a model in which plasma ferritins, `ferr`, is the response variable and skinfold thickness, percent body fat, and female are explanatory variables. Ferritin is a protein that binds to iron, and iron is key for oxygen transport in the body. It's speculated that high levels of ferritin assist in oxygen transport. Construct a table showing the parameter estimates and their standard errors.
b. What is the adjusted coefficient of determination associated with the fitted model? For each of the three explanatory variables, add the t-test and associated p-value to the table of parameter estimates. Describe the strength of evidence supporting the contention that skinfold thickness is associated with plasma ferritins. Repeat for percent body fat and gender.
c. Re-fit the model after removing the variable with the largest p-value. What is the adjusted coefficient of determination associated with the new fitted model? For each of the two remaining explanatory variables, report the p-value and describe the strength of evidence supporting the contention that the variable is associated with plasma ferritins.
d. Re-fit the original three-variable model and extract the residuals. Construct a set of side-by-side boxplots showing the distribution of residuals for each sport. Note that a negative residual associated with y_i implies that the model over-predicted y_i and a positive residual implies that the model under-predicted y_i. Which athletes appear to have greater than expected levels of ferritins?

6.7. Seventy-two female anorexia patients participated in an experiment aimed at assessing the efficacy of two behavioral therapies for the treatment of anorexia [24]. Each subject participated in one of three treatments: control, family therapy, or cognitive behavioral therapy for 6 weeks. Pre- and post-experiment weights were recorded. The objective of this analysis is to determine if the treatments were effective and to estimate the mean weight gains provided that there is evidence that the weight gains are real.

It will be necessary to account for pre-experiment weight in the analysis since there is a great deal of variation due to this variable and a simple comparison of post-experiment weights will be compromised by weight variation within group.

a. The data are contained in the `R` library `MASS` and the data set is named `anorexia`. Load the `MASS` library and retrieve the data set. Examine the first 50 records. Determine the structure of the variables using the function `str`. Determine the number of observations for the three treatment groups using the instruction `table(anorexia$Treat)`.

b. Using `ggplot`, plot post-experiment weights against pre-experiment weights. Identify points originating from a particular treatment group by shape and color:

```
plt = ggplot(anorexia, aes(x = Prewt, y = Postwt))
    + geom_point(aes(shape = Treat,color = Treat), size = 3)
```

c. Enhance the plot by successively adding attributes to the plot. Add least squares regression lines for each treatment group and identify the lines by color:

```
plt = plt + geom_smooth(aes(colour = Treat), method =’lm’)
```

Print `plt` and note that `ggplot` automatically added confidence intervals for the mean response.

d. Add a line with a slope of one and intercept of zero:

```
plt = plt + geom_abline(intercept = 0,slope=1)
```

Examine the plot and notice the position of the treatment-group lines relative to the line with slope equal to one. What, if anything can be said about the efficacy of the treatments based on the relative positions of the treatment-group lines relative to the line with slope one and intercept zero?

e. Another way to view the data splits the plotting window according to treatment group. Use the `ggplot` function `facet_grid`:

```
plt = ggplot(anorexia, aes(Prewt, Postwt)) + facet_grid(.~Treat)
plt = plt + geom_smooth(aes(colour = Treat), method =’lm’)
plt = plt + geom_point(aes(colour = Treat), size = 2)
plt = plt + scale_colour_discrete(guide="none")
```

Print `plt`.

f. Examine each of the regression models shown in the plots. You'll need to fit three separate regression models by using the `subset` argument in the `lm` function call. The control group model can be fit using the instruction

```
summary(lm(Postwt~Prewt,subset = (Treat=='Cont'),data = anorexia))
```

Fit the model for the other treatments. For which of the three treatments is there evidence of an association between pre- and post-treatment weight?

g. Use the centered pre-experiment weight as an explanatory variable:

```
anorexiaPrewtMean = mean(anorexia$Prewt)
lm.obj = lm(Postwt~I(Prewt - anorexiaPrewtMean),
            subset = (Treat=='Cont'),data = anorexia)
summary(lm.obj)
```

When a variable is transformed in the `lm` function call, it usually must be wrapped in the function `I`, as was done in the expression `I(Prewt - anorexiaPrewtMean)`.

Consider the intercepts in each of the fitted models (using centered pre-experiment weight). Report the intercepts and 95% confidence intervals for each. Carefully explain the interpretation of the intercepts. Keep in mind that the intercept is the position on the vertical axis at which the fitted line intersects the vertical axis. If a centered variable is equal to zero, then the un-centered value equals the mean.

h. Does there appear to be a significant difference between control and family therapy groups based on the three intercepts? Does there appear to be a significant difference between control and cognitive behavioral therapy groups? Draw a conclusion regarding the therapies: is there statistical evidence that the therapies are effective? What is your basis for drawing your conclusion?

i. Carry out the extra-sums-of-squares F-test for the significance of treatment. The constrained model should contain pre-treatment weight and treatment; the unconstrained model should contain pre-treatment weight, treatment, and the interaction between pre-treatment weight and treatment. Can we conclude with confidence that treatment affects mean post-experiment weight? Why or why not?

Chapter 7
Healthcare Analytics

Abstract Healthcare analytics refers to data analytic methods applied in the healthcare domain. Healthcare analytics is becoming a prominent data science domain because of the societal and economic burden of disease and the opportunities to better understand the healthcare system through the analysis of data. This chapter introduces the reader to the domain through the analysis of diabetes prevalence and incidence. The data are drawn from the Centers for Disease Control and Prevention's Behavioral Risk Factor Surveillance System.

7.1 Introduction

Healthcare accounts for 17% of the gross domestic product of the United States, much more than any other country in the world [2]. The providers of healthcare are under immense pressure to improve delivery and reduce costs. Reigning in costs while maintaining quality of care is critically needed. Furthermore, the healthcare system is complex, opaque, and undergoing change, all of which makes it difficult to answer even elementary questions about the system and the people that rely on it. Healthcare analytics has developed in the pursuit of information that will assist in improving the delivery of healthcare.

Government agencies, affordable care organizations,[1] and insurers are involved in the analysis of electronic medical records with the intent of identifying best medical practices, cost-effective treatments, and interventions. For example, if an organization identifies a group of clients that are at significant

[1] An affordable care organization (ACO) is a network of physicians and hospitals that provide patient care. ACO's have a responsibility to insure quality care and limit expenditures while allowing patients some freedom in selecting specific medical services.

B. Steele et al., *Algorithms for Data Science*,
DOI 10.1007/978-3-319-45797-0_7

risk of developing a chronic illness, then the organization may provide financial incentives to the clients if they engage in activities that reduce risk. Community and population health is another area in which healthcare analytics are used. The Centers for Disease Control and Prevention and the World Health Organization, for example, are engaged in monitoring and forecasting disease incidence and prevalence with the aim of better understanding relationships between health, health-related behaviors, sanitation and water quality, and other factors.

At the beginning of the chapter, we reported a remarkable statistic: 17% of the gross domestic product of the United States is spent on healthcare. It is not entirely fair to blame institutionalized medicine for this state of affairs. Some of the responsibility falls on the individuals. Purportedly, many Americans have not engaged in responsible behaviors and as a result the incidence of a variety of chronic diseases has increased at the same time that the healthcare system has become more effective at treating disease. Intervention is key for turning the situation around, and intervention depends identifying individuals that are at risk before they become ill. Here lies an opportunity.

To gain some exposure to healthcare data and analytics, we undertake an analysis of diabetes prevalence and incidence. *Prevalence* is the proportion or percent of a population that is affected by a condition or disease. The term is generally applied to chronic diseases such as diabetes, asthma, heart disease, and so in. *Incidence* is the rate at which new cases appear among a population. For example, if the prevalence of diabetes is .12, then 12 of every 100 individuals in the population has the condition. The incidence of diabetes is the annual rate at which individuals become afflicted with the disease. If the incidence is .004 then, annually, 4 new cases are expected per 1000 previously undiagnosed individuals.

Type 2, or adult-onset diabetes is of great importance in the U.S. public health arena because it is a serious chronic disease. Prevalence is in influx and difficult to estimate in part because it may be undiagnosed in some individuals, but we estimate that 10% of the U.S. adult population has type 2 diabetes as of 2014. Furthermore, the prevalence of the disease has been increasing over the past several decades.[2] It's associated with a spectrum of complications including, but not limited to nerve and kidney damage, cardiovascular disease, retinopathy, skin conditions, and hearing impairment. Diabetes is considered to be incurable. For many individuals, however, the disease is preventable and controllable through diet and exercise.[3] Depending on age at onset, lifetime costs for an individual are estimated to be as much as $130,800 [71].

[2] The tutorials of this chapter will reveal substantial geographic differences in prevalence and incidence across the United States.

[3] It's controllable in the sense that related conditions such as retinopathy can be avoided or delayed.

7.2 The Behavioral Risk Factor Surveillance System

The U.S. Centers for Disease Control and Prevention initiated the Behavioral Risk Factor Surveillance System in 1984 for the purpose of learning about the factors that affect health and well-being of the U.S. adult population. This goal is advanced by conducting the largest annual sample survey in the world.[4] The survey asks a sample of U.S. adult residents a large number of questions regarding health and health-related behaviors. The focus in this discussion is on diabetes, and the principal question of interest asks *has a doctor, nurse, or other health professional ever told you that you have diabetes?* The possible responses are (1) *yes*, (2) *no*, (3) *no, (but) pre-diabetes or borderline diabetes*, (4) *yes, but only during pregnancy*, (5) *don't know/not sure*, (6) *refused*. The interviewer may also enter the code 7 indicating that the question was not asked. A very large fraction of the responses are informative; to illustrate, response codes 5, 6, and 7 amounted to .18% of the sample in 2014. The question asked of the respondent does not distinguish between type 1 and type 2 diabetes; however, type 2 is far more common since approximately 4.3% of diabetes cases are estimated to be type 1 [4].

The objective of the following tutorial is to estimate prevalence and incidence by state for the time period 2000–2014. For simplicity, we will identify a case of diabetes only by the first listed answer *yes* and regard all other answers as no. It's to be expected that our estimate of prevalence will be biased downward to a small degree by not eliminating the last three non-responses. We are not able to address the verisimilitude of the answers. The Centers for Disease Control and Prevention (CDC) published the respondents' county of residence until 2013. Since then, out of privacy concerns, the county identifier has been hidden from view and the finest level of spatial resolution is the state.

The Behavioral Risk Factor Surveillance System survey is conducted by telephone, originally by landline only, but since 1993, by contacting respondents via landline and cell phone. The number of sampled individuals has become remarkably large in recent years (464,664 respondents in 2014). The sample design specifies that cell phone numbers are randomly sampled and landline numbers are sampled by disproportionate stratified sampling. A consequence of the design is that the sample is neither random nor representative. Therefore, conventional estimators such as the sample mean do not yield unbiased estimates since some sub-groups of the U.S. population are over-sampled at the expense of other sub-groups. The CDC provides a set of sampling weights that reflect the likelihood of selecting a respondent belonging to a particular subgroup defined by age, gender, race and several other

[4] We've discussed and used BRFSS data in Chap. 3.

demographic variables [10, 11]. These weights provide a means by which bias may be reduced or perhaps eliminated.[5]

Let $v_k, k = 1, \ldots, n$, denote the sampling weight assigned to the kth record or observation where n denotes the number of informative responses for a particular year. It's helpful from a conceptual standpoint to scale the weights to sum to 1 by defining a second set of weights $w_k = v_k / \sum_{j=1}^n v_j$, for $k = 1, \ldots, n$. To adjust the conventional estimators for sampling bias, we recast our estimators as linear estimators.

A linear estimator is a linear combination of the observations. Thus, a linear estimator computed from the observations contained in the vector $\mathbf{y} = [y_1 \ \cdots \ y_n]^T$ may be expressed as a linear combination

$$l(\mathbf{y}) = \sum_{k=1}^{n} w_k y_k, \tag{7.1}$$

where w_k's are known coefficients with the properties $0 \leq w_i \leq 1$ for all i, and $\sum_{i=1}^n w_i = 1$. For instance, the sample mean is a linear predictor since setting $w_k = n^{-1}$ for each k yields the sample mean. Alternatively, an estimator of the mean may be computed by using the BRFSS sampling weights in formula (7.1).

7.2.1 Estimation of Prevalence

Let's set an objective of estimating the prevalence, or the proportion of the adult population that has diabetes for each of the United States, Puerto Rico and the District of Columbia. Prevalence is equivalent to the probability that an individual selected at random from the population has diabetes. Since obtaining a probability sample of clinical diagnoses from the population of each state is beyond the reach of all but the most determined researchers, we instead estimate the proportion of the population that would answer affirmatively to the diabetes question on the BRFSS surveys. The sample proportion of adults that answer affirmatively to the diabetes question on a BRFSS sample is an estimate of prevalence. Of course, there is a possibility of interviewer bias skewing the estimates. Interviewer bias is the tendency of some respondents to give answers that are affected by the presence of the interviewer, say, answers that put the respondent in a more favorable light.

The usual prevalence estimator is the sample proportion of affirmative binary responses if the sampled individuals have been drawn at random from the population. Let's consider random sampling and suppose that the responses to the question are coded as 1 for yes and 0 for no. The kth binary

[5] The sampling weights reflect the likelihood selecting a particular respondent but are not the probability of selecting the respondent.

response is denoted as y_k, and so y_k may take on two values, 0 and 1. Then, the sample proportion can be computed as $n^{-1}\sum y_k$. The weighted prevalence estimator is used if sampling is not random and sampling weights are available. It is a linear estimator (Eq. (7.1)), and we express it as

$$\widehat{\pi}(\mathbf{y}) = \sum_{k=1}^{n} w_k y_k = \frac{\sum_k v_k y_k}{\sum_k v_k},$$

where v_k is the sampling weight associated with the kth respondent.

Another example of a linear estimator is the least squares estimator of the linear regression parameter vector given in Eqs. (3.29) and (3.33):

$$\begin{aligned} \underset{p\times 1}{\widehat{\boldsymbol{\beta}}} &= (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \\ &= \left(\textstyle\sum_{k=1}^{n} \mathbf{x}_k\mathbf{x}_k^T\right)^{-1} \textstyle\sum_{k=1}^{n} \mathbf{x}_k y_k, \end{aligned} \tag{7.2}$$

where $\mathbf{x}_k$ is the vector of predictor variables associated with the kth response y_k. Weights can be incorporated in the estimator of $\boldsymbol{\beta}$ by replacing each unweighted sum by a weighted sum

$$\begin{aligned} \widehat{\boldsymbol{\beta}}_w &= \left(\textstyle\sum_{k=1}^{n} w_k\mathbf{x}_k\mathbf{x}_k^T\right)^{-1} \textstyle\sum_{k=1}^{n} w_k\mathbf{x}_k y_k \\ &= (\mathbf{X}^T\mathbf{W}\mathbf{X})^{-1}\mathbf{X}^T\mathbf{W}\mathbf{y}, \end{aligned} \tag{7.3}$$

where $\mathbf{W}$ is a diagonal matrix with the terms $w_1, \ldots, w_n$ on the diagonal [33].

We will use linear predictors to estimate prevalence for each combination of state and year, and so allowing i and j to index state and year, the estimator of prevalence for state i and year j is

$$\widehat{\pi}_{i,j} = \sum_{k=1}^{n_{ij}} w_{ijk} y_{ijk} = \frac{\sum_k v_{ijk} y_{ijk}}{\sum_k v_{ijk}}. \tag{7.4}$$

where y_{ijk} is the kth response among the n_{ij} responses for state i and year j and v_{ijk} is the BRFSS sampling weight associated with y_{ijk}.

7.2.2 Estimation of Incidence

Diabetes incidence, in the context of the problem at hand, is the average rate of change per year in prevalence for the time period 2000–2014. The definition implies that single value summarizes incidence over the time period. In essence, we are treating incidence as if it were constant over the period. This treatment is pragmatic since 15 years of data is too little to adequately estimate and summarize time-varying incidence for each of 50 states.

An estimator of incidence for the ith state is the least squares estimator of $\beta_{1,i}$ parameterizing the simple linear regression model

$$\pi_{i,j} = \beta_{0,i} + \beta_{1,i}\text{year}_j. \tag{7.5}$$

The intercept $\beta_{0,i}$ has no practical interpretation since it is the prevalence in year 0 A.D. Let's change the setup so that $\beta_{0,i}$ has a practical interpretation. To do so, let x_j denote the difference from midpoint of the time interval (2007) and year_j, and replace year_j with x_j in the model. Then, $\beta_{0,i}$ is the prevalence when $x_j = 0$ or equivalently, for $\text{year}_j = 2007$. If forecasting prevalence were the goal, we might shift year so that $\beta_{0,i}$ is the prevalence in year 2014. Shifting year provides two estimates of interest: the estimated incidence $\widehat{\beta}_{1,i}$, and an estimate of prevalence at the midpoint of the time span, $\widehat{\beta}_{0,i}$, that utilizes all 15 years of data instead of the point estimate $\widehat{\pi}_{i,2007}$.[6] Since shifting the year variable adds a more information to analysis, we proceed with the model

$$\pi_{i,j} = \beta_{0,i} + \beta_{1,i}x_j, \tag{7.6}$$

where $x_j = \text{year}_j - 2007$. For state i, the data to be used in computing $\widehat{\beta}_{0,i}$ and $\widehat{\beta}_{1,i}$ is the set of pairs $\{(-7, \widehat{\pi}_{i,0}), (-6, \widehat{\pi}_{i,1}), \ldots, (7, \widehat{\pi}_{i,14})\}$. It should be recognized that Eq. (7.5) implies a constant rate of change in diabetes prevalence. However, a linear rate of change is not sustainable over a long time interval as it would yield absurd estimates for years distant from 2007 (unless the estimated rate of change were zero). We use the model only for the convenience of obtaining simultaneous estimates of prevalence and incidence from a relatively short series of years. Our scope of inference is limited to the time interval 2000–2014.

We turn now to computational aspects of utilizing BRFSS data for estimation of prevalence and incidence of diabetes by U.S. state and year.

7.3 Tutorial: Diabetes Prevalence and Incidence

The data processing strategy is to reduce the data via a succession of maps to arrive at a dictionary of prevalence estimates organized by year and state. Each set of annual state estimates will then be used to estimate incidence as the rate of annual change in prevalence.

The BRFSS data file structure was discussed in Sect. 3.6. To summarize, the data files are constructed with fixed-width fields. A variable must be extracted from a record as a substring according to its position in the record. The fields, or positions of each variable in the record, and the protocol for

[6] If incidence is approximately constant over the interval, $\widehat{\beta}_{0,i}$ is a more precise estimator of prevalence at the midpoint of the time span.

coding the variable are described in *codebooks* maintained by the CDC. The URL is http://www.cdc.gov/brfss/annual_data/annual_data.htm. Follow the links to reach the codebook for a specific year.

The variables in the data files are coded as integer-valued labels according to a format described in the codebooks. Section 7.2 provided an example of the coding of the diabetes question. If other variables are incorporated into an analysis, then the analyst usually must map the tabulated responses to a more convenient variable for analysis.[7] Table 7.1 summarizes the field positions by year for the variables to be used in the tutorials of this chapter.

Table 7.1 BRFSS data file field positions for sampling weight, gender, income, education, age class, body mass index (BMI), and diabetes

	Sampling weight		Gender	Income		Education	Age class		BMI		Diabetes
Year	Start	End	field	Start	End	field	Start	End	Start	End	field
2000	832	841	174	155	156	153	887	888	862	864	85
2001	686	695	139	125	126	123	717	718	725	730	90
2002	822	831	149	136	137	134	920	921	933	936	100
2003	745	754	161	144	145	142	840	841	854	857	84
2004	760	769	140	124	125	122	877	878	892	895	102
2005	845	854	148	127	128	112	1204	1205	1219	1222	85
2006	845	854	135	114	115	112	1205	1206	1220	1223	85
2007	845	854	146	120	121	118	1210	1211	1225	1228	85
2008	799	808	143	117	118	115	1244	1245	1259	1262	87
2009	993	1002	146	120	121	118	1438	1439	1453	1456	87
2010	990	999	147	120	121	118	1468	1469	1483	1486	87
2011	1475	1484	151	124	125	122	1518	1519	1533	1536	101
2012	1475	1484	141	116	117	114	1629	1630	1644	1647	97
2013	1953	1962	178	152	153	150	2177	2178	2192	2195	109
2014	2007	2016	178	152	153	150	2232	2233	2247	2250	105

The CDC asks the following question regarding diabetes: *Have you ever been told by a doctor that you have diabetes?*[8] Answers to the question that are informative regarding the diabetes status of the respondent are given in Table 7.2.

Table 7.2 BRFSS codes for diabetes

Value	Value label
1	Yes
2	Yes, but female told only during pregnancy
3	No
4	No, pre-diabetes or borderline diabetes

[7] The value labels for a specific question are usually the same from year to year.

[8] This is the question asked in the year 2004 survey. The exact phrasing has changed over time.

These values must be mapped to 0 or 1, where, informally for now, 0 denotes the absence of diabetes and 1 denotes the presence of diabetes. There's some question how to handle response values of 2 and 4. One possibility is to dismiss all response values that are not either unconditionally yes or no. Another possibility is to combine values 2, 3, and 4 as a single negative response based on the logic that diabetes is a chronic, incurable disease, and a response of 2, 3, or 4 is implies that the respondent does not have chronic diabetes. In the tutorial, we map responses of 2, 3, or 4 to 0 instead of dismissing the record.

The tutorial below makes use of functions that have been developed in previous tutorials. The functions and the sections in which the functions were developed are listed in Table 7.3.

Table 7.3 Functions, where they were developed, and their purpose

Function name	Section	Purpose
`convertBMI`	3.6	Convert string to float
`stateCodeBuild`	7.3	Build a dictionary of state names

1. Navigate to http://www.cdc.gov/brfss/annual_data/annual_data.htm and retrieve the files listed in Table 7.4.[9] The individual data files are located in sub-directories labeled by the year of interest. Download the zipped text file by clicking on the link `20xx BRFSS Data (ASCII)`. Unzip the file in your directory. The BRFSS data files will occupy about 7.8 GB of disk space.

Table 7.4 Data sets for the analysis of diabetes prevalence and incidence

cdbrfs00.ASC	cdbrfs01.ASC	cdbrfs02.ASC	CDBRFS03.ASC
CDBRFS04.ASC	CDBRFS05.ASC	CDBRFS06.ASC	CDBRFS07.ASC
CDBRFS08.ASC	CDBRFS09.ASC	CDBRFS10.ASC	LLCP2011.ASC
LLCP2012.ASC	LLCP2013.ASC	LLCP2014.ASC	

2. Create a `Python` script. Enter instructions to load the following modules and functions at the top of your script:

```
import os
import sys
from collections import namedtuple
import numpy as np
```

[9] You may already have some of these from having worked on the tutorial of Chap. 3, Sect. 3.6.

3. You'll need a file that links the state names to the state FIPS[10] codes. Go to http://www2.census.gov/geo/docs/reference/codes/files/national_county.txt and save the webpage as a file named `national_county.txt` in the data directory.
4. Create a dictionary from `national_county.txt` in which each key is a state or territory FIPS code and the associated value is the name of the state or territory. Extract the substrings from each string that contains the FIPS state code and the two-letter state abbreviation. Alternatively, you may create a list by splitting the string at the commas and extracting the FIPS state code and the two-letter abbreviation as items from the list.

   In `national_county.txt`, the two-letter state or territory abbreviation occupies positions 0 and 1 of the string and the FIPS state code occupies fields 3 and 4.

```
path = r'../national_county.txt'
stateCodes = {}
with open(path, encoding = "utf-8") as f:
    for record in f:
        stateCode = int(record[3:5])
        stateCodes[stateCode] = record[0:2]
```

5. Construct a function containing the code segment described in instruction 4. The function builds and returns `stateCodesDict`. It should appear as so:

```
def stateCodeBuild():
    path = r'../national_county.txt'
    ...
    return stateCodeDict
```

6. Test the function using the instructions

```
stateCodeDict = stateCodeBuild()
print(stateCodeDict)
```

   Remove the function from the `Python` script and place it in the `functions` module (`functions.py`).[11] Compile `functions.py` by executing the script.

[10] Federal Information Processing Standards

[11] Chapter 3 Sect. 3.6 discusses the creation of the `functions.py` module.

7. In Sect. 3.6 of Chap. 3, instruction 4, the reader was guided through the construction of the `fieldDict`, a dictionary of dictionaries that stores the field positions of the BRFSS variables by year. We will use `fieldDict` to get the field position of diabetes and sampling weight. If you haven't created the function `fieldDictBuild` and placed it in your `functions` module, then do so by following the instructions given in Chap. 3. In any case, enter field locations for the diabetes variable shown in Table 7.1 for each the year 2000 through 2014. After editing `fieldDictBuild`, execute `functions.py` so that the new entries are available when `fieldDictBuild` is called.
8. Create the dictionary `fieldDict` by calling `fieldDictBuild()`.
9. Turning now to the BRFSS data files, create a `for` loop that will read every file in your data directory by iterating over the files in the directory. We've done this before; see Chap. 3, Sect. 3.6 or Sect. 3.8. All of the BRFSS file names have the two-digit abbreviation of the sampling year in the zero-indexed positions 6 and 7 of the file name (see Table 7.4), and so we try to extract the first two digits of the sampling year from the file name using the instruction `shortYear = int(filename[6:8])`. Converting the string representation of year to an integer representation provides a test that the file is a BRFSS data file. The variable `n` counts the number of informative records and will provide the keys for the data dictionary. Program the code segment below.

```
n = 0
dataDict = {}
path = r'../Data/'
fileList = os.listdir(path)
for filename in fileList:
    print(filename)
    try:
        shortYear = int(filename[6:8])
        year = 2000 + shortYear
        print(year, filename, 'Success')
    except ValueError:
        print(year, filename, 'Failure')
```

10. Add a code segment in the `try` branch of the exception handler to extract the field dictionary for `shortYear`. Get the field positions for the variables sampling weight, body mass index, and diabetes:

```
fields = fieldDict[shortYear]
sWt, eWt = fields['weight']
sBMI, eBMI = fields['bmi']
fDia = fields['diabetes']
```

Indent the code segment so that it executes immediately after the statement `print(year, filename, 'Success')`

11. Create a `namedTuple` type with the name `data` before the `for` loop iterates over `fileList`. The `namedTuple` will be used to store the data extracted from a BRFSS record. We'll store the year, the state code, the sampling weight, and the diabetes variable:

```
data = namedtuple('dataTuple','year stateCode weight diabetes')
```

This instruction creates a new tuple subclass with the name `dataTuple`. We have named the instance of the subclass `data`. To get the value of diabetes, say, stored in an instance of `data`, we may use the syntax `data.diabetes` and do not need to keep track of the position of the variable in the tuple. This is an advantage since more variables will be consumed and stored in `data` in the following tutorial.

12. We are now ready to process the data file using a `for` loop that reads one record at a time. As the records are processed, extract the integer-valued state code and translate the integer to the two-letter representation. Determine the two-letter abbreviation for the state name by looking it up in the `stateCodesDict` dictionary.

```
file = path + filename
with open(file, encoding="utf-8") as f:
    for record in f:
        stateCode = int(record[:2])
        stateName = stateCodesDict[stateCode]
        weight = float(record[sWt-1:eWt])
```

This code segment must be indented so that it executes every time `shortYear` is successfully extracted as an integer from `filename`.

13. Build a function to process the diabetes string. It's probably most efficient to code the function in place so that the variables can be examined in the console if there is an error. When the code performs as expected, move the code to a function. The function, call it `getDiabetes`, takes `diabetesString` as an argument and returns an integer value. Name the variable `diabetes`. As was discussed above, we'll set `diabetes` $=$ 1 if the response to the diabetes question was unequivocally *yes* and set `diabetes` $=$ 0 for the other informative answers. Non-informative responses are identified by setting `diabetes`$= -1$. The function should look like this:

```
def getDiabetes(diabetesString):
    if diabetesString != ’ ’:
        diabetes = int(diabetesString)
        if diabetes in {2,3,4}:
            diabetes = 0
        if diabetes in {7,9}:
            diabetes = -1
    else:
        diabetes = -1
    return diabetes
```

Put the function in `functions.py` and recompile `functions.py`.

14. Extract diabetes from the record and convert it to an integer code by calling `getDiabetes`. Store the data extracted from `record` in `dataDict` as a named tuple:

```
diabetesString = record[fDia-1]

diabetes = functions.getDiabetes(diabetesString)
if diabetes  != -1:
    dataDict[n] = data(year, stateCode, weight, diabetes)
    n += 1
```

Note that the dictionary keys are integers from 0 to n and that n is the number of informative records. Print `n` after processing each file. You should end up with $n = 5{,}584{,}593$ entries in `dataDict`.

15. The next step is to reduce `dataDict` to a much smaller data dictionary by mapping the data stored in `dataDict` to a dictionary that uses pairs consisting of state and year as keys. The values in the dictionary are the components needed to compute the weighted estimator of prevalence for each state and year.

Equation (7.4) shows the prevalence estimator. We'll compute the numerator and denominators for state i and year j while iterating over `dataDict`. The terms to compute are

$$\sum_k v_{ijk} y_{ijk} \text{ and } \sum_k v_{ijk},$$

where y_{ijk} and v_{ijk} are values of the diabetes variable and the sampling weight, respectively, and k indexes observations originating from year j and state i.

For this next mapping, create a dictionary `StateDataDict` to store the sums for each state and year as a list $[y_{ijk}, v_{ijk}]$. The dictionary keys are tuples consisting of state code and year:

```
StateDataDict = {}
for key in dataDict:
    item = dataDict[key]
    stateKey = (item.stateCode, item.year)
    value = StateDataDict.get(stateKey)

    if value is None:
        v = item.weight
        vy = item.weight*item.diabetes
    else:
        vy, y = value
        v += item.weight
        vy += item.weight * item.diabetes
    StateDataDict[stateKey] = [vy, v]
```

Since we've completed building `dataDict`, this code segment executes after the `for` loop running over `fileList` has completed. It should not be indented.

16. Another reduction step is necessary before incidence is estimated by a simple linear regression of prevalence on year. As discussed in Sect. 7.2.2, a separate regression will be carried out for each state. Prevalence will be regressed on year and the slope coefficient will be the estimate of incidence for the state. So, we need to store pairs for each state consisting of prevalence and year.

    The dictionary that stores the regression data, `regressDataDict`, uses states as the keys. The values are lists consisting of pairs where the elements are year and estimated prevalence. Here we use a function from the `collections` module that creates a dictionary entry if none exists and appends to the existing dictionary entry if an entry exists.

```
import collections
regressDataDict = collections.defaultdict(list)

for key in StateDataDict.keys():
    state, year = key
    vy, v = StateDataDict[key]
    regressDataDict[state].append((year,vy/v))
```

Upon completion, the lists associated with a particular state code will look like this:

$$[(2001, 0.042), (2010, 0.0660), ..., (2012, 0.0727)]$$

The observation pairs are not in chronological order, but they need not be ordered for fitting a linear regression equation.

17. Compute the regression coefficients for the model given in Eq. (7.6) for each state. Since the model is to be fit separately for each state, iterate over the dictionary `regressDataDict` and collect the data for a particular state on each iteration. As illustrated by the above example, the data for state i consists of a list of m pairs $[(\text{year}_1, \widehat{\pi}_{i,1}), \ldots, (\text{year}_m, \widehat{\pi}_{i,m})]$ in which the elements are year and estimated prevalence. Using a state's data, create the $m \times 2$ matrix[12] $\mathbf{X}$ by mapping each pair to a list containing the constant 1 and the centered year, for instance `[1,year-2007]`. As the lists are created, stack them using the list comprehension instruction `[[1,year-2007] for year,_ in data]` where `data` is the list stored in `regressDataDict` with the value `stateCode`. The last step translates the list to a `Numpy` matrix. Create a vector of prevalences $\mathbf{y}$ at the same time. The code segment is

```
for stateCode in regressDataDict:
    data = regressDataDict[stateCode]
    X = np.array([[1,year-2007] for year,_ in data])
    y = np.array([prevalence for _,prevalence in data])
```

The underscore character is used as a placeholder when one element of the pair is not used.

18. Compute and print the regression coefficients for each state as the `for` loop iterates over `regressDataDict`.

```
b = np.linalg.solve(X.T.dot(X),X.T.dot(y))
print(stateCodesDict[stateCode], b[0], b[1])
```

Note that $\widehat{\beta}_{0,i} =$ `b[0]` is the estimated prevalence when centered year is 0, or equivalently, for year 2007, for the ith state. Also, $\widehat{\beta}_{1,i} =$ `b[1]` is the estimated annual rate of change of prevalence, i.e., the estimated incidence.

19. Write the regression coefficients to a file as the estimates are computed. We'll do this by opening a file for writing *before* the `for` loop executes and nesting the `for` loop below the `open` instruction. The structure is

```
path = r'.../plottingData.csv'
with open(path, 'w') as f:
    for stateCode in regressDataDict:
        data = regressDataDict[stateCode]
```

[12] The number pairs m will be 15 except for Louisiana and some U.S. territories.

20. Compute and write $\widehat{\boldsymbol{\beta}}_i$ to the output file. Align the indentation of the code segment with the statement `b = np.linalg.solve(X.T*X, X.T*y)`:

```
string = ','.join([stateCodesDict[stateCode], str(b[0]), str(b[1])])
f.write(string + '\n')
print(string)
```

    Prevalence estimates (`b[0]`) should be between .05 and .25 and an incidence estimates (`b[1]`) should be between 0 and .005 per year.

21. To visualize the results, plot estimated incidence against estimated prevalence for each state. You can use the following `R` code:

```
data = read.csv('../plottingData.csv', header = FALSE)
colnames(data) = c('State', 'Prevalence', 'Incidence')
plot(c(.05,.14), c(0,.005), type = 'n',
    xlab = '2007 Estimated prevalence', ylab = 'Estimated incidence')
text(x = data$Prevalence, y = data$Incidence, labels = data$State,
    cex = .8)
```

    Your figure should resemble Fig. 7.1. The figure reveals that prevalence and incidence are moderately correlated ($r = .540$). Further, there is evidence of spatial clustering; for example, the largest estimates aside from Puerto Rico originate from southeastern and Ohio Valley states.

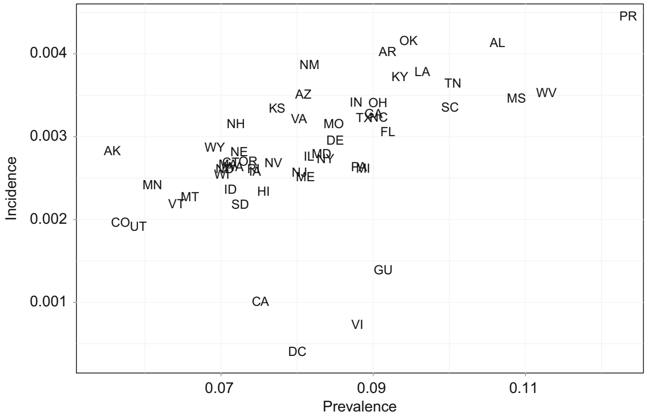


Fig. 7.1 Estimated diabetes incidence plotted against estimated prevalence by state and U.S. territory. Estimated prevalence and incidence pairs are identified by the two-letter abbreviation. $n = 5{,}592{,}946$

7.4 Predicting At-Risk Individuals

There are substantial benefits to identifying individuals whose risk of developing type 2 diabetes is greater than normal. Once identified, these individuals may be proactively treated to prevent the disease or delay the onset and severity should it occur. Significant benefits would be realized if a medical practitioner, affordable care organization, or self-insured organization, could estimate the risk of an individual developing type 2 diabetes using nothing more than self-reported demographic variables. Individuals that were identified as having elevated risk would engage in preventative activities such as participating in a program that promotes healthy diet and exercise.

In the following tutorial, an estimator of the probability that an individual has diabetes is constructed from BRFSS data. The estimator is a function that consumes a predictor vector $\mathbf{x}_0$ consisting of demographic variables measured on the individual—a demographic profile—and returns the probability estimate. The probability estimate is the estimated proportion of the U.S. adult resident population with the same demographic profile and having received a diabetes diagnosis from a doctor.[13] For example, suppose that an individual has the following demographic profile: the individual is between 30 and 34 years of age, has a college degree, an annual household income between \$25,000 and \$35,000, and a body mass index between 30 and 32. If n_0 denotes the number of individuals in the data set that have the same demographic characteristics, and y_0 denotes the number of these individuals that reported a diabetes diagnosis, then, the estimated probability that the individual has diabetes is y_0/n_0. We will call y_0/n_0 an *empirical probability* estimate as it is based purely on the relative frequency of reported diabetes diagnoses among the sampled individuals with the same demographic profile. Mathematically, if f is the estimator that maps demographic profiles to empirical probabilities, then $f(\mathbf{x}_0|D) = y_0/n_0$, where D is the data set from which the demographic profiles were constructed. We cannot compactly express the function mathematically, though in essence, f is a dictionary in which the keys are vectors $\mathbf{x}_1, \ldots, \mathbf{x}_N$ and the values are empirical probabilities $y_1/n_1, \ldots, y_N/n_N$.

The predictive function is not based on a model but instead is based entirely on the available data. Therefore, the predictive function is free of error attributable to model inadequacies. Being free of a model and the burden of defending the model is good. But using these probabilities for prediction produces imprecise estimates when the number of observations with a particular demographic profile is small. Without a model, the precision of a prediction depends only on n_0, the number of observations in the data set that match a target predictor vector $\mathbf{x}_0$ no matter how large the data set. Thus, if the target individual is unusual and n_0 is small, then neither the associated probability estimate nor the prediction will be precise.[14] A predictive *model* usually will produce more precise estimates than a model-free predictive function if the model is a close approximation of reality. In a sense, the predictive model gains an advantage over the model-free predictive function because all ob-

[13] If the BRFSS samples were random samples, then we would call the probability estimate an *empirical probability.*

[14] The precision of a prediction is directly related to the variance of the estimator, and the variance depends on the number of observations used to compute the estimate.

servations are used for estimating a model. The model is used to produce a prediction and all of the observations contribute to the prediction though not equally. Despite the potential advantages of a model-based prediction function, we proceed in this case with the model-free approach because the volume of data is sufficiently large to insure precision for the vast majority of potential predictions. Furthermore, we need not be concerned with the myriad of difficulties in finding a realistic model.

The demographic variables used for prediction are income, education, age, and body mass index.[15] The BRFSS variables that record income, education, age are ordinal; for instance, the age of a respondent is recorded as an interval (e.g., age between 30 and 34 years). These variables are ordinal because any two values can be unambiguously ordered but the interpretation of the difference between values is not necessarily unambiguous. In contrast, body mass index is computed from height and weight and is recorded on the scale kg/m^2. It's convenient, however, to transform body mass index to an ordinal scale by rounding each value to the nearest even integer for values between 18 and 60 kg/m^2. Values greater than 60 and less than 18 are transformed to 60 and 18, respectively. The result is 22 distinct body mass index categories.

Table 7.5 identifies the number of levels of each demographic variable. The number of possible combinations of levels is $14 \times 8 \times 6 \times 22 = 14{,}784$. Each combination is associated with a set of individuals, and we refer to a set of individuals with a common predictor vector (or demographic profile) as a *cohort* and the predictor vector as a *profile vector*. The number of cohorts or cells, if we envision a four-dimensional cross-tabulation of the variables, would be prohibitively large for using empirical probabilities for prediction if this were a conventional data set. Using the data sets from 2000 through 2014 yielded $n = 5{,}195{,}986$ records with informative values on all four variables. The average number of observations per profile vector is 364.1.[16] This average number of observations is satisfactory for most applications of empirical probabilities. Of course, the distribution of observations across cells is variable and there are cells with few or no observations. We'll address that issue in the next section. Before putting the data to use, we ought to think more about the predictive function and its application.

Table 7.5 Ordinal variables and the number of levels of each

Variable	Age	Income	Education	Body mass index
Number of levels	14	8	6	22

The predictive function consumes a vector, say $\mathbf{x}_0 = [3 \quad 4 \quad 4 \quad 36]^T$ and returns an empirical probability; specifically, $f(\mathbf{x}_0|D) = 27/251 = .108$. In a traditional statistical learning application, a prediction of whether an

[15] Other variables are potentially useful for prediction (race and exercise level).

[16] Not every possible profile was observed. The number of observed profiles was 14,270, slightly less than 14,784.

individual with the profile $\mathbf{x}_0$ has diabetes would be generated from the estimate. Ordinarily, the decision rule would be: if $f(\mathbf{x}_0|D) > .5$, then we predict the individual to have diabetes and if $f(\mathbf{x}_0|D) \leq .5$, then we predict the individual not to have diabetes.

But this problem is somewhat different. Our aim is to identify individuals with elevated risk of developing diabetes rather than predicting whether an individual has diabetes. Individuals with elevated risk may or may not have the disease. Identifying individuals that have diabetes would only be of interest if there were a significant proportion of individuals with diabetes that were undiagnosed. So, in this application, the event of interest is elevated risk of developing diabetes. As stated, the event is ambiguous and we need to define the event unambiguously to proceed to the next step of labeling individuals as having elevated or normal risk. We do so by defining a decision rule $g(\mathbf{x}, p)$ such that $g(\mathbf{x}, p) = 1$ identifies elevated risk and $g(\mathbf{x}, p) = 0$ identifies normal risk. The decision rule is

$$g(\mathbf{x}, p) = \begin{cases} 1, & \text{if } f(\mathbf{x}|D) > p, \\ 0, & \text{if } f(\mathbf{x}|D) \leq p. \end{cases} \tag{7.7}$$

The argument p is a threshold that is adjusted in consideration of the costs of failing to identify an at-risk individual and mislabeling a normal-risk individual as having elevated risk. The data set does not contain information on risk status, but logically, most of the individuals with diabetes ought to be identified as having elevated risk by g. However, not all individuals with diabetes who have a demographic profile indicative of the disease will have the disease since there is variability in susceptibility and age of onset. Further, a person's diet and exercise level tends to vary over time. So, an individual may no longer exhibit the attributes (body mass index, in particular) indicative of the disease but may respond affirmatively to the question about diabetes diagnosis. Some individuals that are identified as having elevated risk will be self-identified as *not* having diabetes. Determining who these individuals are would be of substantial interest in a practical application of the decision rule given that the objective is to intervene and reduce elevated risk levels. Of course, the decision rule must be sufficiently accurate for it to be of any value.

7.4.1 Sensitivity and Specificity

The accuracy of the decision rule can be investigated by forming the following pairs: $(y_1, g(\mathbf{x}_1, p))$,. . ., $(y_N, g(\mathbf{x}_N, p))$ where y_i is 1 if the ith individual was self-identified as having diabetes and 0 if the ith individual was not self-identified as having diabetes. Equation (7.7) defined $g(\mathbf{x}_i, p)$. With respect to performance of the rule, it's most important that those individuals having diabetes are identified by the decision rule as having elevated risk. Otherwise, the decision rule is not of much use. To approach the problem analytically, consider the conditional probability that an individual will be identified as having elevated risk given that they have diabetes. This probability is

$$\Pr[g(\mathbf{x},p)=1|y=1]=\frac{\Pr[g(\mathbf{x},p)=1 \text{ and } y=1]}{\Pr(y=1)}. \tag{7.8}$$

The conditional probability $\Pr[g(\mathbf{x},p)=1|y=1]$ is called the *sensitivity* of the decision rule and also the *true positive rate* [2]. A good rule will have a sensitivity close to one. It is also desirable that the *specificity*, or true negative rate $\Pr[g(\mathbf{x},p)=0|y=0]$ is large in conventional testing situations (which this is not). In this context, the specificity describes the probability that a person without diabetes is identified as having normal risk.

Our aim is not to predict diabetes but instead to predict elevated risk.[17] Therefore, we expect that if the decision rule is applied to the data, then there will be instances of identifying someone as having elevated risk when they did not identify themselves as having diabetes. Traditionally, this event is referred to as a false positive and can be expressed mathematically as the event $\{g(\mathbf{x},p)=1|y=0\}$. In this situation, a moderately large false positive rate does not imply the decision rule is poor because it's expected that some at-risk individuals in the BRFSS samples will not have been diagnosed as diabetic. The false positive rate is related to the specificity since $\Pr[g(\mathbf{x},p)=1|y=0]=1-\Pr[g(\mathbf{x},p)=0|y=0]$. Therefore, the specificity $\Pr[g(\mathbf{x},p)=0|y=0]$ need not be large for the decision rule to be deemed useful.

Computing estimates of sensitivity and specificity begins with mapping the set $\{(y_1,g(\mathbf{x}_1,p)),\ldots,(y_N,g(\mathbf{x}_N,p))\}$ to a confusion matrix of the form shown in Table 7.6. Confusion matrices are discussed at length in Sect. 9.7.1. The table entries are the numbers of individuals with a particular combination of self-identified disease state (diabetes or not) and a predicted risk level (elevated or normal). Sensitivity is estimated by

Table 7.6 A confusion matrix showing the classification of risk prediction outcomes of n_{++} individuals

	Predicted risk		
Self-identified	Normal	Elevated	Total
No diabetes	n_{00}	n_{01}	n_{0+}
Diabetes	n_{10}	n_{11}	n_{1+}
Total	n_{+0}	n_{+1}	n_{++}

$$\widehat{\Pr}[g(\mathbf{x},p)=1|y=1]=\frac{n_{11}}{n_{1+}}, \tag{7.9}$$

[17] We could define the event of interest more rigorously as metabolic syndrome, a set of medical conditions that are considered to be precursors to type 2 diabetes.

where n_{1+} is the number of individuals that have self-reported diabetes, and n_{11} is the number of individuals that have self-reported diabetes and were predicted to have elevated risk. Specificity is estimated according to

$$\widehat{\Pr}[g(\mathbf{x}, p) = 0|y = 0] = \frac{n_{00}}{n_{0+}}. \tag{7.10}$$

The false positive rate estimate is

$$\widehat{\Pr}[g(\mathbf{x}, p) = 1|y = 0] = \frac{n_{01}}{n_{0+}}.$$

The threshold p determines whether the empirical probability $f(\mathbf{x}|D)$ associated with a profile vector $\mathbf{x}$ identifies an individual as having elevated risk. Small values of p result in more individuals being identified as having elevated risk. Consequently, lowering the threshold causes n_{+1} to increase and n_{+0} to decrease. Exactly how the increase in n_{+1} is distributed between n_{01} and n_{11} is unknown, but we may expect some increase n_{11}, the number of individuals that have diabetes and have elevated risk. Hence, sensitivity is likely to be improved by reducing p. Lowering the threshold is also likely to reduce the specificity of the prediction function by increasing n_{01}, the proportion of individuals identified as having elevated risk when in fact they do not have diabetes. Figure 7.2 shows that if we hold sensitivity and specificity equally important, then the choice of $p = .13$ will result in both estimated sensitivity and specificity roughly equal and .71 in value. If the prediction function is applied to the BRFSS population, then it's estimated that 71% of the time, a diabetic individual will be identified as having elevated risk. As discussed above, accepting smaller values for specificity than for specificity is reasonable and even desirable. Hence, the choice $p = .1$ may be preferred by noting that we identify an individual as having elevated risk if their demographic cohort contains at least 10% diagnosed diabetics. In this context, a demographic cohort is a set individuals with the same demographics—specifically, with the same predictor vector. Using $p = .1$ as the threshold results in a sensitivity of .806 and specificity of .622. With this choice of p, the proportion of individuals that are identified as having elevated risk but did not report a diagnosis of diabetes is $1 - .622 = .378$. These individuals would be targeted for intervention.

7.5 Tutorial: Identifying At-Risk Individuals

This tutorial builds on the previous and earlier tutorials. In particular, all of the variables have been used before though not all at the same time. We use the data files from years 2002 through 2014. Some of these data were used in the Sect. 7.3 tutorial. Data from years 2002 through 2006 will have

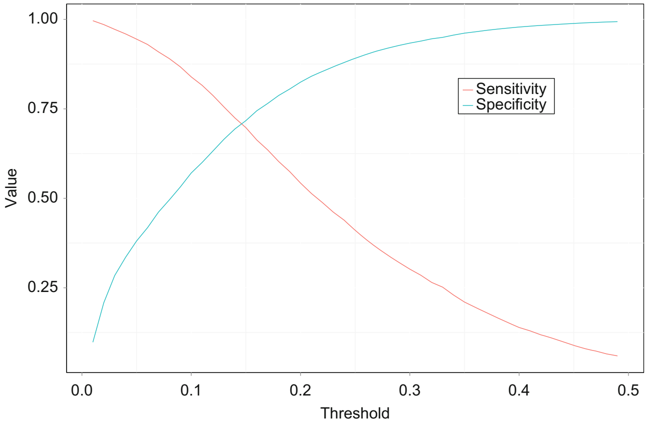


Fig. 7.2 Sensitivity and specificity plotted against the threshold p. Data from years 2001 through 2014, $n = 5{,}195{,}986$

to be retrieved from the Centers for Disease Control and Prevention webpage (`http://www.cdc.gov/brfss/annual_data/annual_data.htm`). Table 7.1 summarizes the field positions by year for the variables to be used in the tutorials of this chapter.

1. Modify or create, a dictionary `fieldDict` that contains the field locations of the variables of interest by year. Field locations are specified in Table 7.1. As in the tutorial of Sect. 7.3, `fieldDict` is a dictionary of dictionaries. The outer dictionary uses the last two digits of the year as the key. For this tutorial, the associated value is an inner dictionary with five key-value pairs. Each key of the inner dictionary is a variable name and the value is the field location of the variable. The initialization of the dictionary is shown as well as the instruction to fill the inner dictionary associated with year 2010.

```
fieldDict = dict.fromkeys([10, 11, 12, 13, 14])
fieldDict[10] = {'bmi':(1483, 1486), 'diabetes':87,
  'income':(120, 121), 'education':118, 'age':(1468, 1469)}
```

   The dictionary may contain additional variables beyond those used in this tutorial. They have no effect on the program.

2. Iterate over the files in the directory containing the BRFSS data files using the code segment from instruction 9 of Sect. 7.3.
3. Create a `ZeroDivisionError` error if `shortYear` is less than 2 so that only data files for years 2001 through 2014 are processed:

```
try:
    shortYear = int(filename[6:8])
    if shortYear < 2:
        1/0
    ...
except(ZeroDivisionError, ValueError):
    pass
```

4. Extract the dictionary `fields` from `fieldDict`. Then, extract the field positions for age using the command `sAge, eAge = fields['age']`.
5. Extract the field positions for the other four variables income, education, body mass index, and diabetes. Instruction 10 of Sect. 7.3 shows the code for extracting weight and body mass index. The field positions are extracted once for each data file and so belong in the code segment above denoted by `...`
6. Open the file with the path and name `file` and process each record in sequence:

   ```
   with open(file, encoding = "utf-8") as f:
       for record in f:
           n += 1
   ```

   This code must be aligned with the instructions that retrieve the field positions of the variables.
7. Use the functions `getIncome` and `getEducation` that were programmed in Chap. 3, Sect. 3.8 to translate the entries in `record` to usable values. If these functions are already in your `functions` module, then load the module and the functions by placing the instructions

   ```
   sys.path.append('/home/.../parent')
   from PythonScripts import functions
   dir(functions)
   ```

   at the top of your script. Here, `/home/.../parent` is the path to the parent directory that contains `functions.py`.[18] Instruction 8 of Chap. 3, Sect. 3.8 provides details on programming the functions.
8. Translate the body mass index string to an integer using the previously programmed function `convertBMI` developed in instruction 14, Chap. 3, Sect. 3.6. Then, map the integer value of body mass index to an ordinal

[18] `functions.py` should reside in a directory below `parent`. For instance, the full path might be `/home/HealthCare/PythonScripts/functions.py`, in which case `parent` is `/home/HealthCare`.

variable. The calculation `int(2*round(bmi/2,0))` rounds body mass index to the nearest even integer. Finally, we map values greater than 60 to 60 and values less than 18 to 18.

```
bmi = functions.convertBMI(record[sBMI-1:eBMI],shortYear)
bmi = int(2*round(bmi/2,0))
if bmi > 60:
    bmi = 60
if bmi < 18:
    bmi = 18
```

9. Extract the diabetes entry and convert it to a binary response using the function `getDiabetes` programmed in instruction 13 of Sect. 7.3.

```
y = functions.getDiabetes(record[fDia-1])
```

The argument passed to `getDiabetes` is the one-character string extracted from `record`.

10. Create the profile vector **x** as a four-tuple but only if the values of all five variables are informative. If the response to the education or income questions was a refusal to answer or *don't know*, then the answer was coded as 9. Do not create **x** if that is the case.

```
if education < 9  and income < 9 and 0 < age < 15 and bmi != 0
        and (y == 0 or y == 1):
    x = (income, education, age, bmi)
print(y, income, education, age, bmi)
```

11. If the profile vector was created in the previous instruction, then update `diabetesDict`. The profile vector is the key and the value is a two-element list $[n, \sum_i^n y]$.

```
value = diabetesDict.get(x)
if value is None:
    value = [1, y]
else:
    value[0] += 1
    value[1] += y
diabetesDict[x] = value
```

12. The remainder of the program computes the sensitivity and specificity of a decision rule $g(\mathbf{x}, p)$ where p is a threshold (Eq. (7.7)). We'll compute sensitivity and specificity for one value of p and then extend to the code to

compute sensitivity and specificity over a range of values for p. The data for computing sensitivity and specificity will be contained in a confusion matrix.

Consequently, the next step is to write the code for building a confusion matrix (Table 7.6). The confusion matrix summarizes the cross-classification of every individual in `diabetesDict` according to their recorded condition (diabetes diagnosis or not) and their prediction of risk (elevated or not). The rows correspond to reported diabetes (absent or present) and the columns correspond to the predicted risk (normal or elevated).

```
threshold = .3
confusionMatrix = np.zeros(shape = (2, 2))
for x in diabetesDict:
    n, y = diabetesDict[x]
    if y/n > threshold:             # Predicted to have elevated risk.
        confusionMatrix[0,1] +=  n - y  # Diabetes not reported.
        confusionMatrix[1,1] +=  y      # Diabetes reported .
    if y/n <= threshold:            # Predicted to have normal risk.
        confusionMatrix[0,0] +=  n - y  # Diabetes not reported.
        confusionMatrix[1,0] +=  y      # Diabetes reported.
```

The boolean result of the statement `y/n > threshold` determines whether the prediction function $g(\mathbf{x}, p)$ yields a prediction of elevated risk. Suppose that `y/n > threshold` is true. The prediction is the same for all n individuals with the profile vector $\mathbf{x}$ because all n have the same profile and are considered to have the same risk (namely, y/n based on the available data). Of those n individuals, y reported a diabetes diagnosis and $n - y$ did not report a diagnosis. Therefore, we add y to the count of individuals that were predicted to have elevated risk and were reported to have a diabetes diagnosis. This count is in `confusionMatrix[1,1]`. The remaining $n - y$ individuals did not report a diabetes diagnosis yet were identified as having elevated risk, and we add $n - y$ to the count in `confusionMatrix[0,1]`.

On the other hand, if `y/n <= threshold` is true, the same logic applies to the assignment of counts. Hence, y is added to `confusionMatrix[1,0]` and $n - y$ is added to `confusionMatrix[0,0]` .

13. The terms for computing sensitivity and specificity are extracted from the confusion matrix after the `for` loop has completed. Formulas 7.9 and 7.10 are used to compute sensitivity and specificity:

```
n00, n01 = confusionMatrix[0,:]
n10, n11 = confusionMatrix[1,:]
sensitivity = n11/(n10 + n11)
specificity = n00/(n00 + n01)
print('sensitivity:', n10, n11, round(sensitivity, 3))
print('specificity', n00, n01, round(specificity, 3))
```

14. The next task is to compute sensitivity and specificity over a range of threshold values $.01, .02, \ldots, .99$ and store the values in a dictionary that uses `threshold` as a key. Begin by creating a vector of threshold values. Use the `Numpy` function `arange` to create a sequence beginning with .01, ending with .99, and incrementing by .01:

```
thresholds = np.arange(start=.01, stop=1.0, step =.01)
```

15. Initialize a dictionary `ssDict` to contain the sensitivity and specificity values for different threshold values and build a `for` loop that iterates over the threshold values. On each iteration of the `for` loop, a decision rule $g(\mathbf{x}, p)$ will be applied to the observations in `dataDict` using the current threshold p. Then, we fill the confusion matrix for the rule, compute sensitivity and specificity of the rule, and store the values in `ssDict`. The first operation of the `for` loop initializes the confusion matrix.

```
ssDict = dict.fromkeys(thresholds)
for threshold in ssDict:
    confusionMatrix = np.zeros(shape = (2,2))
```

16. Insert the code segment (instruction 13) for computing the sensitivity and specificity inside the `for` loop and after the initialization of the confusion matrix. After computing these terms store the sensitivity and specificity values as a list in `ssDict` using the threshold as the key.

```
ssDict[threshold] = [x11/(x10+x11), x00/(x00+x01)]
```

    The left-position element (`ssDict[threshold][0]`) is sensitivity and the right-position element is specificity.

17. After the `for` loop over `ssDict` has completed, plot sensitivity and specificity against threshold using `mathplotlib`. Specifically, plot sensitivity against threshold and then specificity against threshold. It's necessary to sort the values of threshold from smallest to largest and apply the same sort to sensitivity and specificity before plotting. The first operations are to import the module `operator` for sorting and the `pyplot` function from `matplotlib`. Then, sort the dictionary according the threshold.

```
import operator
import matplotlib.pyplot as plt
sortedList = sorted(ssDict.items(), key = operator.itemgetter(1))
```

18. The call to `sorted` returned a list. Each item in the list is a pair in which the first item (`item[0]`) is a key and the second item (`item[1]`) is a two-element list consisting of sensitivity and specificity. Extract the variables from `sortedList` and create the plot.

```
ySE = [value[0] for _,value in sortedList]
ySP = [value[1] for _,value in sortedList]
xS = [item[0] for item in sortedList]
plt.plot(xS, ySE)
plt.plot(xS, ySP)
```

    You should obtain a figure similar to Fig. 7.2.

19. A slightly different predictive function will be built in the next tutorial. We'll recompute `ssDict` using the new rule. To lessen the programming effort, turn your code for building `ssDict` into a function. Build the function around the code segment beginning with the initialization of the vector of thresholds (instruction 14) and ending with the instruction that stores the sensitivity and specificity values as a list in `ssDict`.

```
def ssCompute(diabetesDict):
    thresholds = np.arange(start= .01, stop=1.0,step = .01)
    ssDict = dict.fromkeys(thresholds)
    for threshold in ssDict:
        ...
        ssDict[threshold] = [x11/(x10+x11), x00/(x00+x01)]
    return ssDict
```

20. Move the function to the top of the script. Test the function with the following code segment.

```
ssDict = ssCompute(diabetesDict)
sortedList =  sorted(ssDict.items(), key=operator.itemgetter(1))
ySE = [value[0] for _,value in sortedList]
ySP = [value[1] for _,value in sortedList]
plt.plot(xS,ySP)
plt.plot(xS,ySE)
```

    If you move `ssCompute` to your `functions.py` module, then put the instruction `import numpy as np` at the top of `functions.py`. The `Numpy` library has to be imported into every file that uses it (note that we have called the `Numpy` function `np.arange` in instruction 14). The function call is then `ssDict = functions.ssCompute(diabetesDict)`.

7.6 Unusual Demographic Attribute Vectors

There remains an outstanding problem that needs to be addressed regarding estimating risk and predicting elevated risk of developing diabetes. Suppose that the demographic profile of a target individual is described by the vector $\mathbf{x}_0$ and that $\mathbf{x}_0$ is not among the set of observed profile vectors $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$. Since $\mathbf{x}_0 \notin \mathbf{X}$, the empirical probability of a positive response to the diabetes question is unknown. The predictive function developed in the previous section does not permit a determination of predicted risk. But even if $\mathbf{x}_0 \in \mathbf{X}$, it may be that the number of respondents associated with the vector $\mathbf{x}_0$ is small and the empirical probability will then be imprecise.

The extent of the problem is not large but cannot be ignored. Of the possible 14,784 profile vectors, $N = 14{,}270$, or 96.7%, were observed among the $n = 5{,}195{,}986$ informative observations. It's entirely possible that a prediction may be needed for some $\mathbf{x}_0 \notin \mathbf{X}$.

Furthermore, among those demographic vectors that were observed (and hence, are elements of $\mathbf{X}$), 8094 are associated with at least 30 sampled individuals. The remaining 6176 demographic vectors are each associated with fewer than 30 sampled individuals. The empirical probabilities associated with these sample-size deficient vectors are imprecise though the pejorative *imprecise* is subjective. Certainly, the precision of the sample-deficient profile estimates is worse than the estimates associated with commonly occurring profile vectors. We note parenthetically that these sample size deficient profiles do not significantly impact our estimates of sensitivity and specificity because only 1.03% of all observations in the data set are associated with what we might call unusual profile vectors. Furthermore, if the prediction function were applied to a population with a similar demographic structure to the BRFSS samples, then only 1% of the time would a prediction of risk originate from one of the unusual profiles.

It's necessary, however, to accommodate new and unusual profiles if demographic profile vectors truly are to be used in a practical application. Before proceeding toward a solution, let us recall some notation. For $\mathbf{x}_0 \in \mathbf{X}$, n_0 denotes the number of observations in the data set associated with $\mathbf{x}_0$, and y_0 denotes the number of individuals among the n_0 that responded affirmatively to the question regarding a diabetes diagnosis. We solve the problem of a new or unusual profile vector by a finding a set of observations that are similar with respect to demographics. Mathematically, we find a set of $\mathbf{x}_j$'s in a neighborhood of $\mathbf{x}_0$, the target profile. The neighborhood set consists of observations that are close to $\mathbf{x}_0$ with respect to a distance function computed on the profile vectors. Then, the empirical probability for $\mathbf{x}_0$ is computed using y_0 and n_0 and the observations in the neighborhood of $\mathbf{x}_0$.[19]

[19] The algorithm is essentially an implementation of the one-nearest neighbor prediction function.

The algorithm examines each observation triple $(y_0, n_0, \mathbf{x}_0)$ in the data dictionary. If n_0 is insufficiently large, say $n_0 \leq 50$, then we will find a set of profile vectors $\{\mathbf{x}_1, \ldots, \mathbf{x}_q\}$ that are most similar to $\mathbf{x}_0$ and construct the nearest neighbors set $N_1(\mathbf{x}_0) = \{(y_1, n_1, \mathbf{x}_1), \ldots, (y_q, n_q, \mathbf{x}_q)\}$. The set $N_1(\mathbf{x}_0)$ contains all of the neighbors whose distance to $\mathbf{x}_0$ is equal to the minimum distance. Then, we compute the neighborhood sums

$$\begin{aligned} y_0^* &= y_0 + \textstyle\sum_{j=1}^q y_j = y_i + \sum_{(y_j, n_j, \mathbf{x}_j) \in N_1(\mathbf{x}_0)} y_j \\ n_0^* &= n_0 + \textstyle\sum_{j=1}^q n_j = n_i + \sum_{(y_j, n_j, \mathbf{x}_j) \in N_1(\mathbf{x}_0)} n_j \quad . \end{aligned} \tag{7.11}$$

The original empirical probability estimate y_0/n_0 is then replaced by y_0^*/n_0^* in the dictionary of profiles. If n_0^* is still less than the threshold sample size (50 in the example above), then $N_2(\mathbf{x}_0)$, the set of neighbors whose distance to $\mathbf{x}_0$ is equal to the second-smallest distance is constructed, and y_0^* and n_0^* are computed from the larger neighborhood $N_1(\mathbf{x}_0) \cup N_2(\mathbf{x}_0)$. We continue merging neighborhood sets until n_0^* is sufficiently large. The result is an updated prediction function $f^*(\cdot|D)$ that operates on a target $\mathbf{x}_0$ by searching the profile set $\mathbf{X}$ for a matching profile and assigning the empirical probability estimate associated with the matching profile to $\mathbf{x}_0$.

We haven't yet addressed the problem presented by an individual whose profile is not among the observed profile vectors contained in $\mathbf{X}$. Our solution extends the method used to revise the empirical probability estimates for sample deficient profiles. Suppose that the individual's profile is $\mathbf{x}_0$. The algorithm determines the nearest neighbor set $N_1(\mathbf{x}_0) = \{(y_1, n_1, \mathbf{x}_1), \ldots, (y_q, n_q, \mathbf{x}_q)\}$ and computes the empirical probability estimate p_0 according to

$$\begin{aligned} y_0^* &= \textstyle\sum_{j=1}^q y_j, \\ n_0^* &= \textstyle\sum_{j=1}^q n_j, \\ \text{and } p_0 &= y_0^*/n_0^*. \end{aligned}$$

We don't have to worry about the size of the denominator n_0^* since the algorithm for boosting the sample sizes of the data set profile vectors (formula (7.11)) insures that all empirical probability estimates are computed from sufficiently many sample observations.

The next matter to address is that of determining a nearest neighbor set for a given $\mathbf{x}_0$. Our algorithm computes the distance between $\mathbf{x}_0$ and each $\mathbf{x}_i \in \mathbf{X}$ as the city-block or Manhattan distance between vectors. The city-block distance between $\mathbf{x}_i$ and $\mathbf{x}_0$ is

$$d_C(\mathbf{x}_i, \mathbf{x}_0) = \sum_{j=1}^p |x_{i,j} - x_{0,j}|, \tag{7.12}$$

where p is the number of variables that define a profile.[20] A formal definition of the neighbor set is

$$N_1(\mathbf{x}_0) = \left\{ (y_i, n_i, \mathbf{x}_i) | d_C(\mathbf{x}_i, \mathbf{x}_0) = \min_{\mathbf{x}_k \in \mathbf{X}} d_C(\mathbf{x}_k, \mathbf{x}_0) \right\}.$$

The set $N_1(\mathbf{x}_0)$ may contain one neighbor but often will contain more than one because the smallest distance may be tied among several neighbors.

The following tutorial implements an algorithm for finding neighborhood sets and revising the empirical probabilities for those sample-deficient demographic vectors.

7.7 Tutorial: Building Neighborhood Sets

This tutorial begins where the tutorial of Sect. 7.6 left off. It's assumed that the dictionary `diabetesDict` has been computed. In this dictionary, a key is a profile vector $\mathbf{x}_i$ and the value is a data list $[n_i, y_i]$. The algorithm begins by creating two dictionaries from `diabetesDict`. The first dictionary consists of key-value pairs with commonly occurring profiles vectors (or keys). The second dictionary consists of key-value pairs with unusual profiles vectors. For each profile in the dictionary of unusual demographic profiles, we replace the entry $[n_i, y_i]$ with $[n_i^*, y_i^*]$ (Eq. (7.11)).

1. Initialize two dictionaries, `freqDict` and `infreqDict` to store the triples $(y_i, n_i, \mathbf{x}_i)$.

```
freqDict = {}
infreqDict = {}
count = [0]*2
```

   The dictionary `freqDict` will contain the profiles for which there were at least 50 observations and `infreqDict` will store those profile with less than 50 observations.

2. Fill the two dictionaries by separating the entries in `diabetesDict` according to the value of n_i.

[20] In the tutorial of Sect. 7.5, the predictor variables are age, education, income, and body mass index and so $p = 4$.

```
count = [0]*2
for x in diabetesDict:
    n, y = diabetesDict[x]
    if n >= 50:
        freqDict[x] = n, y
        count[0] += n
    else:
        infreqDict[x] = n, y
        count[1] += n
print(len(diabetesDict),len(freqDict),len(infreqDict))
print(count)
```

The list `count` will contain the number of observations from which each dictionary was built.

3. Build a function for computing the city-block distance (formula (7.12)) between vectors $\mathbf{x}_i$ and $\mathbf{x}_0$. Use the `zip` function to produce an iterable object consisting of the pairs $(x_{i,1}, x_{0,1}) \ldots, (x_{i,p}, x_{0,p})$. Each pair consists of two measurements made on the same demographic attribute.

   Iterate over the `zip` object, extract the elements of each pair as `w` and `z`, compute the absolute differences, and add the absolute difference to the total.

```
def dist(xi, x0):
    s = 0
    for w, z in zip(xi, x0):
        s += abs(w - z)
    return s
```

4. Now we begin the process of replacing the pairs (y_0, n_0) for those $\mathbf{x}_0$ that are unusual because $n_0 < 50$.

   Iterate over the dictionary `infreqDict` and extract each profile vector $\mathbf{x}_0$ in turn. Compute the distance between $\mathbf{x}_0$ to every profile vector $\mathbf{x} \in$ `freqDict` and find the smallest distance between $\mathbf{x}_0$ and every $\mathbf{x} \in$ `freqDict`.

```
for x0 in infreqDict:
    d = [dist(x0, x) for x in freqDict]
    mn = min(d)
```

5. Find the neighborhood set $N_1(\mathbf{x}_0)$ for each $\mathbf{x}_0 \in$ `infreqDict`. First initialize a list named `nHood` to contain the lists $[n_i, y_i]$ from which the updates to n_0 and y_0 will be computed (Eq. (7.11)). Then, iterate over `freqDict` and compute the distance between $\mathbf{x} \in$ `freqDict` and $\mathbf{x}_0$. If the distance between $\mathbf{x}$ and $\mathbf{x}_0$ is equal to the minimum distance `mn`, then extract the list $[n, y]$ associated with $\mathbf{x}$ and append the list to `nHood`:

```
nHood = [infreqDict[x0]]
for x in freqDict:
    if dist(x0,x) == mn:
        lst = freqDict[x]
        nHood.append(lst)
```

This code segment follows immediately after the statement `mn = min(d)`.

The neighborhood list `nHood` was initialized with the list $[n_0, y_0]$ contained in `infreqDict[x0]` since the sums over the neighborhood shown in Eq. (7.11) include n_0 and y_0.

6. When the `for` loop over `freqDict` has completed, compute the sums given in Eq. (7.11). Store the revised data list $[n_0^*, y_0^*]$ in `diabetesDict`:

```
nStar = sum([n for n,_ in nHood])
yStar = sum([y for _,y in nHood])
diabetesDict[x0] = [nStar, yStar] # Careful! Don't reverse the oder.
```

This code segment must be aligned with the `for x in freqDict:` statement.

Warning: the last statement changes the value of an existing dictionary. If there are any errors in the calculation of n_0^* and y_0^* then you'll have to recreate `diabetesDict`.

7. The process of updating `diabetesDict` is slow because of the distance calculation and the search for the nearest neighbors. Add a counter to track the iterations and print the value of the counter every 100 iterations. Execute the code segment beginning with the `for` loop that iterates over `infreqDict`.
8. The tutorial of Sect. 7.5 instructed the reader to write a function, `ssCompute`, for the purpose of computing sensitivity and specificity for each threshold value in a vector of values. Using that function, build a new version of `ssDict` containing sensitivity and specificity values over the range of threshold values $.01, .02, \ldots, .99$.
9. Recreate the plot built in instruction 17 of the Sect. 7.5 tutorial.

7.7.1 Synopsis

In the tutorials of Sects. 7.5 and 7.7, we've tackled the problem of estimating the risk of an adult developing diabetes. The data reduction algorithms that were used are exemplars of the core algorithms of data science. What was done was to reduce nearly 5.2 million observations to 14,270 demographic profiles. Each demographic profile characterizes a cohort,[21] and collectively, the profiles are a computationally efficient representation of the population. In the tutorial of Sect. 7.5, the risk of developing diabetes was estimated for each profile. When presented with a new individual, our estimate of the individual's risk is obtained by determining their demographic profile, the target profile, and matching the target profile with a profile contained in the dictionary of profiles. Their risk estimate is the risk associated with the matching dictionary profile.[22] The risk estimate is the empirical probability of diabetes for the cohort, that is, the sample proportion of individuals with the target profile that responded affirmatively to the diagnosis question. The statistics used in the prediction function are trivial, and building the prediction function would have been trivial as well except for the step of merging infrequently occurring demographic profiles with frequently occurring profiles.

In a practical application, one would estimate the risk of other chronic diseases tracked in the Behavioral Risk Factor Surveillance System survey (asthma, chronic obstructive pulmonary disease, and perhaps mental illnesses, for example).

Our approach to the problem of predicting risk exemplifies the differences between data science and traditional statistics. The standard statistical approach to the problem of estimating risk would involve logistic or binomial regression modeling. A model, or course, is supposed to be an abstraction, a simplification of reality that promotes understanding. A model of risk as a function of the four demographic variables provides at most a superficial understanding of a highly complex process. If we focus on the model not as an abstraction of reality but simply as a predictive tool then other limitations come into view. Most notably, the model is constrained to a particular form such as linear combination of the four demographic variables and perhaps some transformations of the variables. A fitted regression model would be highly constrained because every estimate is determined by the fitted model and the fitted model is determined collectively by all observations. Every observation has some leverage on the fitted model. Constructing demographic profiles and estimating risk using the empirical probabilities produces a prediction function free of model constraints. The estimate associated with a particular profile is completely unrelated to the estimate for any other profile.

Interestingly, a common goodness-of-fit test for binomial regression uses the empirical probabilities as a gold standard against which the fitted model is compared. If the fitted model comes close the replicating the empirical probability estimates, then the model is judged to fit the data well. We have no model. We simply reduced the data to a convenient form and computed

[21] A cohort is a population subgroup with similar characteristics

[22] In the unlikely event that the target profile is not in the dictionary, we find a set of most similar profiles in the dictionary.

the empirical probability of diabetes for each profile. Our prediction function is essentially the gold standard against which the performance of the model is compared.

7.8 Exercises

7.8.1 Conceptual

7.1. Consider the variance of an estimator of prevalence for year x_0 obtained from a linear regression of prevalence on year:

$$\text{var}[\widehat{\mu}(x_0)] = \sigma^2 \left(\frac{1}{n} + \frac{(x_0 - \overline{x})^2}{\sum (x_i - \overline{x})^2} \right),$$

where x_0 is a user-selected year of interest and x_i is the ith year.

a. What choice of x_0 minimizes $\text{var}[\widehat{\mu}(x_0)]$?
b. Suppose that the n years are consecutive. Argue that selecting the midpoint of the time span to be x_0 yields the smallest possible value of $\text{var}[\widehat{\mu}(x_0)]$ among all choices of x_0.

7.2. Argue that Eq. (7.3) is correct by determining the vector $\widehat{\boldsymbol{\beta}}_w$ that minimizes the objective function

$$\begin{aligned} S(\boldsymbol{\beta}) &= \sum w_i (y_i - \mathbf{x}_i^T \boldsymbol{\beta})^2 \\ &= (\mathbf{Y} - \mathbf{X}\boldsymbol{\beta})^T \mathbf{W} (\mathbf{Y} - \mathbf{X}\boldsymbol{\beta}) \end{aligned} \tag{7.13}$$

where $\mathbf{W} = \text{diag}(w_1 \ \cdots w_n)$. Hint: differentiate $S(\boldsymbol{\beta})$ with respect to $\boldsymbol{\beta}$.

7.3. The Centers for Disease Control and Prevention's discussion of the BRFSS sampling design makes it clear that disproportionate sampling must be accounted for by the sampling weights. To gain some insight into the origin of the sampling weights, let us consider the following idealized problem. The objective is to estimate of the mean of a static and finite population $\mathcal{P} = \{y_1, y_2, \ldots, y_N\}$ from a sample of n observations.

a. Give an expression for the population mean μ involving the elements of $\mathcal{P}$.
b. Suppose that a sample of $1 \leq n \leq N$ observations is drawn randomly and with replacement from $\mathcal{P}$. What is the probability π_j that y_j will be included in the sample?
c. Show that the sample mean $\overline{Y} = n^{-1} \sum_{j=1}^{n} Y_j$ is unbiased for μ. An estimator is unbiased if its expectation equals the target value. Specifically, show that $E(\overline{Y}) = \mu$.
d. Suppose now that the sample is obtained not by random sampling, but by some design that draws observations with replacement and for which the

probability that the jth draw selects y_i is $\Pr(Y_j = y_i) = \pi_i$, for $j = 1, \ldots, n$ and $i = 1, \ldots, N$. The sampling probabilities need not be constant, though it is necessary that $\sum_{i=1}^{N} \pi_i = 1$. Show that the estimator

$$\overline{Y}_w = N^{-1} \sum_{i=1}^{n} w_i y_i,$$

where $w_i = p_i^{-1}$ is unbiased for the mean.

7.4. Sampling weights may be incorporated into the sample correlation coefficient by replacing the conventional covariance

$$\widehat{\sigma}_{xy} = \frac{\sum_i (x_i - \overline{x})(y_i - \overline{y})}{n}$$

with the weighted sum

$$\frac{\sum_i w_i (x_i - \overline{x})(y_i - \overline{y})}{\sum_i w_i}.$$

The variance estimates should also be weighted sums, say,

$$\widehat{\sigma}_x^2 = \frac{\sum_i w_i (x_i - \overline{x})^2}{\sum_i w_i}.$$

Then, the weighted sample correlation coefficient is

$$r = \frac{\sum_i w_i (x_i - \overline{x})(y_i - \overline{y})}{\widehat{\sigma}_x \widehat{\sigma}_y \sum_i w_i}.$$

Show that

$$r = \frac{\sum_i v_i x_i y_i - \overline{xy}}{\widehat{\sigma}_x \widehat{\sigma}_y}.$$

where $v_i = w_i / \sum_j w_j$ and $\overline{x}$ and $\overline{y}$ are also weighted according the sampling weights $w_1, \ldots, w_n$.

7.8.2 Computational

7.5. The function `dist` (instruction 3 of Sect. 7.7) can be replaced by one line of code. Do so.

7.6. For each of the state-by-state linear regressions of estimated annual prevalence on centered year, compute the adjusted coefficient of determination

$$R^2_{\text{adjusted}} = \frac{s^2 - \widehat{\sigma}^2_{\text{reg}}}{s^2}$$

where

$$
\begin{aligned}
s^2 &= \frac{\sum_{i=1}^n (y_i - \overline{y})^2}{n-1} \\
&= \frac{\mathbf{y}^T\mathbf{y} - n\overline{y}^2}{n-1}
\end{aligned}
$$

is the sample variance and

$$
\begin{aligned}
\widehat{\sigma}^2_{\text{reg}} &= \frac{\sum_{i=1}^n (y_i - \widehat{y}_i)^2}{n-2} \\
&= \frac{\mathbf{y}^T\mathbf{y} - \mathbf{y}^T\mathbf{X}\widehat{\boldsymbol{\beta}}}{n-2}
\end{aligned}
$$

is the residual variance. Unlike previous large sample applications of R^2_{adjusted}, we must be precise with the denominators. Construct a dot chart showing the R^2_{adjusted} and the state code. You may use the R function call

```
dotchart(D$R2,D$State,cex=.5)
```

after having read the output into an R data frame and assigning the column names State and R2 to the respective columns.

7.7. Estimate diabetes risk using age, income, body mass index, and exercise level (EXERANY2 in the BRFSS codebook). Compute sensitivity and specificity without making any adjustments for small sample sizes. Report the values for $p \in \{.05, \ldots, .2\}$

7.8. The estimated incidence for District of Columbia is half as large as any other state though its estimated prevalence is not far from the median value. Plot the annual estimates of prevalence against year for the District of Columbia and Virginia, a state that is spatially near to the District of Columbia and has a similar estimated prevalence. Use ggplot. Note that lists of year and prevalence estimates stored in statePrevDict are not in sequential order.

7.9. Return to the tutorial of Sect. 7.5 and replace age with ethnicity. Use the BRFSS variable _RACE for which ethnicity is grouped as 8 classes. How many unique profile vectors are produced using each set of predictor vectors? Construct a table that compares sensitivity and specificity at three thresholds: .05, .15, and .30. Is there a difference between the two prediction functions with respect to sensitivity and specificity, and if so, which prediction function is best?

Chapter 8
Cluster Analysis

Abstract Sometimes it's possible to divide a collection of observations into distinct subgroups based on nothing more than the observation attributes. If this can be done, then understanding the population or process generating the observations becomes easier. The intent of cluster analysis is to carry out a division of a data set into *clusters* of observations that are more alike within cluster than between clusters. Clusters are formed either by aggregating observations or dividing a single glob of observations into a collection of smaller sets. The process of cluster formation involves two varieties of algorithms. The first shuffles observations between a fixed number of clusters to maximize within-cluster similarity. The second process begins with singleton clusters and recursively merges the clusters. Alternatively, we may begin with one cluster and recursively split off new clusters. In this chapter, we discuss two popular cluster analysis algorithms (and representatives of the two varieties of algorithms): the k-means algorithm and hierarchical agglomerative clustering.

8.1 Introduction

Cluster analysis is a collection of methods for the task of forming groups where none exist. For example, we may ask whether there are distinct types of visitors to a grocery store, say, customers that purchase a few items infrequently, customers that regularly shop at a particular department, and customers that make frequent visits and purchase a wide variety of items. If so, then the visitors of the first and second group might be offered incentives aimed at shifting them to the third group. Realistically though, it's probably not possible to cleanly divide customers given that many customers

B. Steele et al., *Algorithms for Data Science*,
DOI 10.1007/978-3-319-45797-0_8

exhibit multiple shopping behaviors. With a set of observations on customers and records of their purchases (receipts), cluster analysis may provide insight into the structure of the customer population and shopping behaviors.[1]

In this situation, the analyst does not have a set of data that is labeled according to group. The process of forming groups is therefore without benefit of supervision originating from a training set of labeled observations. Without knowledge of the number and arrangement of the groups, cluster formation proceeds by grouping observations that are most alike or by splitting groups based on the dissimilarity of the member observations. Similarity of observations and clusters is measured on the observation attributes. The only impetus driving cluster formation is similarity and dissimilarity.

Cluster analysis is driven by the mathematical objective of maximizing the similarity of observations within cluster. This objective does not provide much guidance on how to proceed. In contrast, linear regression is strongly driven by the linear model and the objective of minimizing the sum of squared differences between observations and fitted values. The mathematics and algorithms fall neatly into place from this starting point. Clustering algorithms on the other hand, have different approaches, all meritorious and occasionally useful. Even determining the appropriate number of clusters is difficult without prior knowledge of the population sub-divisions. Despite these weaknesses, cluster analysis is still a useful tool of data analytics since the ability to examine groups of similar observations often sheds light on the population or process generating the data.

In this chapter, we sidestep the difficult issues related to the application and interpretation of cluster analysis and focus instead on the mechanics of two somewhat different but fundamental algorithms of cluster analysis: hierarchical agglomerative and k-means clustering. By understanding these basic algorithms, we learn of the strengths and weaknesses of cluster analysis.

Let us begin by defining $D = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ to be the data set. As before, $\mathbf{x}_i$ is a vector of attributes measured on an observational unit. Unlike the data sets of Chap. 6, there is no target y_i to be predicted.

We begin the discussion of cluster analysis with a popular technique that recursively builds clusters by merging smaller clusters.

8.2 Hierarchical Agglomerative Clustering

The hierarchical agglomerative approach begins with each observation defining a singleton cluster. Therefore, the initial set of clusters may be represented by the singleton clusters $\{\mathbf{x}_1\}, \ldots, \{\mathbf{x}_n\}$, where n is the number of observations. The algorithm iteratively reduces the set of clusters by merging

[1] Section 10.6, Chap. 10 works with data originating from grocery store receipts.

similar clusters. On the ith iteration, two clusters A and B, say, are merged to form a cluster $A \cup B$. We write

$$(A, B) \longrightarrow A \cup B$$

to describe the merging of clusters A and B. The choice of clusters to merge is determined by computing a distance between clusters and merging the pair with the minimum inter-cluster distance. Consequently, a metric is needed to measure between-cluster distances. For example, the distance between clusters A and B may be defined to be the smallest distance between any vector belonging A and any vector belonging to B. Mathematically, this distance is defined as

$$d_1(A, B) = \min\{d_C(\mathbf{x}_k, \mathbf{x}_l)|\mathbf{x}_k \in A, \mathbf{x}_l \in B\},$$

where $d_C(\mathbf{x}, \mathbf{y})$ is the city-block distance between vectors $\mathbf{x}$ and $\mathbf{y}$ defined by Eq. (7.12). There's nothing special about city-block distance—other metrics, Euclidean distance, for example, also are popular. The minimum distance metric d_1 tends to produce chain-like clusters. More compact clusters result from a centroid-based metric that utilizes cluster centroids defined by

$$\overline{\mathbf{x}}_A = n_A^{-1} \sum_{\mathbf{x}_k \in A} \mathbf{x}_k, \tag{8.1}$$

where n_A is the numbers of observations in cluster A. Then, the distance between A and B is

$$d_{\text{ave}}(A, B) = d_C(\overline{\mathbf{x}}_A, \overline{\mathbf{x}}_B). \tag{8.2}$$

As described above, the algorithm will progressively merge clusters until there is a single cluster. Merging clusters into a single glob is only interesting if some of the intermediate cluster sets are interesting. If the clusters are to be used for some purpose, then it's necessary to inspect the intermediate sets of clusters to identify the most useful cluster set.

8.3 Comparison of States

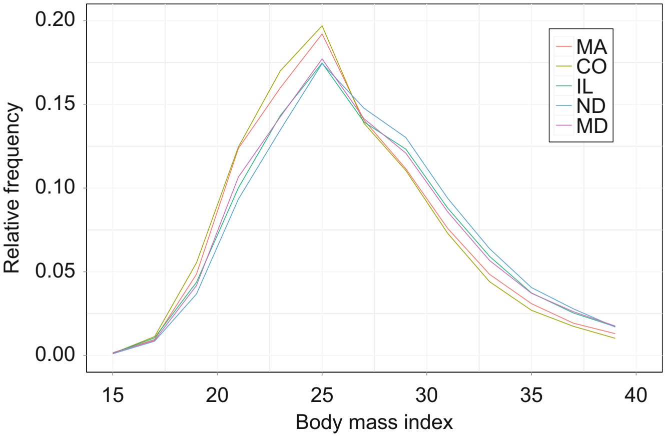
We return to the BRFSS data and search for states that are alike with respect to the distribution of body mass index of residents. The distribution of body mass index for a particular state is represented by a histogram. A histogram, mathematically, is a set of intervals spanning the range of body mass index and the associated sample proportions of individuals belonging to each interval.[2] Each histogram amounts to an *empirical* distribution—that is, a distribution that has been constructed from data rather than a model.

[2] We worked with the mathematical form of the histogram in Chap. 3, Sect. 3.4.2.

As usual, data preparation requires a significant effort since we need to construct a histogram for each state, Puerto Rico, and the District of Columbia. Before delving into the computational problem, let us examine the empirical body mass index distributions for a handful of states.

Figure 8.1 shows the empirical distributions of body mass index for five states: Massachusetts, Colorado, Illinois, North Dakota, and Maryland.[3] The distributions from Massachusetts and Colorado are alike as are the distributions from Illinois, North Dakota, and Maryland. Massachusetts and Colorado are different from Illinois, North Dakota, and Maryland, principally because the Massachusetts and Colorado distributions tend to have relatively more values that are less than 27 kg/m^2 than Illinois, North Dakota, and Maryland, and fewer values that are greater than 27 kg/m^2. Interestingly, Colorado and Massachusetts previously were found to have the second and tenth smallest estimates of diabetes prevalence (Fig. 7.1).

Fig. 8.1 Empirical body mass index distributions for five states. The scale has been truncated on the right so that differences among distributions may be more easily discerned



Recall that the numerical form of a conventional histogram is a set of pairs $H = \{(b_1, p_1), \ldots, (b_h, p_h)\}$, where $b_i = (l_i, u_i]$ is a bin or interval and p_i is the proportion of observations included in the interval.[4] If the data are a representative sample from the population, then H is constructed by defining a set of intervals and counting the number of sample observations falling into each interval. The BRFSS sampling design does not generate representative samples since observations are collected with unequal sampling probabilities. To correct for unequal sampling probabilities, the Centers of Disease Control and Prevention has attached a sampling weight to each observation. Usually, the sampling weight can be incorporated into an estimator with the effect of reducing or eliminating bias created by unequal sampling probabilities.

[3] The data shown in this figure may be plotted as a set of histograms. However, we use a simple line plot instead as it's easier to see the similarities among empirical distributions.

[4] Section 3.4.2 of Chap. 3 discusses histograms in details.

Section 3.4.2 of Chap. 3 presented an algorithm for constructing histograms from sampling weights. The adaption replaced the relative frequency of observations included in a particular interval with the sum of sampling weights associated with the observations.

Let x_j denote a measurement on the variable of interest (in this case, body mass index) for the jth observation (in this case, a respondent). Let w_j denote the sampling weight assigned to the observation. Let's suppose that $x_j, j = 1, 2, \ldots, n$ are from state A. Then, the total sampling weight associated with values belonging to interval b_i and originating from state A can be expressed as

$$s_{A,i} = \sum_{j=1}^{n} w_j I_i(x_j) \tag{8.3}$$

where $I_i(x_j)$ is an indicator variable taking on the value 1 if $x_j \in b_i$ is true (formula (3.10)).[5] If $x_j \in b_i$ is false, then the value of the indicator variable is 0. Transforming the sum $s_{A,i}$ to the height of the histogram bar is straightforward: we compute the estimated proportion of the population belonging to interval b_i as

$$p_{A,i} = \frac{s_{A,i}}{\sum_{k=1}^{h} s_{A,k}}. \tag{8.4}$$

Formula (8.4) was encountered in Chap. 3, formula (3.12). As with a conventional histogram, we iterate over the observations and accumulate the sums $s_{A,1}, \ldots, s_{A,h}$, and then form the histogram for state A as

$$H_A = \{(b_1, s_{A,1}), \ldots, (b_h, s_{A,h})\}, \tag{8.5}$$

A dictionary is maintained to keep track of the clusters as the algorithm progresses. A dictionary key is the cluster label A, and the value is the list of sampling weights $[s_{A,1}, \ldots, s_{A,h}]$. It's not necessary to store the intervals (the b_i's) with each histogram since the intervals are the same for every histogram.

When the cluster formation algorithm begins, the cluster singletons are individual states and the cluster histograms are the histograms for each state. The distance between clusters A and B is measured by the distance between the histograms associated with the clusters. This distance is defined to be

$$d_c(A, B) = \textstyle\sum_{k=1}^{h} |p_{A,k} - p_{B,k}|,$$

where $p_{A,k}$ and $p_{B,k}$ are computed according to Eq. (8.4).

We also need a method for merging the histograms resulting from the merger of clusters A and B. The estimated proportion for interval k should not be a simple average of the interval proportions $p_{A,k}$ and $p_{B,k}$ since the average does not account for the differences in numbers of observations that are contained in clusters A and B. So instead, we compute a weighted average

[5] The statement $x_j \in b_i$ is true if $l_i < x_j \leq u_i$.

that utilizes the numbers of observations n_A and n_B in the respective clusters. The weighted mean is

$$\overline{p}_k = \frac{n_A p_{A,k} + n_B p_{B,k}}{n_A + n_B}.$$

The weights are the proportions $n_A/(n_A + n_B)$ and $n_B/(n_A + n_B)$.

In summary, the hierarchical clustering algorithm can be viewed as a sequence of mappings where a mapping reduces two clusters A and B to one cluster according to

$$\left.\begin{array}{l}[p_{A,1}, \ldots, p_{A,h}] \\ [p_{B,1}, \ldots, p_{B,h}]\end{array}\right\} \longrightarrow [\overline{p}_1, \ldots, \overline{p}_h]\,. \tag{8.6}$$

We may also view the mapping as $(H_A, H_B) \longrightarrow H_A$. Instead of creating a new label $A \cup B$, we assign the label A to $A \cup B$ and delete B from the list of cluster labels. Metaphorically, A has devoured B.

Operationally, merging two clusters requires a search for the most similar pair of clusters. Once the pair (A, B) is identified, cluster A absorbs B by combining the two histograms as one according to the map (8.6). The combined histogram, replaces A. Cluster B is removed from the list of clusters.

The search for most similar clusters requires the inter-cluster distance between every cluster A and B. If the number of observations were large, we would not recompute all inter-cluster distances for every search, but instead maintain a list of inter-cluster distances and only update the distances between the merged cluster and all other clusters. For simplicity, the tutorial below recomputes all inter-cluster distances whenever a pair of clusters is merged.

8.4 Tutorial: Hierarchical Clustering of States

We'll write an algorithm for hierarchical agglomerative cluster formation. A substantial amount of data reduction is necessary before cluster formation can commence. The data reduction stage will map a set of BRFSS annual files to a dictionary in which each key is a state and the value is a list of the sampling weights $s_{A,1}, \ldots, s_{A,h}$ shown in Eq. (8.3). Most of the data reduction has been carried out in one form or another in previous tutorials. Each state in the dictionary will represent one of the initial singleton clusters. The cluster formation algorithm begins with this set of initial clusters. The algorithm iterates over a list of clusters and on each iteration, two clusters are merged as one. When two clusters are merged, the dictionary is updated by replacing one cluster with the merged cluster. The second cluster is removed from the dictionary.

1. At the top of your script, import modules to be used by the program:

```
import os
import importlib
import sys
sys.path.append('/home/.../parent')
from PythonScripts import functions
dir(functions)
```

2. Use the function `stateCodeBuild` developed in instruction 4 of Sect. 7.3, Chap. 7 to create a dictionary of state names and codes. We use the term state loosely as the dictionary will contain entries for the District of Columbia and Puerto Rico. The dictionary `stateCodeDict` uses the FIPS two-digit state codes as keys. The values are standard United States Postal Service two-letter state abbreviations. Call the function and create the dictionary. Also create a list (`namesList`) containing the two-digit abbreviations:

```
stateCodeDict = stateCodeBuild()
namesList = list(stateCodeDict.values())
noDataSet = set(namesList)
```

   The Behavioral Risk Factor Surveillance System collects data from three U.S. territories as well as the states and the District of Columbia. However, `stateCodeDict` contains the names of 57 states, territories, and the District of Columbia. Not all of these geographic units are sampled by the BRFSS survey. We'll identify the geographic units that have no data by removing names from the set `noDataSet` whenever a record is encountered from a particular geographic unit. What's left in `noDataSet` are geographic units with no data.

3. Create a dictionary containing the field positions of body mass index and sampling weight using the function `fieldDictBuild`. It was created in instruction 4 of Sect. 7.3, Chap. 7.

```
fieldDict = functions.fieldDictBuild()
print(fieldDict)
```

4. Create a dictionary of state histograms. Each histogram consists of a set of 30 sub-intervals spanning the interval $(12, 72]$ (kg/m^2) and an associated set of relative frequency measures. The histogram dictionary `histDict` maintains the histogram data for cluster A as a list containing the sampling weight sums $s_{A,1}, \ldots, s_{A,h}$ (Eq. (8.3)). The set of intervals is the same for each cluster and is stored as a single list of pairs named `intervals`.

```
nIntervals = 30
intervals = [(12+2*i,12+2*(i+1)) for i in range(nIntervals)  ]
histDict = {name:[0]*nIntervals for name in namesList}
```

Dictionary comprehension has been used to create `histDict`. The value associated with a `name` is a list of length 30 containing zeros.

5. We will use the same structure for processing a set of BRFSS files as was used previously, say instruction 9 of Chap. 7, Sect. 7.3.

```
n = 0
dataDict = {}
path = r'../Data/'  # Replace with your path.
fileList = os.listdir(path)
for filename in fileList:
    try:
        shortYear = int(filename[6:8])
        year = 2000 + shortYear
        fields = fieldDict[shortYear]
        sWt, eWt = fields['weight']
        sBMI, eBMI = fields['bmi']
        file = path + filename
    except(ValueError):
        pass
    print(n)
```

A `ValueError` will be thrown if `filename[6:8]` does not contain a string of digits. In that case, the remaining statements in the `try` branch of the exception handler are ignored and interpreter executes the `pass` statement.

6. Open each file in the `fileList` within the `try` branch of the exception handler and process the file records one at a time. Determine the respondent's state of residence and extract the state name from the `stateCodeDict`:

```
with open(file, encoding="utf-8", errors='ignore') as f:
    for record in f:
        stateCode = int(record[:2])
        stateName = stateCodeDict[stateCode]
```

It's difficult to resolve errors when an exception handler is invoked as errors are often not visible. Thus, it may be helpful to execute the code segment beginning with `with open(file,...` as you develop the code rather than allowing the exception handler to be invoked. If you proceed this way, you may find it helpful to temporarily set `file` to be one of the BRFSS data files in the list `fileList`.

7. Extract the sampling weight `weight` from `record` (see Sect. 7.3).
8. Translate the body mass index string to a float using the function `convertBMI`:

```
bmiString = record[sBMI-1:eBMI]
bmi = functions.convertBMI(bmiString,shortYear)
```

   The function `convertBMI` was build in the Chap. 3, Sect. 3.6 tutorial (see instruction 14). We're assuming that it resides in your functions module `functions.py`.
9. Increment the sampling weight total for the interval containing `bmi` in the histogram `histDict[stateName]`. This histogram corresponds to the respondent's state of residence.

```
for i, interval in enumerate(intervals):
    if interval[0] < bmi <= interval[1]:
        histDict[stateName][i] += weight
        break
noDataSet = noDataSet - set([stateName])
n += 1
```

   The next-to-last instruction computes the set difference between `noDataSet` and the singleton data set containing the respondent's state of residence. Allow the program to process all of the data sets.
10. Upon completion of the `for` loop iterating over `fileList`, remove the key-value pairs from `histDict` for which there are no data.

```
print(len(histDict))
print(noDataSet)
for stateName in noDataSet:
    del histDict[stateName]
print(len(histDict))
```

11. Transform the sums of weights contained `histDict` so that the sum over the intervals of a histogram is 1.

```
for state in histDict:
    sumWeights = sum(histDict[state])
    histDict[state] = [intervalTotal/sumWeights
                        for intervalTotal in histDict[state]]
```

12. We'll use the data in its reduced form of `histDict` in the next tutorial. It's convenient then to save the dictionary in a file so that `histDict` need

not be rebuilt. A standard method of saving data structures is the `Python` module `pickle`. Write `histDict` as a pickle file:

```
import pickle
picklePath = '../data/histDict.pkl'
pickle.dump(histDict, open(picklePath, "wb" ))
```

The file `histDict.pkl` will be read from disk into a `Python` dictionary at the beginning of the tutorial on the k-means algorithm.

13. Form an initial dictionary of clusters. Each cluster is a key-value pair where the key is a cluster label and a state abbreviation. The value is a singleton list containing the state abbreviation. For example, a dictionary entry will appear as $\{a : [a]\}$.

```
clusterDict = {state: [state] for state in histDict.keys()}
```

When two clusters A and B are merged, the list of states that belong to the cluster A will lengthen by adding those belonging to B. Cluster B will be deleted from the dictionary and the number of key-value pairs in `clusterDict` will be reduced by 1.

14. Begin building a function that merges clusters. It's best not to make it a function (with the keyword `def` and a return statement) until the code segment is complete and free of errors. Create a list of the state codes by extracting the keys of `clusterDict`.

```
stateList = list(clusterDict.keys())
```

15. We'll build the code segment that merges the closest two clusters. Three operations are necessary: build a list of all two-cluster sets, search the list for the closest two clusters, and merge the closest two as one cluster. Building the set of cluster pairs is accomplished using list comprehension:

```
setList = [{a,b} for i,a in enumerate(stateList[:-1])
                    for b in stateList[i+1:]]
```

The outer `for` loop iterates over all elements `stateList` except the last. The inner `for` loop iterates over the elements `stateList[i]` through the last (`stateList[n-1]`).

16. Begin the search for the closest pair by initializing the minimum distance between cluster to be 2.[6] Iterate over `setList`. With clusters A

[6] It can be proved that the distance between any two clusters will be less than 2.

and B, compute the distance between the associated histograms and if the distance is less than the minimum distance, update the minimum distance and save the set with the name `closestSet`. When the `for` loop is complete, `closestSet` will contain the labels of the clusters to merge.

```
mn = 2
for a, b in setList:
    abD = sum([abs(pai - pbi)
                 for pai, pbi in zip(histDict[a], histDict[b])])
    if abD < mn:
        mn = abD
        closestSet = {a,b}
    print(closestSet,mn)
```

Our single-line function for computing the distance `abD` between clusters A and B takes the relative proportions from `histDict` and forms pairs $(p_{A,1}, p_{B,1}), \ldots, (p_{A,h}, p_{B,h})$. The `zip` function performs this operation. Then, the function iterates over the `zip` object and builds a list of the absolute differences, i.e., $[|p_{A,1} - p_{B,1}|, \ldots, |p_{A,h} - p_{B,h}|]$ using list comprehension. The last operation computes the sum of the absolute differences.

17. Let A and B denote the closest two clusters. They are merged by replacing the relative proportions for cluster A with the weighted average of the relative proportions from A and B. Hence, A consumes B. In this code segment, the relative proportions for cluster A are updated.

```
a, b = closestSet
na = len(clusterDict[a])
nb = len(clusterDict[b])
histDict[a] = [(na*pai + nb*pbi)/(na + nb) for pai, pbi
                           in zip(histDict[a], histDict[b])]
```

18. Extend the member list of A with the member list of B. Remove B from the cluster dictionary.

```
clusterDict[a].extend(clusterDict[b])
del clusterDict[b]
print(len(clusterDict))
```

You can test the merging operation by repeatedly running the code beginning with the instruction `stateList = list(histDict.keys())` and ending with `print(len(clusterDict))`. On each execution, the length of `clusterDict` will decrement by 1.

19. When the merging operation functions correctly, move it to a function `mergeClusters` that accepts `clusterDict` and `histDict` as arguments and returns `clusterDict` and `histDict`, say,

```
def mergeClusters(clusterDict, histDict):
    stateList = list(clusterDict.keys())
    ...
    del clusterDict[b]
    return clusterDict, histDict
```

20. Reduce the initial set of 54 singleton clusters to 5 clusters using a `for` loop that repeatedly calls `mergeClusters`.

```
while len(clusterDict) > 5:
    clusterDict, histDict = mergeClusters(clusterDict, histDict)
```

21. Print the clusters and the member states to the console.

```
for k,v in clusterDict.items():
    print(k,v)
```

22. We'll use `pyplot` from the `matplotlib` to draw the cluster histograms on a single figure. We've built almost the same plot in Chap. 3, instruction 26. To improve the readability, the histograms are truncated at the body mass index value of 51 $\mathrm{kg/m^2}$.

```
intervals = [(12+2*i,12+2*(i+1)) for i in 30  ]

import matplotlib.pyplot as plt
x = [np.mean(pair) for pair in intervals][:19] # Ignore large BMI
      values.
for name in clusterDict:
    y = histDict[name][:19]
    plt.plot(x, y)
plt.legend([str(label) for label in range(6)], loc='upper right')
plt.show()
```

8.4.1 Synopsis

We have used a hierarchical clustering algorithm to form clusters of states that are similar with respect to the body mass index of adult residents.

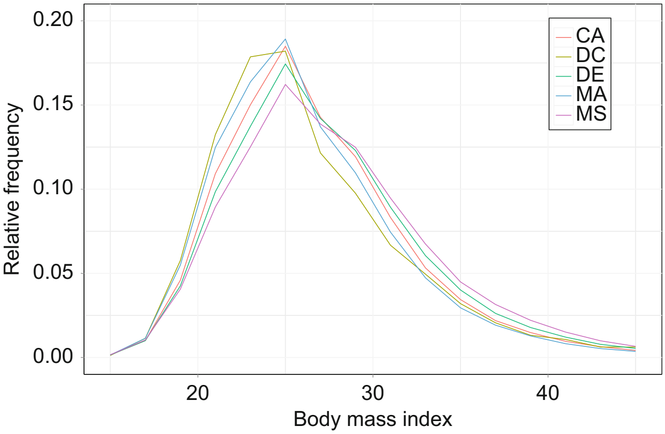


Fig. 8.2 Estimated distributions of body mass index for five clusters of states. Clusters are identified by state abbreviation

The analysis began with the estimation of body mass index distributions for each state. State to state and cluster to cluster similarity was measured by the distance between relative frequency histograms. When two clusters were merged, the merged cluster histogram was computed as a weighted average of the histograms belonging to the merged clusters. The weights reflected the numbers of states belonging to the respective clusters.

This application of cluster analysis corresponds to a more traditional set-up in which each observational unit would have provided a vector of observations on $h = 30$ variables. In our application, an observational unit is a state. The similarity calculation would have used the h variables. Some transformation, say standardization for instance, probably would have been used to try to account for differences in scale among the variables. Scaling was not necessary in this application.

Figure 8.2 shows that the cluster distributions are different though perhaps not to a striking extent. The difference in height between histograms for a fixed value of the horizontally-plotted variable (body mass index in this case) is the primary attribute reflecting differences in distributions. With this in mind, it's clear that the District of Columbia (DC) is very different than the Mississippi cluster (MS). For instance, all of the histograms have a mode at 25 kg/m^2 and the relative frequencies for the interval $[24, 26)$ are .182 and .162 for DC and MS respectively. The differences reverse in sign at the body mass index value 26 kg/m^2, reflecting a greater proportion of individuals with large values of body mass index in the Mississippi cluster than in the District of Columbia. The District of Columbia is a singleton cluster and so informally speaking, there are no states similar to it.

Though the differences are small for any single interval, there are 30 intervals, and the cumulative difference are large. In fact, 30.7% of the individuals in the Mississippi cluster have a body mass index at least 30 kg/m^2 and hence are classified as obese whereas only 21.7% of the District of Columbia sample is obese.

It's often difficult to judge whether a cluster analysis has successfully created clusters that are meaningfully different. In this example, we are able to do so by recalling that the motivation for analyzing body mass index is the prevalent view in public health circles that body mass index is associated with a number of chronic diseases (type 2 diabetes in particular). We may attempt to confirm that position by returning to the analysis of diabetes prevalence and incidence carried out in Chap. 7. In particular, we reconstructed Fig. 7.1 as Fig. 8.3 and identified cluster membership of states by color. States belonging to a particular cluster are usually near each other. Only the green cluster is not in a well-defined region.

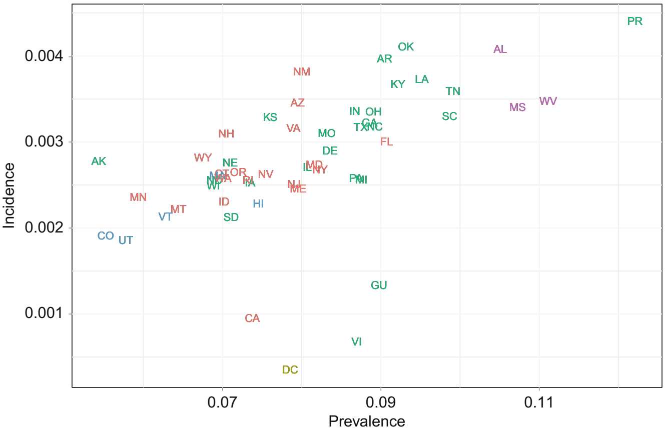


Fig. 8.3 Estimated incidence and prevalence of diabetes. States have been colored according to their membership in one of five clusters

8.5 The *k*-Means Algorithm

Our example of representative-based cluster analysis is the k-means algorithm. Now the number of clusters is determined by the analyst. The algorithm begins by randomly assigning the observation vectors in the data set $D = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ to k clusters. This initial set of clusters is denoted as $\{A_1, \ldots, A_k\}$. Then, the *centroids* are computed for each cluster. Traditionally, the centroids are the multivariate means of the observations belonging to the cluster. The centroids represent the cluster in the calculation of distance from observation to cluster.

The initial configuration is rarely satisfactory and so the bulk of computational effort is aimed at improving on it. The algorithm iterates between two steps: assign every observation to the nearest cluster, then recompute, or update the cluster centroids. If any observation is reassigned, that is, moved out of its currently assigned cluster and into another, then the centroids of the two clusters will change. Therefore, another iteration should take place.

The algorithm continues iterating between the two steps until no further reassignments are made. At that point, each observation belongs to the cluster to which it is closest. Starting from the initial random configuration, the within cluster sums-of-squares has been minimized. This is because the within sums-of-squares is equivalent to the sum of the Euclidean distances between observations and cluster centroids, and because moving any observation now will increase the sum of Euclidean distances (and the within sums-of-squares). It follows that the algorithm has reached a best possible assignment of observations to clusters and a best possible calculation of centroids.

The algorithm has considerable appeal because a popular objective function has been minimized by the algorithm (the within sums-of-squares). The drawback of the algorithm is that the initial configuration is random. A different configuration will often lead to a different set of clusters. If the analyst can begin with a selectively chosen initial configuration, then the algorithm may produce more satisfactory clusters.

Let's look at the details. The centroid of cluster A_i is the multivariate mean

$$\overline{\mathbf{x}}_i = n_i^{-1} \sum_{\mathbf{x}_j \in A_i} \mathbf{x}_j, \tag{8.7}$$

where n_i is the number of observations belonging to A_i and $\mathbf{x}_j{=}[x_{j,1} \cdots x_{j,h}]^T$. The number of attributes is h and the lth element of $\overline{\mathbf{x}}_i$ is

$$\overline{x}_{i,l} = n_i^{-1} \sum_{\mathbf{x}_j \in A_i} x_{j,l}, \tag{8.8}$$

for $l = 1, \ldots, h$. The distance between $\mathbf{x}_j \in D$ and A_i is defined to be the distance between $\mathbf{x}_j$ and the centroid of A_i, $\overline{\mathbf{x}}_i$. Since $\overline{\mathbf{x}}_i = [\overline{x}_{i,1} \ \cdots \ \overline{x}_{i,h}]^T$, the squared Euclidean distance is

$$d_E^2(\mathbf{x}_j, \overline{\mathbf{x}}_i) = \sum_{l=1}^{h} (x_{j,l} - \overline{x}_{i,l})^2. \tag{8.9}$$

We may use the squared Euclidean distance in the determination of the nearest centroid to $\mathbf{x}_j$ instead of Euclidean distance since the ordering, nearest to most distant, will be the same.

Upon completion of the initialization phase, the algorithm begins iterating between two steps.

The first step iterates over the data set D and computes the squared Euclidean distances between each $\mathbf{x}_j \in D$ and each cluster centroid according to Eq. (8.9). If an observation is found to be nearest to a different cluster than its currently assigned cluster, then it is reassigned to the nearest cluster. The second step updates the centroid of cluster A_i according Eq. (8.7). Every cluster that has changed membership in the last iteration must have an update computed to its centroid. That completes the two steps. If any observation has been reassigned, then another iteration commences. The algorithm terminates when no further changes in membership occur.

8.6 Tutorial: The k-Means Algorithm

We'll continue with the exercise of grouping states with respect to body mass index distribution of adult residents. The same data set of state body mass index histograms is used and so this tutorial begins not with data preparation but with programming the k-means algorithm. It's assumed that histogram representations of the body mass index have been computed in the course of the previous tutorial and are stored in the dictionary `histDict`. The keys of the dictionary are states[7] and the values are lists containing the estimated proportion of adult state residents with body mass indexes in the $h = 30$ intervals $(12, 14], (14, 16], \ldots, (70, 72]$.

We choose to create $k = 6$ clusters. Each observation is a state, as before. The contents of the observation vector are the same but we're changing the notation to be consistent with the development of the k-means algorithm. For the jth state, the estimated proportion of individuals in interval l is $x_{j,l}, l = 1, 2, \ldots, h$.[8] The vector of estimated proportions associated with observation j is now denoted generically by

$$\mathbf{x}_j = [x_{j,1} \ \cdots \ x_{j,h}]^T. \tag{8.10}$$

For cluster A_i, the centroid is $\overline{\mathbf{x}}_i = [\overline{x}_{i,1} \ \cdots \ \overline{x}_{i,h}]^T$. The centroid is computed using formula (8.7).

A note on programming the algorithm: the k-means algorithm is very fast and so it may not an advantage to write computationally efficient code if it requires a significant effort beyond writing simple and relatively slow code. For example, the first step of every iteration updates the cluster centroids. There are two ways to update the cluster centroids: recompute every centroid, or update the centroids that need to be updated by computing the centroid sum, subtracting from the sum the observation vectors that have been removed from the cluster and adding to the sum the observation vectors that have been added to the cluster. Then, divide by the number of observations in the cluster. Which is best? It depends on the use of the algorithm. Saving a few seconds of computation time is not worth an hour of programming time. We will recompute all of the centroids.

The program consists of two primary blocks: reading the data and initializing the clusters, and the k-means algorithm.

1. Load your pickle file containing the dictionary of state body mass index histograms and store the contents of the file in a dictionary with the name `histDict`.[9]

[7] Recall from the tutorial of Sect. 8.4 that there are actually 54 geographic entities that we are loosely referring to as state.

[8] The previous notation for the estimated proportion of individuals in interval l and observation j, was $p_{j,l}$.

[9] The pickle file was created in instruction 12 of the tutorial of Sect. 8.4.

```
import numpy as np
import pickle
picklePath = '../histDict.pkl'
histDict = pickle.load( open(picklePath, "rb" ) )
print(len(histDict))
```

2. Randomize the list of states so that the initial assignment of state to cluster will be random. Use the `Numpy` function `random.choice`. The essential arguments to be passed are a list from which to sample from (we pass the keys of `histDict`), the number of units to sample (`size = n`), and the type of sampling. Sampling must be *without* replacement so that a state does not appear in more than one of the k lists of cluster members.

```
k = 6
n = len(histDict)
randomizedNames = np.random.choice(list(histDict.keys()), size = n,
                          replace = False)
```

   The call to `randomize` returns a random sample of size n without replacement. The effect of this code segment is to randomly shuffle the order of the states in `randomizedNames`.

3. Build a dictionary `clusterDict` that contains the assignments of state to initial cluster. Use a `for` loop. The keys are computed as $i \bmod k$ where $k = 6$ is the number of clusters and $i \in \{0, 1, \ldots, n-1\}$. The dictionary values are lists of states.

```
clusterDict = {}
for i, state in enumerate(randomizedNames):
    clusterDict.setdefault(i%k,[]).append(state)
print(clusterDict)
```

   The `setdefault` function was discussed in Sect. 4.6.2 and used in instruction 27 of Sect. 4.6.2. The `enumerate` function was described in instruction 32 of Sect. 4.6.2.

4. Write a function that will recompute a histogram for each cluster in the cluster dictionary `clusterDict`. The histogram for each cluster is computed by iterating over each state in the cluster and adding up the estimated proportions in each histogram interval (Eq. (8.8)). We begin creating a dictionary `clusterHistDict` to store the k cluster histograms. Then, a `for` loop iterates over clusters and a second inner `for` loop iterates over states belonging to clusters. The structure is

```
h = 30
clusterHistDict = dict.fromkeys(clusterDict,[0]*h)
for a in clusterDict:
    sumList = [0]*h
    for state in clusterDict[a]:
        print(a,state)
```

The list sumList will store the sum of estimated proportions for each interval as the program iterates over members of the cluster.

5. Terms are added to sumList by zipping sumList with the state histogram histDict[state]. Then, we use list comprehension to update sumList with the data stored in histDict[state]. The last step is to divide the sums by the number of states belonging to the cluster and store the list in the cluster histogram dictionary.

```
for state in clusterDict[a]:
    sumList = [(sumi + pmi) for sumi, pmi in
                  zip(sumList, histDict[state])]
na = len(clusterDict[a])
clusterHistDict[a] = [sumi/na for sumi in sumList]
```

   This code segment executes *within* the for loop that iterates over clusterDict and replaces the last two lines of instruction 4.

6. Turn the code that updates the cluster centroids stored in clusterHistDict into a function:

```
def clusterHistBuild(clusterDict, histDict):
    ...
    return clusterHistDict
```

7. Move the function to the top of the script. Build the initial cluster centroids by calling the function

```
clusterHistDict = clusterHistBuild(clusterDict,histDict)
```

   This instruction immediately follows the construction of clusterDict (instruction 3).

8. The next code segment is the start of the k-means algorithm iteration phase. There are two steps to be executed. The first step reassigns an observation (a state) to a different cluster if the state is closest to some other cluster besides the cluster that it is currently assigned to. The sec-

ond step updates the centroids stored in `clusterHistDict` if any states have been reassigned. These steps are repeated until no state is reassigned to a different cluster

A conditional structure using the `while` statement repeatedly executes the code within the structure. Program flow breaks out of the structure when the boolean variable `repeat` is false. The structure is so:

```
repeat = True
while repeat == True:
    updateDict = {}
    # Step 1:
    ...
    # Step 2:
    if clusterDict != updateDict:
         clusterDict = updateDict.copy()
         clusterHistDict = clusterHistBuild(clusterDict,histDict)
    else:
        repeat = False
```

The dictionary `updateDict` contains the updated clusters. It has the same structure as `clusterDict` and so there are k key-value pairs. A key is a cluster index ($i = 1, 2, \dots, k$) and the value is a list of member states. It's built by determining the nearest cluster to each state and assigning the state to the nearest cluster.

If `clusterDict` is not equal to `updateDict` then at least one state has been reassigned to a different cluster. If this is the case, then we update the cluster dictionary `clusterDict` and the dictionary of cluster histograms `clusterHistDict`. If `clusterDict` is equal to `updateDict`, then the algorithm is complete and program flow is directed to the next statement following the conditional structure.

We must use the `.copy()` function when assigning the contents of `updateDict` to `clusterDict`. If we set `clusterDict = updateDict`, then the two objects will henceforth use same memory address. Any change to one dictionary immediately makes the same change to the other. Using `.copy()` writes the values of `updateDict` into the memory location of `clusterDict`. The dictionaries contain the same values but are different objects. A change to one has no effect on the other.

Step 2 is in place.

9. Let's program step 1. This code segment executes after `updateDict` is initialized and replaces the placeholder `...` in instruction 8. Its purpose is to build `updateDict`. Iterate over states, find the nearest cluster to a state, and assign the state to that nearest cluster.

```
for state in histDict:
    mnD = 2 # Initialize the nearest state-to-cluster distance.
    stateHist = histDict[state]
    for a in clusterDict:
        clusterHist = clusterHistDict[a]
        abD = sum([(pai-pbi)**2 for pai,pbi in zip(stateHist,
                    clusterHist)])
        if abD < mnD:
            nearestCluster = a
            mnD = abD
    updateDict.setdefault(nearestCluster,[]).append(state)
```

We initialize the nearest distance to be `mn = 2` since the distance between clusters cannot be larger than 2 (see exercise 8.1).

10. After the `else` branch (instruction 8), print out the current assignments of states to clusters.

```
for a in updateDict:
    print('Cluster =',a,' Size = ',len(updateDict[a]),updateDict[a])
```

This `for` statement should be aligned with the `if` and `else` keywords.

11. Plot the k histograms. The code below is only slightly different than that used to plot the cluster histograms in the tutorial of Sect. 3.6.

```
import matplotlib.pyplot as plt

nIntervals = 30
intervals = [(12+2*i,12+2*(i+1)) for i in range(nIntervals)  ]

x = [np.mean(pair) for pair in intervals][:19]
for name in clusterDict:
    print(name )
    y = clusterHistDict[name][:19]

    plt.plot(x, y)
plt.legend([str(label) for label in range(6)], loc='upper right')
plt.show()
```

12. Recall that the initial configuration is random and so the final output may differ if the algorithm executes more than once. To gain some insight into the effect of the initial configuration, execute the code that begins with the random assignment of names to clusters and ends with constructing the figure several times. Try a few values of k, say $k \in \{4, 5, 6\}$. You should observe that a few associations of states are formed with regularity.

8.6.1 Synopsis

The k-means algorithm has the desirable property of always producing the same number of clusters but it also has the unfortunate property of not being deterministic in the sense that the initial random cluster configuration usually affects the final configuration. Having been introduced to two completing methods of cluster analysis, hierarchical agglomerative and k-means algorithms, we are faced with a decision: which to use?

Generally, when choosing one method from among a set of candidate methods that may be used accomplish an analytical objective, we strive to evaluate the candidate methods with respect to the theoretical foundations motivating the methods.[10] The k-means algorithm is an example of a method that can be motivated as the solution to a compelling minimization problem [2].

Suppose that we set out to form clusters so that the members are as close as possible to their centers. To make the objective more concrete, we define the within-cluster sums of squares associated with a particular set of assignments of observations to clusters, call it C, as

$$S(C) = \sum_{i=1}^{k}\sum_{j=1}^{n_i} ||\mathbf{x}_{i,j} - \overline{\mathbf{x}}_i||^2, \tag{8.11}$$

where $\overline{\mathbf{x}}_i = n_i^{-1}\sum_j \mathbf{x}_{i,j}$ is the vector of means computed from the n_i members of cluster i, for $i = 1, \ldots, k$, and $||\mathbf{y}|| = (\mathbf{y}^T\mathbf{y})^{1/2}$ is the Euclidean norm of the vector $\mathbf{y}$.

Given a particular initial configuration, the k-means algorithm will produce a rearrangement C that minimizes the within sums-of-squares. A solution C^* (i.e., an arrangement of observations into clusters) found by the k-means algorithm does not necessarily achieve the global minimum of $S(\cdot)$ over all possible configurations. The global minimum is the smallest value of the within sums-of-squares over every possible arrangement of states into k clusters. However, the algorithm may be executed repeatedly using N different initial configurations. A best solution may be selected as the solution that yielded the smallest value of $S(C)$ among the set of solutions $S(C_1), \ldots, S(C_N)$.

There's a variety of variations on the k-means algorithm. For example, the k-medoids algorithm uses observation vectors as cluster centers. A somewhat more complex algorithm with a solid foundation in statistical theory does not assign observations to clusters but instead estimates the probability of cluster membership of every observation in each cluster. Adopting this probabilistic modeling approach leads to the finite mixtures model and an EM algorithm solution [39].

[10] Other criteria are usually considered and may outweigh theoretical considerations.

8.7 Exercises

8.7.1 *Conceptual*

8.1. In the Sect. 8.4 tutorial, we set the initial minimum distance between clusters to be 2 (instruction 15). Prove the maximum distance between any two clusters is no more than 2, thereby insuring that the final minimum distance will not be 2. Specifically, prove that

$$\sum_i |a_i - b_i| \leq 2, \tag{8.12}$$

where $\sum a_i = \sum_i b_i = 1$.

8.2. For a particular configuration C of observations to clusters, show that the sample mean vectors $\overline{\mathbf{x}}_i = n_i^{-1} \sum_j \mathbf{x}_{ij}$ minimize $S(C)$. Begin with the objective function

$$S(C) = \sum_{i=1}^{k} \sum_{j=1}^{n_i} ||\mathbf{x}_{ij} - \boldsymbol{\mu}_i||^2. \tag{8.13}$$

Determine the vector $\widehat{\boldsymbol{\mu}}_i$ that minimizes $S(C)$ with respect to $\boldsymbol{\mu}_i$. Argue that if $\widehat{\boldsymbol{\mu}}_i$ minimizes $\sum_{j=1}^{n_i} ||\mathbf{x}_{ij} - \boldsymbol{\mu}_i||^2$, then $\widehat{\boldsymbol{\mu}}_1, \ldots, \widehat{\boldsymbol{\mu}}_k$ minimize $S(C)$.

8.7.2 *Computational*

8.3. Change the k-means algorithm so that the distance between state histograms and the cluster centroid is computed using the city-block (or L_1) metric instead of the Euclidean (or L_2) metric. After finding a set of clusters using the Euclidean metric, run the algorithm again using the city-block metric. Use the final configuration from the Euclidean metric as the initial configuration for the city-block metric. How often does a change in metric result in a different arrangement?

8.4. Return to the tutorial of Sect. 8.4 and modify the program so that the inter-cluster distance between clusters A and B is computed according to

a. The smallest distance between any $\mathbf{x}_i \in A$ and any $\mathbf{x}_j \in B$, say,

$$d_{\min}(A, B) = \min_{\mathbf{x}_i \in A, \mathbf{x}_j \in B} d_c(\mathbf{x}_i, \mathbf{x}_j)$$

b. The largest distance between any $\mathbf{x}_i \in A$ and any $\mathbf{x}_j \in B$, say,

$$d_{\max}(A, B) = \max_{\mathbf{x}_i \in A, \mathbf{x}_j \in B} d_c(\mathbf{x}_i, \mathbf{x}_j)$$

c. The average distance between $\mathbf{x}_i \in A$ and $\mathbf{x}_j \in B$, say,

$$d_{\mathrm{mean}}(A,B) = (n_A n_B)^{-1} \sum_{\mathbf{x}_i \in A, \mathbf{x}_j \in B} d_c(\mathbf{x}_i, \mathbf{x}_j)$$

Each one of these variants has a name [29]: single linkage (a), complete linkage (b), and average linkage (c). The metric programmed in Sect. 8.4 is known as centroid linkage.

8.5. Use the hierarchical agglomerative algorithm to build $k = 6$ clusters. Use these clusters as the initial configuration to the k-means algorithm. Compare the initial and final configurations.

Part III
Predictive Analytics

Chapter 9
k-Nearest Neighbor Prediction Functions

Abstract The purpose of the k-nearest neighbor prediction function is to predict a target variable from a predictor vector. Commonly, the target is a categorical variable, a label identifying the group from which the observation was drawn. The analyst has no knowledge of the membership label but does have the information coded in the attributes of the predictor vector. The predictor vector and the k-nearest neighbor prediction function generate a prediction of membership. In addition to qualitative attributes, the k-nearest neighbor prediction function may be used to predict quantitative target variables. The k-nearest-neighbor prediction functions are conceptually and computationally simple and often rival far more sophisticated prediction functions with respect to accuracy. The functions are nonparametric in the sense that the mathematical basis supporting the prediction functions is not a model. Instead the k-nearest neighbor prediction function utilizes a set of *training* observations on target and predictor vector pairs and, in essence examines the target values of the training observations nearest to the target. If the target variable is a group membership label, the target is predicted to be to the most common label among the nearest neighbors. If the target is quantitative, then the prediction is an average of the target values associated with the nearest neighbors.

9.1 Introduction

A common problem in data science is to identify the group or class membership of an observational unit using a set of attributes measured on the unit. For example, most email clients (programs that handle incoming email) direct email messages to folders with labels similar to *primary*, *social*, *promotions*, and *spam*. It's desirable that messages that are sent en mass to large numbers of recipients are quarantined or rejected. These messages are sometimes

B. Steele et al., *Algorithms for Data Science*,
DOI 10.1007/978-3-319-45797-0_9

referred to using the pejorative *spam* and the content of these messages is often advertisements. Sometimes malware or links to sites that host malware are embedded in the messages. Therefore, the client takes on the task of labeling messages as spam or ham (ham is not spam). But when an email message is received, there are no definitive attributes that identify a message as spam and the client must use a predictive function for the assignment of spam or ham labels to the incoming messages.

An algorithm for classifying email messages extracts information about a message such as the presence of specific words and characters and the length of the longest run of capitalized letters. This information is encapsulated in a vector of predictor attributes that is passed to a prediction function. The prediction function returns a prediction of the message type, and it will be one of spam or ham, or perhaps one of primary, social, promotions, or spam. Before the prediction function is put into use, an assessment of the function's accuracy is valuable if not essential. There are a wide variety of prediction functions and several methods for assessing the accuracy of a prediction function. Collectively, the subject of prediction and accuracy assessment is referred to as *predictive analytics*.[1] Problems in which the target variable identifies membership in a group are sometimes referred to as *classification* problems.

This chapter discusses the first of two types of prediction functions presented in this text, the k-nearest-neighbor prediction functions. The k-nearest neighbor prediction functions are simple conceptually and mathematically yet are often quite accurate. Their drawback is that the computational demand of using a k-nearest-neighbor prediction function may be impractically great. We also discuss accuracy assessment, an essential part of predictive analytics in this chapter.

9.1.1 The Prediction Task

Our discussion of k-nearest neighbor prediction functions begins with the prediction task. For now, suppose that the target variable is qualitative, or categorical, and that it identifies the membership of an observational unit in one of g possible groups. Section 9.8 discusses the use of k-nearest neighbor prediction functions for predicting quantitative target variables.

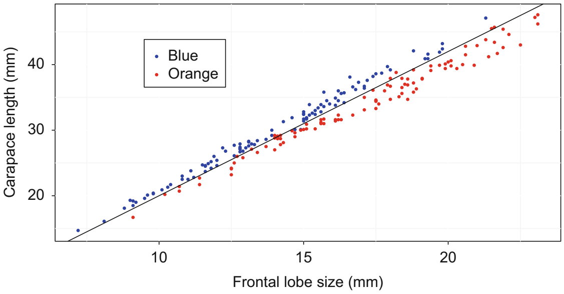
A k-nearest neighbor prediction function will use a set of observation pairs $D = \{(y_1, \mathbf{x}_1), \ldots, (y_n, \mathbf{x}_n)\}$ for which all of the target values $y_1, \ldots, y_n$ have been observed. A target observation is a pair $(y_0, \mathbf{x}_0)$ for which the label y_0 is missing (or at least is to be predicted) and $\mathbf{x}_0$ has been observed. The

[1] Predictive analytics is used as a term primarily in data science. Statistics and computer science have their own names, predominantly, statistical learning and machine learning, respectively.

elementary version of the k-nearest neighbor prediction function for classification problems operates by determining the k most similar observations to $(y_0, \mathbf{x}_0)$ among the training set based on the distances between $\mathbf{x}_0$ and $\mathbf{x}_1, \ldots, \mathbf{x}_n$. The prediction of y_0 is the most common label among the most similar observations. These k most similar observations are the k-nearest neighbors. In the case that the target variables are quantitative, the elementary k-nearest neighbor regression function predicts y_0 to be the mean of the k-nearest neighbor target values.

A simple example is provided by the `crabs` data set [9, 64] from the `R` `MASS` library. Figure 9.1 shows that the two color forms of the crab species Leptograpsus variegatus can be imperfectly separated on the basis of carapace length and frontal lobe size. The graphed line is a boundary between the two color forms determined by trial and error. The boundary may be used to build a prediction function that predicts a crab of unknown color form to be blue or red depending on whether its frontal lobe size and carapace length locate it above or below the line.

Fig. 9.1 Observations on carapace length and frontal lobe size measured on two color forms of the species Leptograpsus variegatus. $n = 100$



Alternatively, a five-nearest neighbor prediction function predicts the color form by determining the five nearest neighbors to $(y_0, \mathbf{x}_0)$ and then the most common color among the neighbors. For most of the blue form data pairs, the majority of the nearest five neighbors are also blue. For the majority of the near neighbors of the orange form with frontal lobe size greater than 15, the near neighbors of the orange form are also orange. When frontal lobe size is less than 14 mm, the five nearest neighbors are often predominantly blue, and so the five-nearest neighbor prediction function will tend to incorrectly assign membership to orange crabs with frontal lobe size less than 14 mm.

In this example, it appears that the ad hoc boundary line prediction function will be about as accurate as the five-nearest neighbor function. If there were another morphological attribute, then determining the boundary becomes more difficult.[2] If there were more than three morphological attributes, then determining the boundary by trail and error is impractical and we will have to resort to a nontrival mathematical solution. The computational demand of a five-nearest neighbor prediction function changes very little as more attributes are brought to bear on the prediction problem.

[2] The boundary would be a plane.

9.2 Notation and Terminology

An observation pair with an unknown target value y_0 is denoted by $\mathbf{z}_0 = (y_0, \mathbf{x}_0)$. The p-length predictor vector $\mathbf{x}_0 = [x_{0,1} \;\; \cdots \;\; x_{0,p}]^T$ has been observed and will be used to predict y_0 via a prediction function. The prediction function $f(\cdot|D)$ is built from, or trained on, the data set $D = \{\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_n\}$. The conditional notation in the expression $f(\cdot|D)$ emphasizes the role of the training set in fitting the predictive function. In this discussion, D is referred to as the training set, and so $\mathbf{z}_i = (y_i, \mathbf{x}_i)$ is the ith training observation. For all pairs $\mathbf{z}_i \in D$, y_i and $\mathbf{x}_i$ are known. A prediction of y_0 is denoted as $\widehat{y}_0 = f(\mathbf{x}_0|D)$. For example, if the targets are quantitative, we may consider using the linear regression prediction function $f(\mathbf{x}_0|D) = \mathbf{x}_0^T\widehat{\boldsymbol{\beta}}$. The prediction function often does not have a simple closed form as just shown and $f(\cdot|D)$ is more transparently described as a (multi-step) algorithm that takes $\mathbf{x}_0$ as an input and outputs $\widehat{y}_0$.

If the target variable is qualitative, then it is assumed herein that targets are labels that identify group or subpopulation membership. The number of possible groups is g, and for convenience, the set of labels is $\{0, 1, \ldots, g-1\}$. Given the predictor vector $\mathbf{x}_0$, the probability that $\mathbf{z}_0$ is a member of group j is expressed as $\Pr(y_0 = j|\mathbf{x}_0)$. Every unit belongs to exactly one of the g groups and so $\sum_{j=0}^{g-1} \Pr(y_0 = j|\mathbf{x}_0) = 1$.

It's convenient to use indicator functions to identify group membership. The indicator of membership in group j is defined to be

$$I_j(y) = \begin{cases} 1, \text{ if } y = j, \\ 0, \text{ if } y \neq j, \end{cases}$$

where $y \in \mathbb{R}$, though usually y is an actual or predicted group label. Without the predictor vectors as a source of information, the probability of membership in group j routinely is estimated by the sample proportion of training observations belonging to group j. The sample proportion can be expressed using indicator functions:

$$\widehat{\Pr}(y_0 = j) = n^{-1} \sum_{i=1}^{n} I_j(y_i).$$

If we were to predict group membership without knowledge of the predictor vectors, and we train $f(\cdot|D)$ by maximizing its accuracy when applied to the training observations, then every unlabeled unit would be predicted to be a member of the group with the largest sample proportion.[3] The accuracy of this trivial prediction function is a baseline against which the accuracy of more complex prediction functions may be compared. This baseline is similar

[3] We're assuming that accuracy of the prediction function is estimated by applying the prediction function to the training observations and computing the proportion of correct predictions.

in spirit to the baseline fitted model $\widehat{\mu}_i = \overline{y}$, for $i = 1, \ldots, n$, used in the linear regression measure of fit R^2_{adjusted}.

If the predictor vectors $\mathbf{x}_1,\ldots,\mathbf{x}_n$ are informative, then the baseline accuracy can be improved upon by examining the target values associated with training vectors that are similar to $\mathbf{x}_0$. This is the k-nearest-neighbor idea. More concretely, only the training observations that are in a neighborhood of $\mathbf{x}_0$ are to be used for prediction. Supposing the targets are group labels, then the unlabeled unit $\mathbf{z}_0$ would then be predicted to belong to the most commonly occurring group in the neighborhood. The neighborhood and its members are determined by the distances between $\mathbf{x}_0$ and $\mathbf{x}_1, \ldots, \mathbf{x}_n$.

The distances between the predictor vector $\mathbf{x}_0$ and $\mathbf{x}_1, \ldots, \mathbf{x}_n$ are denoted as $d(\mathbf{x}_0, \mathbf{x}_1), d(\mathbf{x}_0, \mathbf{x}_2), \ldots, d(\mathbf{x}_0, \mathbf{x}_n)$. We determine the order of these distances from smallest to largest and arrange the training set in the same order. Mathematically, the ordered training set is an n-tuple[4]

$$(\mathbf{z}_{[1]}, \mathbf{z}_{[2]}, \ldots, \mathbf{z}_{[n]}), \tag{9.1}$$

where $\mathbf{z}_{[k]} = (y_{[k]}, \mathbf{x}_{[k]})$. The square bracket notation identifies $\mathbf{x}_{[1]}$ as the nearest predictor vector to $\mathbf{x}_0$, and $\mathbf{x}_{[2]}$ as second-nearest to $\mathbf{x}_0$, and so on. The label, or target value, of the nearest *neighbor* $\mathbf{z}_{[1]}$ is denoted as $y_{[1]}$, the target value of the second nearest neighbor $\mathbf{z}_{[2]}$ is $y_{[2]}$, and so on.

9.3 Distance Metrics

Euclidean and Manhattan metrics are commonly used to compute distances between predictor vectors. To produce sensible results both metrics require that all of the attributes comprising the predictor vector are quantitative. If this condition is met, then the Euclidean distance between $\mathbf{x}_i = [x_{i,1} \ \cdots \ x_{i,p}]^T$ and $\mathbf{x}_0 = [x_{0,1} \ \cdots \ x_{0,p}]^T$ is

$$d_E(\mathbf{x}_i, \mathbf{x}_0) = \left[\textstyle\sum_{j=1}^{p} (x_{i,j} - x_{0,j})^2\right]^{1/2}.$$

If the predictor variables differ substantially with respect to variability, then it is often beneficial to scale each variable by the sample standard deviation of the variable. Without scaling, the squared differences $(x_{i,j} - x_{0,j})^2, j = 1, \ldots, p$ will tend to be largest for the variable with the largest variance. Consequently, the variable with the largest variability will have the greatest influence toward determining the distance between $\mathbf{x}_i$ and $\mathbf{x}_0$ regardless of the information content of the variable toward prediction. It's desirable to

[4] A point in the $\mathbb{R}^2$ is a two-tuple, otherwise known as a pair.

carry out scaling at the same time as the distance is computed. With scaling, the distance metric becomes

$$d_S(\mathbf{x}_i, \mathbf{x}_0) = \left[\sum_{j=1}^{p} \left(\frac{x_{i,j} - x_{0,j}}{s_j} \right)^2 \right]^{1/2},$$

where s_j is the estimated standard deviation of the jth predictor variable. The sample variance, from which s_j is computed, is

$$s_j^2 = (n-g)^{-1} \sum_{k=1}^{g} \sum_{i=1}^{n} I_k(y_i)(x_{i,j} - \overline{x}_{j,k})^2, \tag{9.2}$$

where $\overline{x}_{j,k}$ is the sample mean of attribute j computed from observations belonging to group k. The presence of the term $I_k(y_i)$ insures that only observations belonging to group k contribute to the inner sum. In statistics, s_j^2 is known as the pooled sample variance.

The Manhattan or city-block distance between $\mathbf{x}_i$ and $\mathbf{x}_0$ is

$$d_C(\mathbf{x}_i, \mathbf{x}_0) = \sum_{j=1}^{p} |x_{i,j} - x_{0,j}|.$$

Scaling may be employed with city-block distance. In most applications of the k-nearest neighbor prediction functions, selecting a best neighborhood size k is more important than selecting a best distance metric since different metrics often result in similar orderings and neighborhood sets.

If a predictor variable consists of p qualitative variables, then Hamming distance is a useful metric for comparing $\mathbf{x}_0$ and $\mathbf{x}_i$. The Hamming distance between $\mathbf{x}_0$ and $\mathbf{x}_i$ is the number of the p attributes comprising $\mathbf{x}_0$ and $\mathbf{x}_i$ are not the same. The distance is computed according to

$$d_H(\mathbf{x}_i, \mathbf{x}_0) = p - \sum_{j=1}^{p} I_{x_{i,j}}(x_{0,j}). \tag{9.3}$$

Note that $I_{x_{i,j}}(x_{0,j})$ is 1 if $x_{i,j} = x_{0,j}$, and 0 otherwise, and so the sum in Eq. (9.3) counts the number of attributes, or positions, in which $\mathbf{x}_0$ and $\mathbf{x}_i$ contain the same values. Since there are p total attributes, $p - \sum_{j=1}^{p} I_{x_{i,j}}(x_{0,j})$ is the number of positions that do not match.

9.4 The k-Nearest Neighbor Prediction Function

Assume for now that y_0 is a group label. The k-nearest neighbor prediction of y_0 is constructed by determining a neighborhood of k training observations nearest to $\mathbf{x}_0$. Then, the proportion of the k neighbors that belong to group

j is computed and y_0 is predicted to be the group with the largest proportion of neighbors. This prediction rule is equivalent to predicting $\mathbf{z}_0$ to belong to the most common group among the k nearest neighbors.

To develop the prediction function formally, the estimated probability of membership in group j is defined to be the proportion of group j members among the k nearest neighbors:

$$\widehat{\Pr}(y_0 = j|\mathbf{x}_0) = \frac{n_j}{k}, j = 1, \ldots, g,$$

where n_j is the number of the k-nearest neighbors that belong to group j. An expression for $\widehat{\Pr}(y_0 = j|\mathbf{x}_0)$ in terms of indicator variables will be helpful in the next section:

$$\widehat{\Pr}(y_0 = j|\mathbf{x}_0) = k^{-1}\sum_{i=1}^{k} I_j(y_{[i]}). \tag{9.4}$$

The estimated probabilities of membership are collected as a vector

$$\underset{g\times 1}{\widehat{\mathbf{p}}_0} = \left[\widehat{\Pr}(y_0 = 1|\mathbf{x}_0) \quad \cdots \quad \widehat{\Pr}(y_0 = g|\mathbf{x}_0)\right]^T.$$

The final stage of computing the prediction determines the largest value in $\widehat{\mathbf{p}}_0$. The arg max$(\cdot)$ function determines which element of $\widehat{\mathbf{p}}_0$ is largest. Suppose that $\mathbf{w} = [w_1 \quad \cdots \quad w_g]^T$. Then, arg max$(\mathbf{w})$ is the index of the largest element in $\mathbf{w}$. For example, if $\mathbf{w} = [0 \;\; 0 \;\; 1]^T$, then arg max$(\mathbf{w}) = 3$. Thus, the k-nearest-neighbor prediction function is

$$f(\mathbf{x}_0|D) = \arg\max\left(\widehat{\mathbf{p}}_0\right). \tag{9.5}$$

If the maximum estimated probability is common to two or more groups, then the `Numpy` function `argmax` returns the index of first occurrence of the maximum value in the vector. Testing for and breaking ties requires a little effort. The tie can be broken randomly or by increasing the neighborhood size by one additional neighbor and recomputing $\widehat{\mathbf{p}}_0$ until the tie is broken. We'll avoid ties entirely by using a variant of the conventional k-nearest-neighbor function (Sect. 9.5) that does not produce ties among the probability estimates.

A computationally efficient algorithm for k-nearest neighbor prediction consists of two functions. The first function computes the complete ordered arrangement $\mathbf{y}^o = (y_{[1]}, y_{[2]}, \ldots, y_{[n]})$ from the inputs $\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_n$. The second function computes the elements of $\widehat{\mathbf{p}}$ according to Eq. (9.4) and using the first k terms of $\mathbf{y}^o$. However, the complete arrangement $\mathbf{y}^o$ is useful if several k-nearest-neighbor functions are used at once, for example, in a search for a best k, or if there's a tie among the largest estimated probabilities of group membership. Since the first function returns the ordered arrangement, there's no need to compute and sort the distances more than once. Computationally, the first function is expensive time-wise because of the sort operation. In comparison, the second function is very fast.

9.5 Exponentially Weighted k-Nearest Neighbors

The conventional k-nearest neighbor prediction function can be improved modestly with respect to accuracy and programming effort. The idea is to use all of the neighbors of a target rather than only the nearest k neighbors. Each neighbor is assigned a measure of importance, or weight towards determining $f(\mathbf{x}_0|D)$, according to its relative distance to $\mathbf{x}_0$. The nearest neighbor receives the largest weight and the weights decay toward zero as we move toward more distant neighbors.

We begin by generalizing the estimator defined in formula (9.4) as a weighted sum over all n neighbors. The weights are

$$w_i = \begin{cases} 1/k, & \text{if } i \leq k \\ 0, & \text{if } i > k, \end{cases} \tag{9.6}$$

for $i = 1, \ldots, n$. Then, an alternative expression for the estimator of the probability of membership in group j is

$$\widehat{\Pr}(y_0 = j|\mathbf{x}_0) = \sum_{i=1}^{n} w_i I_j(y_{[i]}). \tag{9.7}$$

In formula (9.7), the weights are equal to $1/k$ for the first k neighbors and then are equal to zero for the rest of the neighbors. It's difficult to justify the weighting scheme—why should the information content drop abruptly to 0 at a boundary between $\mathbf{z}_k$ and $\mathbf{z}_{k+1}$? A more plausible scenario is that information content decreases with each successively more distant neighbor. These considerations suggest that the weights in formula (9.7) ought to be replaced with a set of weights that decay smoothly as k increases. Therefore, let's replace the weights of formula (9.6) by weights that decay exponentially.

The *exponentially weighted k-nearest neighbor prediction function* estimates the probability of membership using formula (9.7) with a different set of weights $w_1, \ldots, w_n$ derived from theoretical considerations [58]. In the conventional k-nearest-neighbor function, the influence of neighbors is controlled through the choice of the neighborhood size k. With the exponentially weighted k-nearest neighbor prediction function, the influence of neighbors is controlled through the choice of a tuning constant α, a number bounded by zero and one. The weights are a function of α defined by

$$w_i = \alpha(1-\alpha)^{i-1}, i = 1, 2, \ldots, n, \tag{9.8}$$

for $0 < \alpha < 1$. The sum of the weights is approximately 1 provided that n is large (exercise 9.1). The left panel of Fig. 9.2 shows the weights corresponding to the conventional k-nearest neighbor prediction functions for k equal to 3, 5, 10, and 20. The right panel shows similar weights for the exponentially weighted k-nearest neighbor prediction functions. Specifically, we show weights for α equal to $.33\overline{3}$, .2, .1, and .05, (the reciprocals of $k \in \{3, 5, 10, 20\}$) and corresponding to weighting schemes one through four, respectively. Large values of α place more weight on the nearest neighbors and produce weights that decay rapidly to zero whereas the rate of decay is slower for smaller values of α.

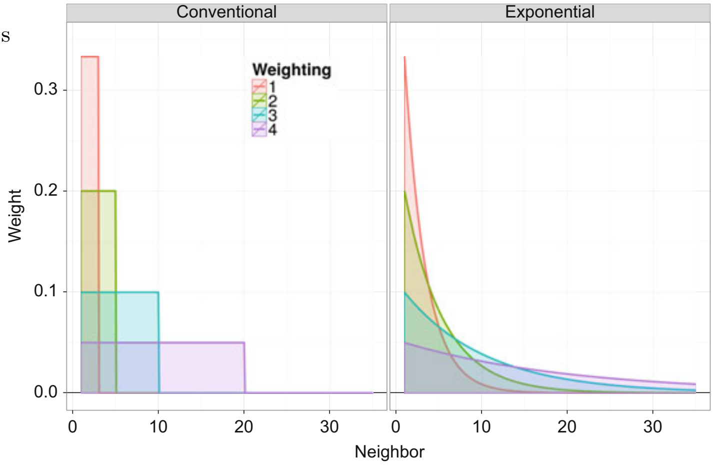


Fig. 9.2 Weights assigned to neighbors by the conventional k-nearest neighbor and exponentially-weighted k-nearest neighbor prediction functions. See the text for details

Choosing α is sometimes easier if one keeps in mind that the nearest neighbor receives weight α. For instance, $\alpha = .2$ implies that the nearest neighbor receives the same weight as a conventional five-nearest neighbor prediction function since $\alpha = .2 = 1/5$.

9.6 Tutorial: Digit Recognition

The Kaggle [32] competition *Digit Recognizer* provides an interesting data set for k-nearest-neighbor prediction. The data were donated in a competition to correctly label optically-scanned handwritten digits. For details, navigate to the website https://www.kaggle.com/c/digit-recognizer. The data consist of 42,000 digitized images of digits, one record per image. Accordingly, the group label of a particular optical image must be one of the set $\{0, 1, \ldots, 9\}$. The predictor vector extracted from an optical image consists of darkness intensity[5] measured on each of the $28 \times 28 = 748$ pixels comprising the optical image. The prediction problem is to use the vector of 748 darkness values obtained from an unlabeled image and correctly predict the handwritten digit.

[5] Darkness is recorded on a scale of 0–255.

The tutorial guides the reader through the programming of a `Python` script that will estimate the accuracy of the conventional and exponentially weighted k-nearest neighbor prediction functions for the task posed above. Our approach to accuracy estimation is to draw a subset R from the data set D and use it to build a conventional k-nearest-neighbor prediction function and an exponentially weighted k-nearest-neighbor prediction function. A second subset E, disjoint from R, is drawn to evaluate the accuracy of the prediction functions. The subsets R and E are referred to as the training and test sets, respectively. We estimate accuracy by obtaining a prediction $\widehat{y}_i = f(\mathbf{x}_i|R)$ from each observation $\mathbf{z}_i \in E$. A comparison of the predicted and actual labels yields the proportion of correct predictions. Insuring that the training and test sets are disjoint is important because using training observations to evaluate the accuracy of a prediction function poses a significant risk of overestimating accuracy.

But more can be done with the results than computing the proportion of correctly labeled test observations. The result of a prediction, say, $\widehat{y}_i = f(\mathbf{x}_i|R)$ may be one of $g = 10$ values, as may be the actual value. Thus, there are 10×10 possible combinations of outcomes. It may be that some digits are more frequently confused than others. For instance, 3 and 8 might be confused more often than 1 and 8. To extract some information on the types of errors that are most likely, we'll cross-classify the predictions against the actual target labels by tabulating the number of times that each combination is observed. The table containing the cross-classification of actual and predicted target values is called the *confusion matrix*. Let's suppose that a prediction is computed for $\mathbf{z}_0$ and that the actual label of y_0 is j and the predicted label is h. Then, the count in row j and column h will be incremented by one. After processing a reasonably large number of test observations, we'll have an understanding of the type of errors incurred by the prediction function. A convenient way of determining the number of correctly classified test observations is to compute the sum of the diagonal elements of the confusion matrix. Since every test observation is classified once, the accuracy is estimated by the sum of the diagonal divided by the sum over the entire table.

Section 9.7 goes into greater depth on the subject of accuracy assessment. In the tutorial, we'll build a three-dimensional array consisting of two back-to-back $g \times g$ confusion matrices, one for the conventional k-nearest-neighbor prediction function and the second for the exponentially weighted k-nearest-neighbor prediction function. The label pairs $\{(y_i, \widehat{y}_i)|\mathbf{z}_i \in E\}$ provide the data that fills the confusion matrix.

The data set is large by conventional standards with 42,000 observations and consequently, execution time is slow for the k-nearest neighbor prediction functions. To speed development and testing of the `Python` code, the tutorial does not use all of the observations in D.

The principal four steps of the tutorial are as follows.

1. Create training and test sets. The prediction functions will be constructed from, or trained on, the training set R and tested on the test set E. The formation of R and E is accomplished by systematically sampling

the Kaggle training set, `train.csv`. Though `train.csv` is small enough (66 MB) that in-memory storage is feasible, the tutorial instructs the reader to process the file one record at a time. In any case, the first task is to draw training and test sets of $n_R = 4200$ and $n_E = 420$ observations respectively from the data set.

2. Construct a function f_{order} that will determine the neighbors of $\mathbf{z}_0 \in E$, ordered with respect to the distance between $\mathbf{x}_0$ and $\mathbf{x}_1, \ldots, \mathbf{x}_{n_R}$. The function will return the ordered labels of the neighbors. More formally, the function will compute an ordered arrangement of the training observation labels $\mathbf{y}^o = (y_{[1]}, \ldots, y_{[n_R]})$, hence, $\mathbf{y}^o = f_{\text{order}}(\mathbf{x}_0|R)$.
3. Write a function f_{pred} that computes two predictions of y_0 from $\mathbf{y}^o$ using the conventional and exponentially weighted k-nearest-neighbor prediction functions, respectively. The three arguments passed to f_{pred} are $\mathbf{y}^o$, the neighborhood size k, and the vector of weights $\mathbf{w} = [w_1 \ \cdots \ w_{n_R}]^T$ defined by Eq. (9.8). The weight vector determines the exponentially weighted k-nearest-neighbor function in the same way that k determines the conventional k-nearest-neighbor function.
4. Fill the confusion matrix with the outcome pairs $\{(y_i, \widehat{y}_i)|\mathbf{z}_i \in E\}$. To do so, every test observation $\mathbf{z}_i = (y_i, \mathbf{x}_i) \in E$ will be cross-classified according to its actual (y_i) and predicted label $(\widehat{y}_i)$. Accuracy estimates for the two prediction functions will be computed from the matrix.

Detailed instructions follow.

1. Download `train.csv` from https://www.kaggle.com/c/digit-recognizer. The file is rectangular in the sense that aside from the first record containing the variable names, each record has the same number of attribute values. There are $748 = 28^2$ attributes, each of which is a measurement of darkness for one pixel in a 28×28 field. Attributes are comma-delimited.
2. Initialize dictionaries `R` and `E` to store the training and test sets, respectively. Read the data file one record at a time. The first record contains the column names. Extract that record using the `readline` attribute of `f` before iterating over the remainder of the file. Print the variable names. Iterate over the file and print the record counter `i`.

```
import sys
import numpy as np
R = {}
E = {}
path = '../train.csv'
with open(path, encoding = "utf-8") as f:
    variables = f.readline().split(',')
    print(variables)
    for i, string in enumerate(f):
        print(i)
```

On each iteration, `i` is incremented by using the `enumerate` function.

3. As the file is processed, build the dictionaries R and E. The dictionary keys are record counters, or indexes, and the dictionary values will be pairs consisting of a target value y and a predictor vector $\underset{748\times 1}{\mathbf{x}}$. Store $\mathbf{x}$ as a list.

   Add observation pairs to the training dictionary whenever the count of processed records is a multiple of 10. Add pairs to the test dictionary whenever $i \bmod 100 = 1$. None of the training pairs will be included in the test set.

```
if i%10 == 0:
    record = string.split(',')
    y = int(record[0])
    x = [int(record[j]) for j in np.arange(1, 785)]
    R[i] = (y, x)
if i%100 == 1:
    record = string.split(',')
    y = int(record[0])
    x = [int(record[j]) for j in np.arange(1, 785)]
    E[i] = (y, x)
```

   Note that the first element in `record` identifies the digit and therefore is the target value of the ith observation. This code segment must execute every time that a record is read.

4. Initialize an array `confusionArray` to store the results of predicting the test targets using the conventional and exponentially weighted k-nearest-neighbor prediction functions. Since there are two prediction functions (conventional and exponentially weighted k-nearest-neighbor prediction functions), we need two 10 × 10 confusion matrices to count the occurrences of each combination. It's convenient use one three-dimensional array to store the two confusion matrices as side-by-side 10 × 10 arrays.

   Initialize the constants and the storage array for the confusion matrices.

```
p = 748   # Number of attributes.
nGroups = 10

confusionArray = np.zeros(shape = (nGroups, nGroups, 2))
acc = [0]*2  # Contains the proportion of correct predictions.
```

   The code segment executes upon completion of building the dictionaries R and E.

5. Create an n_R-element list containing the labels of the training observations.

```
labels = [R[i][0] for i in R]
```

6. Set the neighborhood size k and smoothing constant α. Construct the n_R-element list of weights for the exponentially weighted k-nearest-neighbor prediction function.

```
nR = len(R)
k = 5
alpha = 1/k
wts = [alpha*(1 - alpha)**i for i in range(nR)]
```

Check that $\sum_i w_i = 1$ by summing the elements of `wts`.

7. The program flow for the prediction task is shown in the next code segment. We iterate over the observation pairs in the test set `E` and extract a predictor vector and label on each iteration. Each predictor vector ($\mathbf{x}_0$) is passed with the training set `R` to the function `fOrder`. The function `fOrder` returns the ordered training labels $\mathbf{y}^o$ as a list named `nhbrs`. The list `nhbrs` is passed to `fPredict` to compute predictions $\widehat{y}_{\text{conv}}$ and $\widehat{y}_{\text{exp}}$ using the conventional and exponentially weighted k-nearest-neighbor prediction functions, respectively. The function `fPredict` returns a two-element list `yhats` containing $\widehat{y}_{\text{conv}}$ and $\widehat{y}_{\text{exp}}$. The `for` loop indexed by `j` updates the confusion matrices and computes the estimated accuracy rates as the proportion of correctly classified test observations.

```
yhats = [0]*2
for index in E:
    y0, x0 = E[index]

    #nhbrs = fOrder(R,x0)
    #yhats = fPredict(k,wts,nhbrs)
    for j in range(2):  # Store the results of the prediction.
        confusionArray[y0,yhats[j],j] += 1
        acc[j] = sum(np.diag(confusionArray[:,:,j]))
               /sum(sum(confusionArray[:,:,j]))
    print(round(acc[0],3),’  ’,round(acc[1],3))
```

In the segment code above, accuracy is estimated after each test observation is processed as a means of tracing the execution of the program. The functions `fOrder` and `fPredict` do not exist at this point, of course. Introduce the code segment into your script. To test the code, temporarily set `yhats = [y0, y0]`. Execute the script and verify that `acc` contains 1.0 in both positions.

8. It remains to implement the k-nearest-neighbor prediction function. It's best to implement the code not as a function, but in the main program because variables computed inside functions are local and cannot be referenced outside the function. When you're satisfied that the code is correct, then move the code to a function *outside* of the loop.

   The first function to program is `fOrder`. Code it within the `for` loop that iterates over the test set `E`. Its purpose is to create the ordered vector of labels $\mathbf{y}^o$ from a test vector $\mathbf{x}_0$ and the training set `R`. Ordering is determined by the distances of each training vector to $\mathbf{x}_0$. We'll compute the distances in a `for` loop that iterates over `R`. On each iteration of the `for` loop, compute the distance between $\mathbf{x}_0$ and $\mathbf{x}_i \in R$. Save the distances in a list named `d`:

```
d = [0]*len(R)
for i, key in enumerate(R):
    xi = R[key][1]  # The ith predictor vector.
    d[i] = sum([abs(x0j - xij) for x0j, xij in zip(x0,xi)])
```

   The distance `d[i]` is computed by zipping the vectors `x0` and `xi` together. Using list comprehension, we iterate over the zip object and build a list containing the absolute differences $|x_{0,j} - x_{i,j}|$, for $j = 1, \ldots, p$. The last operation computes the sum of the list.

9. Compute a vector $\mathbf{v}$ that will sort the distances from smallest to largest. That is, it is a vector of indexes such that

$$\texttt{d}[\mathrm{v}[0]] \leq \texttt{d}[\mathrm{v}[1]] \leq \texttt{d}[\mathrm{v}[2]] \leq \cdots.$$

   The vector $\mathbf{v}$ will also sort the vector of training observation labels so that the label of the nearest observation is `labels[v[0]]`, the label of the second nearest observation is `labels[v[1]]` and so on. The `Numpy` function `argsort` computes $\mathbf{v}$ from $\mathbf{d}$. Using `v`, list comprehension is used to compute the ordered neighbors $\mathbf{y}^o$, or `nhbrs`.

```
v = np.argsort(d)
nhbrs = [labels[j] for j in v] # Create a sorted list of labels.
```

   This code segment executes *after* `d` has been filled.

10. Test the code by printing the labels of the k-nearest neighbors and the label y_0. There should be good agreement—for most test observations the majority of neighbors should have the same group label as y_0.
11. When the code appears to function correctly, move the `fOrder` function outside of the `for` loop. This function computes the distances between $\mathbf{x}_0$ and $\mathbf{x}_i \in R$ and the ordering vector `v`. Lastly, it arranges the neighbors according to the distances (instructions 8 and 9). The definition and return statements are

```
def fOrder(R,x0):
    ...
    return nhbrs
```

The function call is

```
nhbrs = fOrder(R,x0)
```

12. Move the code described in instructions 8 and 9 to the function definition. Call the function instead of executing the code in the main program. Check that the nearest k neighbors usually match the target y_0.

    The function `fOrder` will be used again in the tutorial of Sect. 9.9.
13. The next step is to determine which group is most common among the k-nearest neighbors. This task will be performed by the function `fPred`. As with `fOrder`, write the code in place. When it works, move it out of the main program and into a function.

    Begin with the conventional k-nearest-neighbor prediction of y_0. Initialize a list of length g to store the number of the k-nearest neighbors that belong to each of the g groups. Count the number of nearest neighbors belonging to each group:

```
counts = [0]*nGroups
for nhbr in nhbrs[:k]:
    counts[nhbr] += 1
```

    Since `nhbr` is a group label, and either $0, 1, \ldots, 8$ or 9, the statement `counts[nhbr] += 1` increments the count of the k-nearest neighbors belonging to the group membership of the neighbor.
14. The exponentially weighted k-nearest-neighbor prediction function estimates the group membership probabilities $\widehat{\Pr}(y_0 = j|\mathbf{x}_0)$ for each group $j \in \{0, \ldots, g-1\}$. The calculation is described by Eq. (9.7), and it uses the list of weights `wts` computed in instruction 6. Iterate over the n_R neighbors of $\mathbf{z}_0$ using `i` as an index. Accumulate the weights for each group by adding w_i to the group to which $\mathbf{z}_{[i]}$ belongs.

    Use the `for` loop coded in instruction 13 but iterate over *all* of the neighbors instead of ending with the kth neighbor. The code segment is

```
counts = [0]*nGroups
probs = [0]*nGroups
for i, nhbr in enumerate(nhbrs): # Iterate over all neighbors.
    if i < k:
        counts[nhbr] += 1
    probs[nhbr] += wts[i]        # Increment the probability of
                                 # membership in the nhbr's group.
```

Execute the code and print `probs` and the sum of `probs` as each test observation is processed. The sums (`sum(probs)`) must be equal to 1.

15. Identify the most common group among the k-nearest neighbors for the conventional k-nearest-neighbor prediction function. Also determine the index of the largest estimated probability for the exponentially weighted k-nearest-neighbor prediction function:

```
yhats = [np.argmax(counts), np.argmax(probs)]
```

The `Numpy` function `np.argmax(u)` identifies the index of the largest element in `u`. If more than one value is the maximum in `counts`, the `Numpy` function `argmax` identifies the first occurrence of the maximum. It's desirable to break ties in some other fashion. Exercise 9.4 provides some guidance.

Test the code by printing `counts` and `yhats`. The first value of `yhats` should index the largest count. The two predictions contained in `yhats` should be in agreement most of the time.

16. Build the function `fPred` using the code segment developed in instructions 13 through 15. The arguments passed to the function are `nhbrs` and `nGroups` and the function returns `yhats`.
17. Compute and print the overall accuracy estimates as the program iterates over the test set observations. Accuracy estimates that are less than .75 are improbable for these data and suggest programming errors.
18. Double n_R and n_E and execute the script.

9.6.1 Remarks

The k-nearest-neighbor prediction functions are computationally demanding because of the sorting step. The execution time of the best sorting functions increase not linearly with the number of items n to sort, but more rapidly than a linear function of n.[6] For the digits problem, using all of the 42,000

[6] Sorting algorithm run-times are, at best, on the order of $n\log(n)$ [56].

observations is desirable in a practical application. But using all of the training observations often would result in a too-slow prediction algorithm if the algorithm is applied to a process, say, reading zipcodes on envelopes in a post office.

One solution is to replace the original training set D by a smaller set of group representatives. For example, the representatives may be chosen by randomly sampling each group. A concern with this approach is that the sample may not span the full spectrum of hand-written variants of a digit present in D. It's easy to investigate subsampling using the script written in Sect. 9.6 by modifying the conditional statement that draws every tenth observation to form `R` so that every fiftieth observation is drawn into `R`.

Random sampling may be improved on by applying a k-means clustering algorithm to each group. The k-means algorithm produces cluster means that tend to differ and so are more likely to span the range of variation in hand-written digits. For the digits example, the clustering algorithm may generate $k = 100$ representatives, specifically, cluster means, for each digit. Combining the representatives as a single training set produces a much smaller training set of 1000 observations. The execution time of a k-nearest-neighbor prediction algorithm constructed from 1000 representatives would be satisfactory for a wide range of problems.

9.7 Accuracy Assessment

The k-nearest-neighbor prediction functions are two among many prediction functions that are used in data science. There is no single prediction function that is uniformly most accurate, or even predictably most accurate among the commonly used prediction functions. What is most important in determining accuracies is the extent to which the space $\mathcal{X}$ spanned by the predictor vectors can be partitioned into pure or nearly pure regions. In this context, a region is pure if all predictor vectors that belong to the region are members of a single group. Figure 9.1 is an example in which the space is nearly perfectly split by a straight line boundary. An interrelated factor affecting accuracy is the complexity of boundaries between pure, or nearly pure regions. Some prediction functions are defeated by complex boundaries, that is, boundaries that cannot be described by lines, planes, or other simple geometric objects. When the number of predictor variables is more than two or three, understanding $\mathcal{X}$ is difficult because visualizing $\mathcal{X}$ is difficult, and in most cases, trial and error is used to find a good prediction function.

In any case, it's necessary to analyze estimated accuracy to find a best prediction function among a set of candidate prediction functions. Once a best prediction function is selected, it's necessary to determine whether the accuracy of the prediction function is sufficient for the intended use. Let's consider a prediction function f that is used to detect rabies while the disease is treatable. The overall accuracy rate of the function is the probability that a prediction $y_0 = f(\mathbf{x}_0)$ is correct. But there's more to consider. Two errors can be made: predicting that a healthy individual is rabid, and predicting that an rabid individual is healthy. Both error rates are important and need to be estimated.

The standard parameters used to describe the accuracy of a prediction function are the unconditional accuracy rate and the conditional accuracy rates. The overall accuracy of a prediction function $f(\cdot|D)$ is the proportion of vectors $\mathbf{x}$ in a population, or vectors generated by a process, that are correctly labeled by $f(\cdot|D)$. This probability is unconditional, since it doesn't depend on the group membership y or the prediction of the group membership $\widehat{y}$. We express the unconditional accuracy rate as

$$\gamma = \Pr[f(\mathbf{x}_0) = y_0|D] = \Pr(\widehat{y}_0 = y_0|D),$$

where $\mathbf{z}_0 = (y_0, \mathbf{x}_0)$ is a randomly selected observation from the population of interest.

The accuracy rate for group i is the conditional probability that an observation belonging to group i is predicted to belong to group i. We express the conditional probability as

$$\alpha_i = \Pr(y_0 = \widehat{y}_0|y_0 = i).$$

The conditional probability that a prediction of membership in group j is correct is

$$\beta_j = \Pr(y_0 = \widehat{y}_0|\widehat{y}_0 = j).$$

In this case, we're conditioning on the outcome $\widehat{y}_0 = j$. This probability is of interest after computing the prediction $\widehat{y}_0 = j$ and we question how likely the prediction is to be correct. For example, a drug test is reported to be positive—should the test result be accepted without dispute or is it possible that the test is incorrect? The question can be addressed by estimating β_j.

The aforementioned probabilities must be estimated in almost all situations. Superficially, the accuracy estimators are simple. Let's assume that the original group labels have been translated to the labels $0, 1 \ldots, g-1$. Also, assume that a test set E is available and that the prediction function $f(\cdot|D)$ or a close approximation, say $f(\cdot|T)$, where $T \subset D$, is used to compute a prediction of y_0 for each $(y_0, \mathbf{x}_0) \in E$. The result is a collection of actual and predicted labels, say

$$L = [(y_0, \widehat{y}_0)|(y_0, \mathbf{x}_0) \in E]. \tag{9.9}$$

From these data, we may compute estimates of the accuracy rates γ, α_i, and β_j. The estimates are easily computed from the *confusion matrix.*

9.7.1 Confusion Matrices

The confusion matrix tabulates the number of test observations belonging to group i that are predicted to belong to group j, for $i = 0, 1, \ldots, g-1$, and $j = 0, 1, \ldots, g-1$. Rows of the matrix identify the actual group label and columns identify the predicted group. The entry in row i and column j is the number of occurrences of (i, j) among a collection of predictions for which the actual group labels are known. Table 9.1 shows the usual lay-out of a confusion matrix.

Table 9.1 A confusion matrix showing the results of predicting the group memberships of a set of test observations. The entry n_{ij} is the number of test observations belonging to group i and predicted to be members of group j. The term n_{i+} is the sum of row i, n_{+j} is the sum of column j, and n_{++} is the sum over all rows and columns

	Predicted group				
Actual group	0	1	$\cdots$	$g-1$	Total
0	n_{00}	n_{01}	$\cdots$	$n_{0,g-1}$	n_{0+}
1	n_{10}	n_{11}	$\cdots$	$n_{1,g-1}$	n_{1+}
$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$
$g-1$	$n_{g-1,0}$	$n_{g-1,1}$	$\cdots$	$n_{g-1,g-1}$	$n_{g-1,+}$
Total	n_{+0}	n_{+1}	$\cdots$	$n_{+,g-1}$	n_{++}

We suppose that a collection of actual and predicted pairs has been created as the collection L (Eq. (9.9)). From L, a confusion matrix has been constructed. Our estimators are defined using the counts contained in the confusion matrix (Table 9.1). The estimator of the overall accuracy rate γ is the proportion of correctly predicted test set observations; i.e.,

$$\widehat{\gamma} = \frac{\sum_{j=0}^{g-1} n_{jj}}{n_{++}},$$

where n_{++} is the number of pairs in L.

The group-specific accuracy rate for group i, α_i, is the probability that an observation *belonging* to group i is correctly classified. The estimator of α_i is the proportion of test observations belonging to group i that are predicted to belong to group i:

$$\widehat{\alpha}_i = \widehat{\Pr}(y_0 = \widehat{y} | y_0 = i) = \frac{n_{ii}}{n_{i+}}.$$

Similarly, the estimated probability that an observation *predicted* to belong to group j truly is a member of group j is

$$\widehat{\beta}_j = \widehat{\Pr}(y_0 = \widehat{y} | \widehat{y} = j) = \frac{n_{jj}}{n_{+j}}.$$

Table 9.2 provides an example of accuracy estimates from the digit recognition tutorial. Tabled values are estimates of the conditional probability that a prediction is correct given that the actual label is i ($\widehat{\alpha}_i$), or that the predicted label is j ($\widehat{\beta}_j$). From the table, we see that a prediction of the digit 8 is least likely to be correct since $\widehat{\beta}_8 = \widehat{\Pr}(y_0 = \widehat{y}|\widehat{y} = 8) = .883$ and that the digit most likely to be incorrectly identified is 1 since $\widehat{\alpha}_1 = \widehat{\Pr}(y_0 = \widehat{y}|y_0 = 1) = .912$.

Table 9.2 Estimated conditional accuracies obtained from the conventional eight-nearest neighbor prediction function using a training set of 37,800 observations and a test set of 4200 observations. The overall accuracy rate is estimated to be $\widehat{\gamma} = .958$

Digit	$\widehat{\alpha}_i = \widehat{\Pr}(y_0 = \widehat{y}\|y_0 = i)$	$\widehat{\beta}_i = \widehat{\Pr}(y_0 = \widehat{y}\|\widehat{y} = j)$
0	.973	.990
1	.912	.998
2	.986	.926
3	.949	.967
4	.978	.962
5	.953	.945
6	.968	.987
7	.934	.972
8	.994	.883
9	.950	.940

9.8 k-Nearest Neighbor Regression

The k-nearest neighbor prediction functions may be adapted for use with quantitative target variables without much trouble. For predicting quantitative variables, the k-nearest neighbor prediction functions are nonparametric alternatives to regression-based prediction functions. The term nonparametric describes methods that are not based on a model and so are free of parameters in contrast to parametric models. The objective is to predict the *quantitative* target variable y_0, and as before, we will use an associated predictor vector $\mathbf{x}_0$ and a function trained on a set of target and predictor variable pairs $D = \{(y_1, \mathbf{x}_1), \ldots, (y_n, \mathbf{x}_n)\}$. The prediction is a weighted average of the ordered training set targets $y_{[1]}, \ldots, y_{[n]}$ given by

$$\widehat{y}_0 = \sum_{i=1}^{n} w_i y_{[i]}, \tag{9.10}$$

where the ordered arrangement $y_{[1]}, \ldots, y_{[n]}$ is determined by the distances from $\mathbf{x}_0$ to $\mathbf{x}_1, \ldots, \mathbf{x}_n$. Formula (9.6) shows the weights used with the conventional k-nearest neighbor regression function and formula (9.8) defines the weights used with the exponentially weighted k-nearest neighbor regression function. Both weighting schemes are used in k-nearest-neighbor regression.

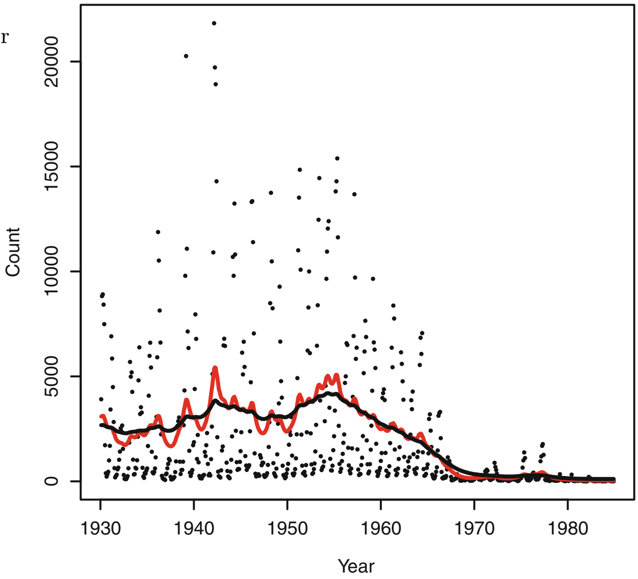


Fig. 9.3 Number of reported measles cases by month in California. The *red* and *black smooths* were obtained by setting $\alpha = .05$ and $\alpha = .02$, respectively

To illustrate, Fig. 9.3 shows the number of reported measles cases by month in California from 1930 to 1985 [63]. In addition to the monthly counts, we show the numbers of measles cases predicted by the exponentially weighted k-nearest neighbor regression function for two choices of the smoothing constant α. The predictions created with the choice of $\alpha = .05$ (in red) is much less smooth than the predictions created from $\alpha = .02$ (in black). These lines, which show the trend after removing much of the short term variation are often called *smooths*. There was an enormous amount of variation in the number of cases from 1930 to 1960. A measles vaccine licensed in 1963 accounts for the decline in cases to nearly zero by 1980. The red smooth suggests a several-year cycle in numbers of cases. The black smooth captures the long-term trend free of the several-year cycle.

The next tutorial applies the exponentially weighted k-nearest-neighbor regression function for forecasting.

9.9 Forecasting the S&P 500

In the tutorial that follows, the reader is guided through the development of a pattern recognition algorithm for 1-day-ahead forecasting of the S&P 500 daily price index. We use the term forecast when the prediction is of a target observed in the future. The S&P (Standard and Poor's) 500 index is a daily aggregation of the prices of 500 leading companies listed on either the New York Stock Exchange or the NASDAQ.[7] The targets to be predicted are the S&P 500 daily price indexes. The predictor vectors consist of index values from the 5 days preceding the target day. The predictor vectors will be referred to as *pattern vectors*. The idea behind the pattern recognition approach is that if two vectors $\mathbf{x}_i$ and $\mathbf{x}_j$ exhibit a similar pattern of day-to-day changes in the S&P index, then the following-day values s_{i+1} and s_{j+1} are likely to be similar. If this premise is accepted, then we'll collect a set of similar patterns and use an average of the following-day values to forecast s_{i+1}. We'll use exponentially weighted k-nearest-neighbor regression for building the prediction function. In essence, on the ith day, the algorithm searches the past pattern vectors for those that resemble the ith day pattern. Associated with each past pattern vector is the S&P index observed on the following day. The forecast is weighted average of the following-day S&P indexes of those patterns that are most similar to the ith day pattern.

There is a complication. The sequence of S&P 500 indexes varies substantially over the course of several years. Consequently, two 5-day sequences may exhibit a similar pattern of daily changes yet are relatively dissimilar according to a standard metric such as city-block distance because the mean level of the two sequences are different. For example, the day-to-day movements exhibited in one pattern about the mean level of 1000 may strongly resemble the movements in another pattern with a level of 1200, yet the city-block distance between the patterns will be large because of the differences in the mean levels. We would like to use one pattern to predict the next price index of the other, but they will not be close in distance. Even if we succeed in identifying them as similar, and use one to predict the other, the predictions may be inaccurate because the mean levels are different. A solution is to center every pattern by subtracting the mean of the pattern from all values in the pattern, in which case, the predictor vector consists of differences from the mean.[8] Then, the prediction $\widehat{y}_{i+1}$ is the predicted deviation from the mean. Adding $\widehat{y}_{i+1}$ to the mean of the ith pattern yields the forecast $\widehat{s}_{i+1}$.

9.10 Tutorial: Forecasting by Pattern Recognition

The objective of this tutorial is to program a pattern recognition algorithm for forecasting the S&P 500 price index. Using the algorithm, we'll compute 1-day-ahead forecasts and the plot the forecasts against day and compute a measure of agreement between actual and forecasted values.

[7] NASDAQ is the abbreviation for the National Association of Securities Dealers Automated Quotations system.

[8] Centering is a type of *detrending* [27].

Some new notation is needed. Let $s_1, \ldots, s_N$ denote the series of S&P 500 indexes. The index i implies chronological ordering.[9] As we aim to forecast the price index for day $i+1$ on the ith day, the forecasting target is s_{i+1}. We cannot use any data observed after the ith day for forecasting s_{i+1}.

We'll use all of the available S&P 500 index patterns created from the preceding days to forecast a target S&P 500 index, though. The target s_{i+1} is forecasted using the most recent pattern vector $\mathbf{s}_i$ as an input to a forecasting function built from all of the patterns preceding the ith pattern. The forecast is $\widehat{s}_{i+1} = f(\mathbf{s}_i|D_i)$, where D_i is a training set built on the ith day.

The length of the pattern vectors is p. We'll use $p = 5$ days to form the pattern vectors. Thus, a pattern vector $\mathbf{s}_i$ consists of the five S&P 500 index values preceding the ith day. Hence,

$$\mathbf{s}_i = [s_{i-p} \quad s_{i-p+1} \quad \cdots \quad s_{i-1}]^T, \text{ for } 0 < p < i.$$

The training set consists of data pairs of the form

$$\mathbf{z}_i = (s_i, \mathbf{s}_i) = (s_i, [s_{i-p} \quad s_{i-p+1} \quad \cdots \quad s_{i-1}]^T). \tag{9.11}$$

The training set built on day i for forecasting day s_{i+1} consists of the set of pairs

$$D_i = \{(s_{p+1}, \mathbf{s}_{p+1}), \ldots, (s_i, \mathbf{s}_i)\} = \{\mathbf{z}_{p+1}, \ldots, \mathbf{z}_i\}. \tag{9.12}$$

The first training pair that can be formed is $(s_{p+1}, \mathbf{s}_{p+1})$, and so the first training set is $D_{p+1} = \{(s_{p+1}, \mathbf{s}_{p+1})\}$. The first forecast that can be computed is $\widehat{s}_{p+2} = f(\mathbf{s}_{p+1}|D_{p+1})$. We'll see that $\widehat{s}_{p+2} = s_{p+1}$. The second forecast that can be computed is $\widehat{s}_{p+3}$ and it's a weighted average of s_{p+1} and s_{p+2}. In general, $\widehat{s}_{i+1} = f(\mathbf{s}_i|D_i)$ is a weighted average of $s_{p+1}, \ldots, s_i$ and the weights are determined by the similarity of the pattern vectors $\mathbf{s}_{p+1}, \ldots, \mathbf{s}_i$ to $\mathbf{s}_i$. The comparison of each past pattern to the predictor pattern $\mathbf{s}_i$ determines how the weights are assigned to the past patterns.

Because there's substantial trend in the S&P index values over several years, we'll forecast the detrended, or centered data. The data are centered by subtracting the mean value of each $\mathbf{s}_i$ from the elements of the $\mathbf{s}_i$. The mean of the vector $\mathbf{s}_i$ is

$$\overline{\mathbf{s}}_i = p^{-1} \sum_{j=i-p}^{i-1} s_j, \text{ for } 0 < p < i.$$

[9] By chronologically ordered, we mean that $i < j$ implies s_i was observed before s_j.

The (centered) predictor vector is

$$\mathbf{x}_i = \begin{bmatrix} s_{i-p} - \overline{\mathbf{s}}_i \\ s_{i-p+1} - \overline{\mathbf{s}}_i \\ \vdots \\ s_{i-1} - \overline{\mathbf{s}}_i \end{bmatrix} \text{ for } 0 < p < i.$$

The ith target value is also centered. We'll refer to the centered value as a *deviation*, and so the ith target deviation is

$$y_i = s_i - \overline{\mathbf{s}}_i.$$

Since $y_i = s_i - \overline{\mathbf{s}}_i$, we can express the S&P index value as $s_i = y_i + \overline{\mathbf{s}}_i$. The k-nearest-neighbor prediction function will compute forecasts of y_{i+1} from which the forecast of s_{i+1} is computed according to

$$\widehat{s}_{i+1} = \widehat{y}_{i+1} + \overline{\mathbf{s}}_i. \tag{9.13}$$

Let's amend the construction of the training sets, $D_{p+1}, \ldots, D_N$. It's helpful to incorporate the mean $\overline{\mathbf{s}}_i$ into the datum $\mathbf{z}_i$ so that the forecast $\widehat{s}_{i+1}$ can be readily constructed from the forecast $\widehat{y}_{i+1}$. So, the ith data triple is

$$\mathbf{z}_i = (y_i, \mathbf{x}_i, \overline{\mathbf{s}}_i)$$

and the forecasting function computes $\widehat{y}_{i+1} = f(\mathbf{x}_i|D_i)$. The centered training set D_i for the S&P index value forecast for day $i+1$ consists of triples $(y_j, \mathbf{x}_j, \overline{\mathbf{s}}_j)$, for $p+1 < j \leq i$ observed before day $i+1$. Thus,

$$\begin{aligned} D_{p+1} &= \{(y_{p+1}, \mathbf{x}_{p+1}, \overline{\mathbf{s}}_{p+1})\} = \{\mathbf{z}_{p+1}\}, \\ D_{p+2} &= \{\mathbf{z}_{p+1}, \mathbf{z}_{p+2}\}, \\ &\vdots \\ D_N &= \{\mathbf{z}_{p+1}, \mathbf{z}_{p+2}, \ldots, \mathbf{z}_N\}. \end{aligned} \tag{9.14}$$

Turning now to the k-nearest-neighbor regression function, the algorithm computes $\widehat{s}_{i+1} = f(\mathbf{x}_i|D_i)$ in three steps:

1. order the observations in the training set D_i according to the distances between the input pattern $\mathbf{x}_i$ and $\mathbf{x}_j$ for $\mathbf{z}_j \in D_i$;
2. compute the forecast of the centered value y_{i+1} as a weighted mean of the ordered targets in D_i; and,
3. translate the centered forecast to un-centered forecast $\widehat{s}_{i+1}$.

Turning to the first step, the distance between $\mathbf{x}_i$ and $\mathbf{x}_j$ is the city block distance

$$d_C(\mathbf{x}_i, \mathbf{x}_j) = \sum_{k=1}^{p} |x_{i,k} - x_{j,k}|,$$

where $x_{j,k}$ is the kth element of $\mathbf{x}_j$. With the distances $d_C(\mathbf{x}_i, \mathbf{x}_j), j = p + 1, \ldots, i$, the training targets are ordered and denoted as $y_{[1]}, \ldots, y_{[i-p]}$. The indexes on the ordered training targets denote the ordering (nearest, second-nearest, and so on), *not* the chronological index, which explains why they begin with 1 and end with $i - p$. The forecast of y_{i+1} is

$$\widehat{y}_{i+1} = \sum_{j=1}^{i-p} w_j y_{[j]},$$

where the w_j's are scaled weights originating from formula (9.8). The weights are scaled because they must sum to one, but a short sequence of r terms $\alpha(1-\alpha)^{j-1}, j = 1, \ldots, r$, will not sum to one.[10] The terms in Eq. (9.8) will have to be scaled. The scaled weights are

$$w_j = \frac{\alpha(1-\alpha)^{j-1}}{\sum_{k=1}^{i-p} \alpha(1-\alpha)^{k-1}}, j = 1, \ldots, i - p. \tag{9.15}$$

Because the targets have been centered, the final prediction is obtained from Eq. (9.13).

1. Download a series of S&P 500 indexes from the Federal Reserve Bank of St. Louis. The URL is http://research.stlouisfed.org/fred2/series/SP500.
2. We'll suppose that you have retrieved a file with a `txt` extension. Therefore, the observation day and value are tab-delimited. The following code will ingest the data file.

```
path = '../Data/SP500.txt'
s = []
with open(path, encoding = "utf-8") as f:
    f.readline()
    for string in f:
        data = string.replace('\n','').split('\t')
        if data[1] != '0':
            s.append(float(data[1]))
print(s)
```

   The instruction `data = string.replace('\n','').split('\t')` will remove the end-of-line marker (`'\n'`) and split the string at the tab. The S&P index value is appended to the list `s`. There are some zero values in the data series associated with holidays. These values are omitted from the data list $\mathbf{s} = [s_1, s_2, \ldots, s_N]$.
3. The next operation is to create and store the data tuples $\mathbf{z}_{p+1}, \ldots, \mathbf{z}_N$ where

[10] Precisely, $1 = \sum_{k=0}^{\infty} \alpha(1-\alpha)^k$. In practice, the sum is sufficiently close to 1 if the number of terms exceeds $100/\alpha$.

$$\begin{aligned} \mathbf{z}_i &= (y_i, \mathbf{x}_i, \overline{\mathbf{s}}_i) \\ &= \big(s_i - \overline{\mathbf{s}}_i, [s_{i-p} - \overline{\mathbf{s}}_i \quad \cdots \quad s_{i-1} - \overline{\mathbf{s}}_i]^T, \overline{\mathbf{s}}_i\big)\,. \end{aligned} \tag{9.16}$$

We need compute the means $\overline{\mathbf{s}}_{p+1}, \ldots, \overline{\mathbf{s}}_N$ and create the centered pattern vectors. To extract the appropriate elements from `s`, the index j is drawn from the sequence $i-p, i-p+1, \ldots, i-1$ using the `Numpy` function call `arange(i-p,i)`. The following `for` loop iterates over i. For each value of i, list comprehension is used to create the un-centered pattern vector as $\mathbf{u}_i$. Each $\mathbf{x}_i$ is created by centering $\mathbf{u}_i$:

```
p = 5
N = len(s)
for i in np.arange(p, N):
    u = [s[j] for j in np.arange(i-p, i)] # Using zero-indexing!
    sMean = np.mean(u)
    x = u - sMean
    z = (s[i] - sMean, x, sMean)

    if i == p:
        zList = [z]
    else:
        zList.append(z)
print(zList)
```

The triple created by the statement `z = (s[i] - sMean, x, sMean)` contains the centered target $y_i = s_i - \overline{\mathbf{s}}_i$, the centered pattern vector $\mathbf{x}_i$, and the uncentered pattern vector mean $\overline{\mathbf{s}}_i$.

4. Build a dictionary that contains the target and predictor pair $\mathbf{z}_i$ and the training set used to build the function for forecasting s_{i+1}, for each $i \in \{p+1, p+2, \ldots, N\}$. The training sets are shown in Eq. (9.14). Once, again, use a `for` loop that iterates over $i \in \{p+1, p+2, \ldots, N\}$. The data dictionary D_i is built by making a copy of D_{i-1} and creating a new entry using i as the key and $\mathbf{z}_i$ as the value. Add the new data set to the dictionary of data sets `D` using i as the key and the data set D_i as the value.

```
D = {}
for i in np.arange(p, N):
    if i == p:
        D[i] = [zList[i-p]] # Using zero-indexing.
    else:
        value = D[i-1].copy()
        value.append(zList[i-p])
        print(len(value))
        D[i] = value
```

Be sure to copy `D[i-1]` to `value` rather than assigning it with the '=' operator; otherwise, every dictionary entry will be `D[N-1]`. If you're using `Python` 2.7, then the instruction `D[i-1].copy()` is replaced with `D[i-1].copy.copy()` and the module `copy` must be imported.

5. With the data arranged in a convenient form, we will begin computing forecasts of y_{i+1} and s_{i+1}. Iterate over `D` and extract each data set beginning with D_{p+1}. Remove the test observation $\mathbf{z}_i$ from training set D_i.

```
for i in np.arange(p,N):
    data = D[i]  # Using zero-indexing.
    zi = data.pop()
```

The function `.pop()` removes the last value from the list `data` and stores it in the variable `zi`. We'll use the centered pattern stored in `zi` as an input to the forecasting function constructed from `data`.

6. From `zi`, extract y_i, $\mathbf{x}_i$ and $\overline{s}_i$ immediately after creating `zi` using the instruction `yi, xi, sMean = zi`.
7. Still within the `for` loop over `np.arange(p,N)`, create a dictionary[11] containing the training data for the prediction of y_{i+1}:

```
R = {j:z for j, z in enumerate(data)}
```

Note that `R` is created using *dictionary* comprehension.

8. Insert the `fOrder` function created in Tutorial 9.6 near the top of the script and before the data processing code segments.
9. Modify the function by replacing the instruction `nhbrs = [labels[j] for j in v]` with

```
nhbrs = [R[j][0] for j in v]
```

When the function executes, the sorted neighbors will be centered S&P index values sorted according to the similarity of test predictor $\mathbf{x}_i$ to the predictor vectors belonging to `R`.

10. Return to the `for` loop. On each iteration of the `for` loop and after building `R`, determine the sorted neighbors of $\mathbf{x}_i$ by calling `fOrder`. The call is

```
nhbrs = fOrder(R,xi)
```

The list `nhbrs` will contain the S&P 500 indexes sorted according to the distance between the target pattern vector $\mathbf{x}_i$ and the training pattern vectors $\mathbf{x}_{p+1}, \ldots, \mathbf{x}_{i-1}$.

[11] We will use the `fOrder` function programmed in the previous tutorial and it requires that the training set is a dictionary.

11. Set a value for the tuning constant α, say, $.2 < \alpha < .5$ before the `for` loop. Compute the list of weights for the exponentially weighted k-nearest-neighbor regression function. The weights must sum to one, so scale the values to sum to one. On every iteration of the `for` loop, n_R increases and `wts` must be recomputed.

```
s = sum([alpha*(1 - alpha)**j for j in range(nR)])
wts = [(alpha/s)*(1 - alpha)**j for j in range(nR)]
```

12. Compute the predictions immediately after computing `wts`.

```
deviation = sum(a*b for a, b in zip(wts, nhbrs))
sHat = deviation + sMean
print(yi + sMean,’   ’,sHat)
```

The print statement prints the observed and predicted S&P index values.

13. Trace the performance of the prediction function. Initialize a list to store the squared errors, say `sqrErrors = []` and another list to store the plotting data (`plottingData`). After computing a forecast, store the squared differences between the centered forecasts and centered observations.

```
sqrErrors.append((yi - deviation)**2)
plottingData.append([i,yi + sMean,sHat])
print(np.sqrt(sum(sqrErrors)/len(sqrErrors)))
```

The last statement prints the current estimate of root mean square error.

14. Plot a set of 300 predictions and observations against day.

```
import matplotlib.pyplot as plt
day = [day for day, _ , _ in plottingData if 1000 < day < 1301]
y = [yobs for day, yobs , _ in plottingData if 1000 < day < 1301]
plt.plot(day, y)
sHat = [hat for day, _ , hat in plottingData if 1000 < day < 1301]
plt.plot(day, sHat)
plt.show()
```

The observed values will be plotted in blue.

15. Compute a measure of relative forecasting accuracy by comparing the mean squared difference between the actual and forecasted values to the sample variance.[12] The measure of forecasting accuracy is the adjusted coefficient of determination

[12] Recall that the sample variance is, essentially, the mean squared difference between the observations and the sample mean.

$$R^2_{\text{adjusted}} = \frac{\widehat{\sigma}^2 - \widehat{\sigma}^2_{\text{kNN}}}{\widehat{\sigma}^2} = 1 - \frac{\widehat{\sigma}^2_{\text{kNN}}}{\widehat{\sigma}^2}.$$

We'll use the Numpy functions mean() and var() to compute the mean and standard deviation of the target values:

```
yList = [yobs for _, yobs ,_ in plottingData ]
rAdj = 1 - np.mean(sqrErrors)/np.var(yList)
print(rAdj)
```

9.10.1 Remark

The adjusted coefficient of determination is very large ($R^2_{\text{adjusted}} > .99$) in the S&P 500 forecasting problem, a result that begs for explanation. The S&P index series exhibits a great deal of variation over long periods, say, several years, and the average change within a week is *much* smaller (Fig. 9.4). The long term variation produces a large sample variance $\widehat{\sigma}^2$ since the sample variance is the average squared difference between the actual values and the series mean. As a baseline forecasting function, the sample mean sets a very low bar. It's much better to use a smaller set of past values for forecasting.

A better baseline against which to compare the k-nearest-neighbor regression forecasting function is the sample mean of most recent past p days. This forecast is sometimes called a moving average because it changes (moves) as time advances. The moving average forecast accounts for long-term trend as the moving average means will reflect long-term trend.

We computed $\widehat{\sigma}^2_{\text{MV}}$, the mean squared difference between the targets and the sample mean of the preceding $p = 10$ values. With $\widehat{\sigma}^2_{\text{MV}}$, we then computed the relative reduction in mean square error

$$\frac{\widehat{\sigma}^2_{\text{MV}} - \widehat{\sigma}^2_{\text{kNN}}}{\widehat{\sigma}^2_{\text{MV}}}. \tag{9.17}$$

The relative reduction in mean square forecasting error was .562, and so it can be said that the k-nearest-neighbor regression prediction function reduced the mean square forecasting error by 56.2% relative to the moving average forecasting function. Exercise 9.5 asks the reader to compute $\widehat{\sigma}^2_{\text{MV}}$.

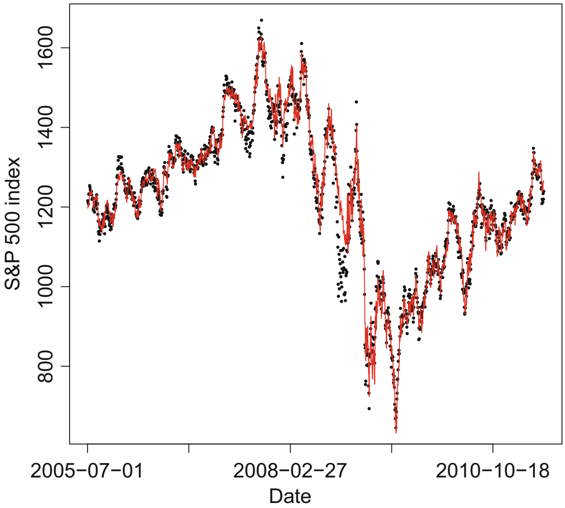


Fig. 9.4 S&P 500 indexes and the exponentially weighted k-nearest neighbor regression predictions plotted against date. The exponentially weighted k-nearest neighbor regression was formulated with $\alpha = .25$ using $p = 30$ past values as predictors

9.11 Cross-Validation

Let's take a closer look at accuracy assessment. As usual, let $D=\{(y_1, \mathbf{x}_1), \ldots, (y_n, \mathbf{x}_n)\}$ denote the data set. The usual approach to accuracy assessment begins by creating a test set E and training set R. Suppose that $R \cap E \neq \emptyset$. This situation presents a potentially serious problem—some observations (those in $R \cap E$) are being used for both building the prediction function and estimating the error of the prediction function. The resulting accuracy estimates often are optimistically biased. Accuracy estimates are called *apparent* or *plug-in* estimates in the worst case of $D = R = E$. The extent of bias is difficult to determine without an involved analysis.

There is a somewhat unique case in which the consequence of using the same observations in both sets can be seen with no trouble. Consider the one-nearest neighbor prediction function, and suppose that $\mathbf{x}_0 \in R \cap E$. Suppose that there are no ties whatsoever and we investigate the accuracy of $f(\mathbf{x}_0|R)$ using the test observations in E. Since $\mathbf{x}_0 \in R$, $\mathbf{x}_0$ will be closest to itself and the nearest neighbor label will be its own label. The prediction of y_0 is certain to be correct, and every observation in $R \cap E$ will be correctly predicted. We learn nothing of how the function will perform given $\mathbf{x}_0 \notin R$ from these test observations. Even worse, the accuracy estimates will be optimistically biased because the prediction function fits the test data better than a completely unrelated or independent data set. This phenomenon is known as *over-fitting*.

Over-fitting can be eliminated by insuring that no observation in R is used as a test observation. Suppose then that random or systematic sampling has produced E and R such that $D = R \cup E$ and $R \cap E = \emptyset$. In other words, E and R form a partition of D. We'll approximate the prediction function $f(\cdot|D)$ that we intend to use by the function $f(\cdot|R)$ and apply $f(\cdot|R)$ to every test observation in E. The result is a set of predictions that are free of over-fitting bias. This process is called *validation* accuracy assessment. We used it in the digit recognition and S&P tutorials.

Occasionally, D is too small for validation accuracy assessment because removing a sufficiently large test set results in $f(\cdot|R)$ being a poor approximation of $f(\cdot|D)$. Then, the accuracy of $f(\cdot|R)$ will be significantly less than the accuracy of $f(\cdot|D)$. Furthermore, when the sample is small, there will be a substantial degree of variation among accuracy estimates computed from different test sets. It would be wise not to trust the results derived from a single random partitioning of D.

The k-fold cross-validation algorithm is a solution to these accuracy estimation problems.

A cross-validation algorithm withdraws a set of observation pairs E from the training sample D to serve as a test set. The remaining observations in R are used to construct a prediction function $f(\cdot|R)$. Then $f(\cdot|R)$ is applied to each test set vector $\mathbf{x}_0 \in E$ to produce a prediction. The result is a set $L = \{(y_0, f[\mathbf{x}_0|R])|(y_0, \mathbf{x}_0) \in E\}$ from which a confusion matrix and estimates of the accuracy parameters γ, α_i, and β_j can be computed. Since the held-out sample was not used in constructing the prediction function, over-fitting bias has been eliminated.

So far, nothing is new. What is new with cross-validation is that not one test set is drawn from D, but instead k, where $1 < k \leq n$. The hold-out test sets are $E_1, \ldots, E_k$, and the training sets are $R_k = D \cap E_k^c$. The outcomes of predicting $\mathbf{x}_0 \in E_i$ are collected in the set $L_i = \{(y_0, f[\mathbf{x}_0|R_i])|(y_0, \mathbf{x}_0) \in E_i\}$. After all test sets have been processed, the confusion matrix is computed from $L = \cup_{i=1}^k L_i$. Now the training sets R_i can be much closer to D in size and the prediction functions $f(\cdot|R_i)$ will be good approximations of $f(\cdot|D)$. Finally, the accuracy estimates will be good because L is large (and provided that D is representative of the population or process to which the prediction function will be applied).

It's best if $E_1, \ldots, E_k$ form a partition of D so that every observation is a target exactly once.[13] Formally, a *k-fold* cross-validation algorithm partitions D as k disjoint subsets each containing approximately n/k observations. There are a variety of ways to implement the algorithm. For example, each observation $i = 1, 2, \ldots, n$, may be assigned a random number drawn from $\{1, 2, \ldots, k\}$ that identifies membership in one of the subsets $E_1, \ldots, E_k$. The cross-validation algorithm iterates over $i \in \{1, 2, \ldots, k\}$. On the ith iteration, E_i is the test set and $R_i = D \cap E_i^c$ is the training set. The prediction function $f(\cdot|R_i)$ is constructed and used to predict y_0 for each $\mathbf{x}_0 \in E_i$. The results

[13] A lazy alternative is to draw independent random samples $E_1, \ldots, E_k$. A test observation may appear in more than one test set. This presents no risk of bias, but there's less information gained on the second and subsequent predictions of a test observation.

are collected in L_i. Accuracy estimates are computed from $L = \cup_i L_i$ at the completion of the k iterations. A commonly used value for k is 10. Another popular choice is $k = n$.

As the cross-validation accuracy estimates will depend on the initial partitioning of D, some number of repetitions of the algorithm may be carried out using different random partitions. Sufficiently many repetitions will reduce the variation associated with random partitioning to a negligible fraction of the estimate. At completion, the estimates from different repetitions are aggregated in the construction of the confusion matrix.

9.12 Exercises

9.12.1 Conceptual

9.1. Show that for $0 < \alpha < 1$, $1 = \sum_{i=0}^{\infty} \alpha(1-\alpha)^i$.

9.2. Consider the exponentially weighted k-nearest-neighbor prediction function. Argue that the prediction of y_0 will be $y_{[1]}$ for any choice of $\alpha \geq .5$, and hence, the exponentially weighted k-nearest neighbor prediction function is equivalent to the conventional one-nearest neighbor prediction function.

9.3. Suppose that the conventional k-nearest neighbor prediction function is used for predicting group membership in one of $g = 2$ groups. Compute the possible values of $\widehat{\Pr}(y_0 = l|\mathbf{x}_0)$ for $k \in \{1, 3, 5\}$ where l is the label of one of the groups. Comment on the accuracy of $\widehat{\Pr}(y_0 = l|\mathbf{x}_0)$ as an estimator of $\Pr(y_0 = l|\mathbf{x}_0)$ for small values of k.

9.12.2 Computational

9.4. Revisit instruction 15 of the Sect. 9.6 tutorial. Modify the function `fPred` to break ties if more than one value is the maximum among the values in `counts`. Proceed as follows. After computing the conventional k-nearest-neighbor prediction, test whether there is more than one count in `counts` equal to the maximum. If so, then enlarge the neighborhood to include $k+1$ neighbors and recompute the maximum and the prediction. Test for ties. Repeat the process of enlarging the neighborhood until the tie is broken.

9.5. Return to the S&P prediction problem and compute the moving average mean squared error using the most recent p observations

$$\widehat{y}_{\mathrm{MV},i} = p^{-1} \sum_{j=i-p+1}^{i} y_j,$$

at time step i, for $i = p + 1, \ldots, N$. As pointed out in Sect. 9.10.1, a more informative measure of forecasting error compares $\widehat{\sigma}^2_{\mathrm{MV}}$ (Eq. (9.17)), the mean square prediction error associated with the moving average forecasts, to $\widehat{\sigma}^2_{\mathrm{kNN}}$.

a. Verify that .562 is the relative reduction in forecasting error.
b. Use `ggplot` to plot a set of 300 observations on S&P 500 index against day as points. Add two more layers, one showing the k-nearest-neighbor regression forecasts as a line and another showing the moving average forecasts as a line. Distinguish between the two sets of forecasts using color.

9.6. Use the Wisconsin breast cancer data set, a small training set consisting of observations obtained from biopsies on benign and malignant breast cancer tumors.[14] You may retrieve it from the University of California Irvine Machine Learning Repository https://archive.ics.uci.edu/ml/datasets/. The problem posed by the breast cancer data is to predict the type of tumor from a vector of measurements on tumor cell morphology.

Compute apparent and *leave-one-out* cross-validation accuracy estimates for neighborhood sizes $k \in \{1, 2, 4, 8, 16\}$ and using the conventional k-nearest-neighbor prediction function. Use the `R` function `knn` for computing apparent accuracy estimates and `knn.cv` for cross-validation accuracy estimates. `knn` and `knn.cv` are part of the `class` package and the `BreastCancer` data set is part of the package `mlbench`. Construct a summary table of the form exhibited by Table 9.3. Comment on the relationship between overfitting bias and k.

9.7. The tutorial of Sect. 9.10 discussed a k-nearest-neighbor prediction function for predicting the S&P 500 index 1 day ahead.

a. Modify the algorithm so that the predictions are made several days ahead. Predict y_{i+d}, for each combination of $d \in \{1, 3, 5\}$ and $\alpha \in \{.05, .1, .3\}$ using exponentially weighted k-nearest-neighbor regression. Build a table showing the estimated root mean square error $\widehat{\sigma}_{\mathrm{kNN}}$ for each combination of α and d.
b. Investigate the effect of different pattern lengths p on the estimated root mean square error $\widehat{\sigma}_{\mathrm{kNN}}$. Try $p \in \{5, 10, 20\}$.

[14] The Wisconsin breast cancer data set is widely used as a machine learning benchmark data set.

Table 9.3 Apparent and cross-validation accuracy estimates for the k-nearest-neighbor prediction function. The differences between apparent estimates and cross-validation estimates are attributable to over-fitting bias

	Accuracy estimate	
k	Apparent	Cross-validation
1	$\widehat{\gamma}_1^{\text{apparent}}$	$\widehat{\gamma}_1^{\text{cv}}$
2	$\widehat{\gamma}_2^{\text{apparent}}$	$\widehat{\gamma}_2^{\text{cv}}$
$\vdots$	$\vdots$	$\vdots$
16	$\widehat{\gamma}_{16}^{\text{apparent}}$	$\widehat{\gamma}_{16}^{\text{cv}}$

Chapter 10
The Multinomial Naïve Bayes Prediction Function

Abstract The naïve Bayes prediction function is a computationally and conceptually simple algorithm. While the performance of the algorithm generally is not best among competitors when the predictor variables are quantitative, it does well with categorical predictor variables, and it's especially well-suited for categorical predictor variables with many categories. In this chapter we develop the *multinomial* naïve Bayes prediction function, the incarnation of naïve Bayes for categorical predictors. We develop the function from its mathematical foundation before applying it to two very different problems: predicting the authorship of the Federalist Papers and a problem from the business marketing domain—classifying shoppers based on their grocery store purchases. The Federalist Papers application provides the opportunity to work with textual data.

10.1 Introduction

Consider prediction problems in which the predictor variables are categorical. For instance, suppose that customers that shop at a membership grocery store are to be classified into one of several groups based on buying habits at the time of a sale. The training data may consist of a rather long list of purchased items, each of which may be identified by a category such as grocery, hardware, electronics, and so on. The number of categories may be large, for instance, 1102 items are used for prediction in the tutorial of Sect. 10.5. The computational demands preclude using k-nearest-neighbor prediction functions when the training samples and numbers of items are large and an alternative approach is necessary. There are a variety of possibilities, though one stands alone with respect to algorithmic simplicity—the multinomial naïve Bayes prediction function. Before developing the mathematics behind the algorithm, we discuss a different application of multinomial naïve Bayes.

B. Steele et al., *Algorithms for Data Science*,
DOI 10.1007/978-3-319-45797-0_10

10.2 The Federalist Papers

The Federalist Papers are a collection of 85 essays written by James Madison, Alexander Hamilton, and John Jay arguing for the ratification of the United States Constitution. The Federalist Papers go beyond the legal framework laid out in the relatively short Constitution and expound upon the foundations and principles supporting the Constitution. At the time of publication, between October 1787 and August 1788, the authors of the Federalist Papers were anonymous. All were signed with the pseudonym Publius. Authorship of most of the papers were revealed some years later by Hamilton, though his claim to authorship of 12 papers were disputed for nearly 200 years. Table 10.1 shows the authors and the papers attributed to each. Analyses carried out by Adair [1] and Mosteller and Wallace [41] in the mid-twentieth century attributed authorship of the disputed papers to James Madison and the question presumably has been put to rest. In this chapter, we take another look at the disputed papers.

Table 10.1 Authors of the Federalist papers

Author	Papers
Jay	2, 3, 4, 5, 64
Madison	10, 14, 37–48
Hamilton	1, 6, 7, 8, 9, 11, 12, 13, 15, 16, 17, 21–36, 59, 60, 61, 65–85
Hamilton and Madison	18, 19, 20
Disputed	49–58 62, 63

Assigning authorship to the disputed 12 papers is a problem appropriate for the multinomial naïve Bayes prediction function. The process of predicting authorship centers around building a prediction function trained on undisputed papers. The prediction function may be applied to a disputed paper in two steps. First, the paper is mapped to a vector of predictor variables. The second step produces a prediction of authorship by estimating the probabilities that each of the candidate authors wrote the paper. The author is predicted to the candidate author whose estimated probability of being the author is largest among the three. Our analysis suggests that Madison was not the author of all 12 disputed papers and that Hamilton was author of six of the disputed papers.

A significant amount of preliminary data processing is necessary to transform a Federalist Paper to an appropriate unit for predictive analysis. In this context, each of the 73 Federalist papers of undisputed authorship represents an observational unit with a known label (the author). Each paper is mapped to a predictor vector consisting of quantitative attributes. The attributes are identified with two qualities in mind. First, the mapping $g : P_0 \to \mathbf{x}_0$ of paper P_0 to predictor vector $\mathbf{x}_0$ must be well-defined—there can be only one possible value for a particular paper. Secondly, the attributes of $\mathbf{x}_0$ ought to be useful for discriminating among authors. Quantifiable stylistic attributes such as sentence length and most frequently used words are appropriate provided that the three authors differ with respect to the attributes.

The mapping of a passage of text to a vector of quantitative variables is one of several challenging steps in algorithmic extraction of information from textual data [6]. We dedicate the first of two tutorials to the process. The second tutorial builds a prediction function by exploiting differences in word choice among the authors. Before tackling the data, the multinomial naïve Bayes prediction function is developed in the next section.

10.3 The Multinomial Naïve Bayes Prediction Function

The problem is set up as follows. The class membership of a target $\mathbf{z}_0 = (y_0, \mathbf{x}_0)$ is y_0 and $\mathbf{x}_0$ is a concomitant predictor vector. A prediction function $f(\cdot|D)$ is to be built from D, the training set of observation pairs. The prediction of y_0 is $\widehat{y}_0 = f(\mathbf{x}_0|D)$. What is novel is that $\mathbf{x}_0$ is a vector of counts. Each element of $\mathbf{x}_0$ records the number of times that a particular type, category, or level of a qualitative variable was observed. For instance, $x_{0,j}$ may denote the number of times that word w_j appeared in a disputed Federalist paper P_0. The vector $\mathbf{x}_0$ consists of the frequencies of each of n different words in P_0.

We'll continue to develop the multinomial naïve Bayes prediction function in the context of the authorship attribution problem with the tacit understanding that the algorithm generalizes straightforwardly to other situations. Exercise 10.3 provides the reader with the opportunity to do so as it involves predicting the type of shopping trip made by a Walmart shopper based on the types of items purchased (groceries, hardware, etc.).

Let $x_{0,j}$ denote the number of times that word w_j appeared in paper P_0. Then, $\mathbf{x}_0 = [x_{0,1} \ \cdots \ x_{0,n}]^T$ contains the word frequencies for paper P_0. There are n different words across all of the Federalist Papers and these words are collected as a set W.[1] The author of P_0 is denoted by y_0, where $y_0 \in \{\text{Hamilton}, \text{Jay}, \text{Madison}\} = \{A_1, A_2, A_3\}$. The probability that word w_j is drawn at random from P_0 is

$$\pi_{k,j} = \Pr(w_j \in P_0 | y_0 = A_k). \tag{10.1}$$

The probability $\pi_{k,j}$ is specific to the author but does not vary among papers written by the author. Since w_j is certain to be a member of W, $1 = \sum_j^n \pi_{k,j}$. The probabilities of occurrence for w_j may vary among authors, hence, $\pi_{1,j}$

[1] Standard practice is to omit very common words such as prepositions and pronouns from W.

may differ from $\pi_{2,j}$ and $\pi_{3,j}$. If the differences are substantial for more than a few words, then the prediction function will be able to discriminate among authors given a vector of word frequencies $\mathbf{x}_0$.

If there were only two words used among all three authors, and their appearances in a paper were independent, then the probability of observing a particular vector of frequencies, say $\mathbf{x}_0 = [25 \quad 37]^T$, could be calculated using the binomial distribution:

$$\Pr(\mathbf{x}_0|y_0 = A_k) = \frac{(25+37)!}{25!37!}\pi_{k,1}^{25}\pi_{k,2}^{37}.$$

If this notation troublesome, recall that the binomial random variable computes the probability of observing x successes in n trials. The variable x counts the number of successes. The probability of observing x successes and $n-x$ failures is

$$\Pr(x) = \frac{n!}{x!(n-x)!}\pi^x(1-\pi)^{n-x}, \tag{10.2}$$

for $x \in \{0, 1, \ldots, n\}$. Formula 10.2 uses the fact that $\pi_2 = 1 - \pi_1$. If the authorship of P_0 is disputed, then based on the word frequency vector $\mathbf{x}_0 = [25 \quad 37]^T$, we would predict that the author is A^* where $\Pr(\mathbf{x}_0|y_0 = A^*)$ is largest among $\Pr(\mathbf{x}_0|y_0 = A_1), \Pr(\mathbf{x}_0|y_0 = A_2)$, and $\Pr(\mathbf{x}_0|y_0 = A_3)$. The $\pi_{k,i}$'s usually must be estimated. If so, we substitute probability estimates in the calculation.

A precise definition of the prediction function is

$$\widehat{y}_0 = f(\mathbf{x}_0|D) = \arg\max\{\widehat{\Pr}(\mathbf{x}_0|y_0 = A_1), \widehat{\Pr}(\mathbf{x}_0|y_0 = A_2), \widehat{\Pr}(\mathbf{x}_0|y_0 = A_3)\}.$$

For the author attribution problem, the number of words is much greater than two and the probability calculation requires an extension of the binomial distribution. The multinomial probability function provides the extension as it accommodates $n \geq 2$ words. As with the binomial distribution, if it were true that the occurrences of the words in a paper were independent events, then the probability of observing a frequency vector $\mathbf{x}_0$ is given by the multinomial probability function

$$\Pr(\mathbf{x}_0|y_0 = A_k) = \frac{(\sum_{i=1}^n x_{0,i})!}{\prod_{i=1}^n x_{0,i}!}\pi_{k,1}^{x_{0,1}} \times \cdots \times \pi_{k,n}^{x_{0,n}}. \tag{10.3}$$

The leftmost term involving factorials is known as the multinomial coefficient, just as the term $n!/[x!(n-x)!]$ is known as the binomial coefficient. We'll see below that the multinomial coefficient can be ignored in the calculation of the prediction function.

10.3.1 Posterior Probabilities

The multinomial naïve Bayes prediction function improves on the method just discussed by accounting for prior information. In this situation, there are 70 papers of undisputed authorship, 51 of which were written by Alexander Hamilton, 14 by James Madison, and 5 by John Jay. If there were a tie among the probabilities obtained from formula (10.3), then based on the larger numbers of papers written by Hamilton, we are inclined to attribute authorship to Hamilton. Naïve Bayes quantifies the tyranny of the majority logic. Prior to observing the word frequencies, the prior probabilities of authorship are estimated to be $\Pr(y_0 = A_1) = 51/70$, $\Pr(y_0 = A_2) = 14/70$, and $\Pr(y_0 = A_3) = 5/70$. In fact, if we had no information besides the numbers of papers written by each then we would predict that Hamilton was the author of any and all disputed papers because the estimated prior probability that Hamilton wrote P_0 is largest among the three estimates. The use of the term *prior* stems from assigning some probability of authorship prior to observing the information contained in the frequency vector $\mathbf{x}_0$.

Bayes formula brings together the information contained in the word frequencies and the prior information by computing the posterior[2] probability of authorship

$$\Pr(y_0 = A_k|\mathbf{x}_0) = \frac{\Pr(\mathbf{x}_0|y_0 = A_k)\Pr(y_0 = A_k)}{\Pr(\mathbf{x}_0)}.$$

The predicted author is the author with the largest posterior probability. Specifically, the predicted author is

$$\begin{aligned}\widehat{y}_0 &= \arg\max_k\{\Pr(y_0 = A_k|\mathbf{x}_0)\}\\ &= \arg\max_k\left\{\frac{\Pr(\mathbf{x}_0|y_0 = A_k)\Pr(y_0 = A_k)}{\Pr(\mathbf{x}_0)}\right\}\end{aligned}$$

The denominator $\Pr(\mathbf{x}_0)$ scales each of the posterior probabilities by the same quantity and doesn't affect the relative ordering of the posterior probabilities. Therefore, a simpler and more practical expression for the prediction function is

$$\widehat{y}_0 = \arg\max_k\left\{\Pr(\mathbf{x}_0|y_0 = A_k)\Pr(y_0 = A_k)\right\}.$$

The multinomial coefficient depends only on $\mathbf{x}_0$ and so takes on the same value for every A_k. Therefore, it may be ignored in the calculation $\Pr(\mathbf{x}_0|y_0 = A_k)$ if we are only interested in determining which posterior probability is largest.

The multinomial probabilities (the $\pi_{k,j}$'s) are unknown but easily estimated. We'll drop the multinomial coefficient and write the multinomial naïve Bayes function in terms of the estimated probabilities:

[2] After having seen the data.

$$
\begin{aligned}
\widehat{y}_0 &= \arg\max_k\{\widehat{\Pr}(y_0 = A_k|\mathbf{x}_0)\}\\
&= \arg\max_k\left\{\widehat{\pi}_{k,1}^{x_{0,1}} \times \cdots \times \widehat{\pi}_{k,n}^{x_{0,n}} \times \widehat{\pi}_k\right\},
\end{aligned}
\tag{10.4}
$$

where $x_{0,j}$ is the frequency of occurrence of w_j in the target paper P_0.

The probability estimate $\widehat{\pi}_{k,j}$ is the relative frequency of occurrence of w_j across all papers written by author A_k. Lastly, $\widehat{\pi}_k = \widehat{\Pr}(y_0 = A_k)$ is the estimated prior probability that the paper was written by A_k. Once $\mathbf{x}_0$ is observed, we use the information carried by $\mathbf{x}_0$ to update the prior probabilities in the form of posterior probabilities. If the analyst has no information regarding priors, she should use the *non-informative* priors $\pi_1 = \cdots = \pi_g = 1/g$, assuming that there are g classes (authors). In this application of the multinomial naïve Bayes prediction function, the prior probability π_k is estimated by the proportion of undisputed papers that have been attributed to A_k.

The application of formula (10.4) requires estimates for the probabilities $\pi_k, \pi_{k,j}, j = 1, 2, \ldots, n$, and $k = 1, 2$, and 3. We use the sample proportions or empirical probabilities:

$$
\widehat{\pi}_{k,j} = \frac{x_{k,j}}{n_k},
\tag{10.5}
$$

where $x_{k,j}$ is the number of instances of w_j in the papers written by author A_k and $n_k = \sum_j x_{k,j}$ is the total of all word frequencies for author A_k. Arithmetic underflow may occur when the some of the exponents in formula (10.4) are large and the bases are small. Underflow is avoided by identifying the author with the largest *log-posterior* probability since

$$
\arg\max_k\{\widehat{\Pr}(y_0 = A_k|\mathbf{x}_0)\} = \arg\max_k\left\{\log\left[\widehat{\Pr}(y_0 = A_k|\mathbf{x}_0)\right]\right\}.
$$

The final version of the multinomial naïve Bayes prediction function is

$$
\begin{aligned}
\widehat{y}_0 &= \arg\max_k\left\{\log\left[\widehat{\Pr}(y_0 = A_k|\mathbf{x}_0)\right]\right\}\\
&= \arg\max_k\left\{\log(\widehat{\pi}_k) + \textstyle\sum_{j=1}^n x_{0,j}\log(\widehat{\pi}_{k,j})\right\}.
\end{aligned}
\tag{10.6}
$$

The following tutorial instructs the reader in creating the first of two `Python` scripts for the Federalist Papers authorship attribution problem. The task of attributing authorship requires a considerable amount of data processing and coding because of the nature of the data: a single unstructured text file containing all 85 papers. The file will be read and stored as one long character string from which the individual papers are extracted. Each paper is then split into words. Many words are regarded as useless for natural language processing.[3] These words are referred to as *stop-words*. They consist of ubiquitous words such as prepositions (e.g., the, and, at) and simple verbs (e.g., is, be, go). A reduced set of words is created by discarding

[3] A sub-area of linguistics devoted to using machines to extract information from text.

the stopwords and keeping only those words that have been used at least once by each author. The last step of the reduction produces word frequency distributions for each paper and for each author.

The reader will program an algorithm for assigning authorship to a disputed paper using the multinomial naïve Bayes prediction function in the second tutorial. The prediction function compares the word frequency distribution of the disputed paper to each author's word frequency distribution. Authorship is predicted based on the similarity of the frequency distribution of a disputed paper to each author's frequency distribution. Therefore, word frequency distributions must be computed for each paper and for each author using the undisputed papers. Building the word frequency distributions is the next subject.

10.4 Tutorial: Reducing the Federalist Papers

Several mappings are needed to transform the textual data to a form compatible with the naïve Bayes prediction function. When applied one after the other, the mappings form the composite mapping discussed in Sect. 10.2. Specifically, $g : P_0 \rightarrow \mathbf{x}_0$ maps a Federalist paper P_0 to a word frequency distribution $\mathbf{x}_0$. The tutorial begins with retrieving two data sets.

1. Retrieve an electronic version of the Federalist Papers from the Gutenberg project [44], https://www.gutenberg.org/. Use the search facility to search for *Federalist Papers*. Several versions are available; this tutorial has been developed using the plain text version `1404-8.txt`.
2. Table 10.1 lists the presumptive authors of each paper. A file (`owners.txt`) containing the paper numbers and authors as shown in Table 10.1 may be obtained from the textbook website. The 12 disputed are papers 49 through 58 and 62 and 63. In `owners.txt`, the disputed papers are attributed to Madison. Retrieve the file or create a facsimile from Table 10.1.
3. Create a set of stopwords using the `stop-words` module. Install the module by submitting the instruction `pip install stop-words` from within a Linux terminal or a Windows command prompt window. Add the following instructions to your `Python` script to import the module and save the English stop-words as a set.

```
from stop_words import get_stop_words
stop_words = get_stop_words(’en’)
stopWordSet = set(stop_words) | set(’ ’)
print(len(stopWordSet))
```

We'll prevent the blank string `’ ’` from becoming a word in the word frequency distributions by adding it to the stop-words.

4. Build a dictionary from `owners.txt` that identifies the author of each Federalist paper. Read the file as a sequence of character strings. Remove the end-of-line or new-line character `’\n’` from each string.

```
paperDict = {}
path = ’../Data/owners.txt’
with open(path, "rU") as f:
    for string in f:
        string = string.replace(’\n’, ’ ’)
        print(string)
```

   The first instruction initializes a dictionary `paperDict` for storing the authors and paper numbers.

5. Split each string at the comma and use the first element of the list (the paper number) as a dictionary key in `paperDict` and the second element (the author) as the value.

```
key, value = string.split(’,’)
paperDict[int(key)] = value
```

   The code segment must have the same indentation as the print instruction.

6. Read the text file containing the Federalist papers and create a single string.

```
path = ’../Data/1404-8.txt’
sentenceDict = {}
nSentences = 0
sentence = ’’
String = ’’
with open(path, "rU") as f:
    for string in f:
        String = String + string.replace(’\n’,’ ’)
```

   The end-of-line marker `\n` is replaced with a white space before extending the string.

7. The next code segment builds a dictionary that contains each Federalist paper as a list of words. The key will be a pair—a two-tuple consisting of the paper number and the presumed author. The principal difficulty in building the dictionary is to form the papers. To do this, it’s necessary to determine where each paper begins and ends.

   The phrase *To the People of the State of New York* opens each paper. We’ll use the phrase to identify the beginning of a paper. The pseudonym *PUBLIUS* marks the end of all papers except 37. Open the file `1404-8.txt` in an editor and insert *PUBLIUS* at the end of Paper 37 if it is missing.

The program iterates over `String` and determines beginning and ending positions of each paper in `String`. We'll use the `re.finditer` function, so import the regular expression module at the top of the script (`import re`). The `finditer` function returns an iterator so that all instances of a sub-string in a string can be located. The following code segment creates a dictionary in which the keys are the paper number and the values are lists containing the position of the first and last character in the paper. We iterate over `String` twice—once to find the starting position of each paper and a second time to find the ending position.

```
positionDict = {}
opening  = ’To the People of the State of New York’
counter = 0
for m in re.finditer(opening, String):
    counter+= 1
    positionDict[counter] = [m.end()]

close  = ’PUBLIUS’
counter = 0
for m in re.finditer(close, String):
    counter+= 1
    positionDict[counter].append(m.start())
```

In the first iteration, we search `String` until the starting position is found as the ending character in the phrase *To the People of the State of New York*. Then, `counter` is incremented and the position marking the start of the paper is stored in `positionDict`. Another iteration begins the search for the next starting position. Iterations continue until the end of `String` is found.

The second `for` loop determines the end of the paper by finding the position of the first character in *PUBLIUS*.

8. Now extract each paper from `String` by extracting the text between each starting and ending position. Iterate over the dictionary `positionDict` to get the beginning and ending positions. Also, create a label consisting of the paper number and the author's name.

```
wordDict = {}
for paperNumber in positionDict:
    b, e = positionDict[paperNumber]
    author = paperDict[paperNumber]
    label = (paperNumber,author)
    paper = String[b+1:e-1]
```

The last statement stores the extracted text as a sub-string named `paper`.

9. There’s a substantial amount of bookkeeping in the mapping of papers to word frequency distributions. To expedite matters, we use a `namedtuple` type for dictionary keys. The elements of a named tuple may be addressed using a name rather than a positional index thereby improving readability of the code and reducing the likelihood of programming errors. The `namedtuple` type is contained within the `collections` module. Import `namedtuple` by putting an import instruction at the beginning of the `Python` script. Create a type of tuple named `identifier` with two elements, a class name and field identifiers. The class name will be label and the field identifiers are index and author. The import and initialization instructions are

```
from collections import namedtuple
from collections import Counter
identifier = namedtuple(’label’, ’index author’)
```

   We are importing the dictionary subclass `Counter` to expedite computing the word frequency distributions.
10. In the code segment above (instruction 8), replace `label = (paperCount, author)` with

```
label = identifier(paperCount,author)
```

   This change allows us to reference the author of a particular paper using `label.author` or by position, i.e., `label[1]`.
11. We’ll add the instructions necessary to convert the string `paper` to a set of words. Three operations are carried out: remove punctuation marks, translate each letter to lowercase, and remove the stop-words from the paper. Still in the `for` loop and after extracting `paper` from `String`, add the code segment

```
for char in ’.,?!/;:-"()’:
    paper = paper.replace(char,’’)
paper = paper.lower().split(’ ’)
for sw in stopWordSet:
    paper = [w for w in paper if w != sw]
```

   The instruction `paper = paper.lower().split(’ ’)` translates capital letters to lowercase. Then, the string is by split into sub-strings wherever a blank is encountered. The result is a list.

   The `for` loop over `stopWordSet` uses list comprehension to repeatedly build a new copy of `paper`. On each iteration of the `for` loop, a stop word is removed from `paper`.

12. The last step is to generate the word distributions. We use a Counter dictionary subclass from the collections module. The function call Counter(paper) returns a dictionary in which the keys of the dictionary are items in the list (words in this case) and the values are the frequencies of occurrence of each item in the argument (the paper in this case).

    ```
    wordDict[label] = Counter(paper)
    print(label.index,label.author,len(wordDict[label]))
    ```

    The last statement prints the paper number and author and the number of words in the word frequency distribution for the paper. Both statements must be indented to execute within the for loop.
13. Construct a table showing the numbers of papers attributed to each author.

    ```
    table = dict.fromkeys(set(paperDict.values()),0)
    for label in wordDict:
        table[label.author] += 1
    print(table)
    ```

    You should find that Jay is identified as the author of 5 papers, Madison is identified as the author of 26 papers, and Hamilton is identified as the author of 51 papers.
14. Since some of the papers are of disputed authorship and three are joint, we should not use all 85 for building the prediction function.[4] The next code segment constructs a list trainLabels containing the labels of papers that will be used to build the prediction function. Specifically, these papers will be used to build word frequency distributions for Hamilton, Madison, and Jay.

    ```
    skip = [18, 19, 20, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 62, 63]
    trainLabels = []
    for label in wordDict:
        number, author = label
        if number not in skip:
            trainLabels.append(label)
    print(len(trainLabels))
    ```

    The list skip contains the numbers of the three co-authored papers and the 12 disputed papers.

[4] The disputed papers are in fact the targets though we've used an ownership list that attributes authorship to all 85 papers.

15. Build a list of words that were used in the training set papers and by all of the authors.[5] We refer these words as the common words. The first operation is to create a list of training papers.

```
disputedList = [18,19,20,49,50,51,52,53,54,55,56,57,58,62,63]
trainLabels = []
for label in wordDict:
    number, author = label
    if number not in disputedList:
        trainLabels.append(label)
print(len(trainLabels))
```

    The 85 dictionaries of word frequencies contain words that were not used by all authors. This will cause a difficulty when comparing distributions. If a word is not used by Hamilton, say, then the estimated probability of its use by Hamilton is zero since the probability estimate is the relative frequency of use. Probability estimates of zero have severe consequences, and we discuss the matter at length in the tutorial of Sect. 10.6.1. One solution is to modify all of the probability estimates so that none can take on the value zero. A second solution is to eliminate words that are not used by all three authors. We proceed with the second solution.

16. To construct the list of common words, we begin by building a dictionary named `usedDict` that uses words as keys. The value associated with a word is the *set* of authors that used the word. The list of common words—we call it `commonList`—is built from those words in the dictionary that were used by all three authors.

    Iterate over the training set, specifically, iterate over `trainLabels`.

```
usedDict = {}
for label in trainLabels:
    words = list(wordDict[label].keys())
    for word in words:
        value = usedDict.get(word)
        if value is None:
            usedDict[word] = set([label.author])
        else:
            usedDict[word] = value | set([label.author])
commonList = [word for word in usedDict if len(usedDict[word]) == 3]
```

    The instruction `wordDict[label].keys()` extracts the words used in the paper indexed by `label`. The `|` operator is the set function union (hence, `A | B` $= A \cup B$). The list `commonList` is built by list comprehension. The test for three authors in the set insures that words included in the common list have been used by all three authors.

[5] Stop-words are excluded.

17. Remove any word appearing in `wordDict` that is not common to the three authors. Items cannot be removed from an object while iterating over the object. To work around this constraint, we create a new dictionary named `newDict` for each paper in `wordDict` and fill it with words from the dictionary associated with the paper. After iterating over all of the words in the frequency distribution associated with a particular paper, we replace the longer dictionary with the shortened, common words only dictionary.

```
for label in wordDict:
    D = wordDict[label]
    newDict = {}
    for word in D:
        if word in commonList:
            newDict[word] = D[word]
    wordDict[label] = newDict
    print(label,len(wordDict[label]))
```

Not every paper has the same length word frequency distribution. However, every word appearing in a word frequency distribution has been used by all three authors. Therefore, the probability estimates $\widehat{\pi}_{k,i}$ are non-zero for every author (indexed by k) and every word (indexed by i).

10.4.1 Summary

The Federalist papers have been mapped to a set of word frequency distributions stored in the dictionary `wordDict`. Each distribution lists the non-zero frequency of occurrence words in a common set of 1102 words. The frequency of each word in the common set is easily determined for a particular paper of interest, and so we are now ready to construct the multinomial naïve Bayes prediction function.

10.5 Tutorial: Predicting Authorship of the Disputed Federalist Papers

We proceed under the premise that stylistic differences in the authors' writing may be used to predict the author of a disputed paper. We use one stylistic attribute for this purpose: word choice. Word choice is quantified by the relative frequency of occurrence of each word in the set of common words. We hope that there are substantial differences among authors with respect to the relative frequencies of occurrence of a significant number of words. If so, then we should be able to construct an accurate naïve Bayes prediction function for the purpose of predicting authorship of a Federalist paper. To build the prediction function, we compute, for each author, the relative frequency of occurrence of each word in the common set. The relative frequencies are our estimates of the probability of observing a particular word in a random draw from a Federalist paper written by a particular author (Eq. (10.5)).

Given a disputed paper and its prediction vector $\mathbf{x}_0$ consisting of the frequencies of occurrence of words in the common set, we compute the estimated probability of observing $\mathbf{x}_0$ if the author was Jay, Madison, and then Hamilton. Our prediction of the author y_0 is determined by the largest estimated posterior probability $\widehat{\Pr}(y_0 = \text{Jay}|\mathbf{x}_0)$, $\widehat{\Pr}(y_0 = \text{Madison}|\mathbf{x}_0)$, and $\widehat{\Pr}(y_0 = \text{Hamilton}|\mathbf{x}_0)$.

It remains to estimate the probability $\pi_{k,i}$ that the common set word w_i will be randomly sampled from among the non-stop-words comprising a Federalist paper written by author A_k, for $k = 1, 2, 3$.

1. For each of the three authors, a frequency distribution will be created in the form of a dictionary in which the keys are words and the values are the frequencies of occurrence of each word. This will be accomplished by summing the frequencies of occurrence of a particular word over all undisputed papers that are attributed to the author. The code segment begins by counting the number of papers attributed to each author. These counts yield the estimated prior probabilities of authorship.

```
logPriors = dict.fromkeys(authors,0)
freqDistnDict = dict.fromkeys(authors)
for label in trainLabels:
    number, author = label
    D = wordDict[label]
    distn = freqDistnDict.get(author)
    if distn is None:
        distn = D
    else:
        for word in D:
            value = distn.get(word)
            if value is not None:
                distn[word] += D[word]
            else:
                distn[word] = D[word]
    freqDistnDict[author] = distn
    logPriors[author] +=1
```

The dictionary `freqDistnDict` contains a dictionary for each the three authors. The value associated with a particular author is a frequency dictionary, hence, each key is a word and the associated value is the frequency of occurrence of the word.

2. Before constructing the prediction function compute the relative frequency distributions for each author. A relative frequency distribution is constructed as a dictionary. Each key is a word and the associated value is the log-relative frequency of occurrence of the word (identified as $\log(\widehat{\pi}_{k,j})$ in formula (10.6)). The relative frequency of occurrence of a word is computed as the number of times that an author used the word in his undisputed papers divided by total number of uses of any common word across his undisputed papers. The dictionary `distnDict` will contain the estimated priors and estimated probabilities of word occurrence for the three authors.

```
nR = len(trainLabels)
logProbDict = dict.fromkeys(authors,{})
distnDict = dict.fromkeys(authors)
for author in authors:
    authorDict = {}
    logPriors[author] = np.log(logPriors[author]/nR)

    nWords = sum([freqDistnDict[author][word] for word in commonList ])
    print(nWords)

    for word in commonList:
        relFreq = freqDistnDict[author][word]/nWords
        authorDict[word] = np.log(relFreq)

    distnDict[author]  = [logPriors[author], authorDict]
```

The total number of uses of common words by an author is computed as `nWords`. We iterate over the words in `commonList` to compute $\log(\widehat{\pi}_{k,j})$ for the jth word and author k. The estimate is stored temporarily in `authorDict` using word as a key. After completing the iteration over `commonList`, `authorDict` is stored with the estimated log-prior probability in `distnDict`.

3. The next code segment applies the multinomial naïve Bayes prediction function to the undisputed papers to gain some information about the accuracy of the prediction function. Recall that a predictor vector $\mathbf{x}_i = [x_{i,1} \ \cdots \ x_{i,n}]^T$ is the vector of frequencies of occurrence for each of the n words in paper i. The prediction of authorship is

$$f(\mathbf{x}_i|R) = \arg\max_k \left\{ \log(\widehat{\pi}_k) + \sum_{j=1}^{n} x_{i,j} \log(\widehat{\pi}_{k,j}) \right\}, \tag{10.7}$$

where n is the number of common words. The terms $\log(\widehat{\pi}_k)$ and $\log(\widehat{\pi}_{k,j})$ for the three authors are stored in the dictionary `distnDict` and the test vector $\mathbf{x}_j$ is stored in a dictionary in `wordDict`. Therefore, we treat `wordDict` as a test set, and apply the prediction function to each $\mathbf{x}_i$ in this test set by iterating over `wordDict`. None of the papers in the list `skip` should be used since we are excluding disputed papers from accuracy assessment. Iterate over papers in `wordDict` and exclude disputed and joint authorship papers:

```
nGroups = len(authors)
confusionMatrix = np.zeros(shape = (nGroups,nGroups))

skip = [18,19,20,49,50,51,52,53,54,55,56,57,58,62,63]

for label in wordDict:
    testNumber, testAuthor = label

    if testNumber not in skip:
        xi = wordDict[label]
        postProb = dict.fromkeys(authors,0)
        for author in authors:
            logPrior, logProbDict = distnDict[author]
            postProb[author] = logPrior
                + sum([xi[word]*logProbDict[word] for word in xi])

        postProbList = list(postProb.values())
        postProbAuthors = list(postProb.keys())
        maxIndex = np.argmax(postProbList)
        prediction = postProbAuthors[maxIndex]
        print(testAuthor,prediction)
```

The estimated log-posterior probabilities are computed for $\mathbf{x}_i$ according to

$$\log(\widehat{\pi}_k|\mathbf{x}_i) = \log(\widehat{\pi}_k) + \sum_{j=1}^{n} x_{i,j} \log(\widehat{\pi}_{k,j}). \tag{10.8}$$

The predicted author of the paper corresponds to the largest of the three log-posterior probability estimates $\log(\widehat{\pi}_{\text{Hamilton}}|\mathbf{x}_i)$, $\log(\widehat{\pi}_{\text{Madison}}|\mathbf{x}_i)$, and $\log(\widehat{\pi}_{\text{Jay}}|\mathbf{x}_i)$. We've converted the dictionary keys and values to lists and used the `Numpy` function `argmax` to extract the index of the largest log-posterior probability estimate. We've used it before in instruction 15 of the Sect. 9.6 tutorial (Chap. 9).

4. Tabulate the result in the confusion matrix. Rows of the confusion matrix correspond to the known author and columns correspond to the predicted author. Insert the following code segment so that it executes immediately after the statement `print(testAuthor, prediction)`.

```
i = list(authors).index(testAuthor)
j = list(authors).index(prediction)
confusionMatrix[i,j] += 1
```

5. Print the confusion matrix and the estimated overall accuracy at the completion of the `for` loop.

```
print(confusionMatrix)
print('acc = ',sum(np.diag(confusionMatrix))/sum(sum(confusionMatrix)))
```

   You should find that the prediction function has an apparent accuracy rate of 1.
6. Determine the predictions of the disputed papers by changing the skip list so that only the joint papers are excluded from the calculation of accuracy. The outcome is shown in Table 10.2.

Table 10.2 A confusion matrix showing the results of predicting the authors of the Federalist papers. The actual authors consist of the known authors and the predicted authors according to researchers [1, 41]. Predicted authors are the outcomes of the multinomial naïve Bayes prediction function

	Predicted author			
Actual author	Jay	Madison	Hamilton	Total
Jay	5	0	0	5
Madison	0	20	6	26
Hamilton	0	0	51	51
Total	5	20	57	82

10.5.1 Remark

Our results disagree with the results of Mosteller and Wallace [41] as we find that six of the disputed papers were more likely authored by Hamilton as he claimed.

10.6 Tutorial: Customer Segmentation

A recurring challenge in business marketing is to identify customers that are alike with regard to purchasing habits. If distinct groups or *segments* of customers can be identified, then a business may characterize their customers with respect to demographics or behavior. Tailoring their communications to specific segments allows for more relevant messages and improvements in customer experience. From the standpoint of the business, monitoring activity by customer segment promotes the understanding of their customers and allows for timely changes and improvements to business practices. A group of customers that are alike with respect to purchasing habits is referred to as a *customer segment*. The process of partitioning a set of customers into segments is known as *customer segmentation*.

We return to the grocery store data set of Chap. 5. The investigation was loosely organized about understanding shopping behaviors by customer segment. There are clear differences between segments with respect to several variables.

The following customer segments have been identified:

Primary These members appear to utilize the co-op as the primary place they grocery shop.

Secondary These members shop regularly at the co-op but are likely to shop at other grocery stores.

Light These members have joined the co-op but shop very seldom and tend to purchase few items.

Niche This is a collection of 17 segments containing relatively few members. These customers shop one department almost exclusively. Examples of the niche segments are *produce*, *packaged grocery*, and *cheese*.

Both members and non-members buy at the co-op. Non-members are not identifiable by segment at the time of a transaction. However, the co-op would like to provide incentives and services at the time of sale to the non-members based on their predicted customer segment. With this goal in mind, we ask the following question: based on the receipt, that is, a list of purchased items, can we identify a customer segment that the non-member customer most resembles?[6] In analytical terms, the objective is to estimate the relative likelihood that a non-member belongs to each of the 20 customer segments based on their point-of-sale receipt. The last step is to label the non-member according to the most likely segment.

We return to a problem already encountered in the Federalist Papers analysis before tackling the prediction problem.

10.6.1 Additive Smoothing

The incidence of non-stop words that were not used by all three authors presented a complication in the Federalist Papers problem. Some of the non-stop words were not used by all three authors, and so the frequency of occurrence of such a word was zero for at least one author. For these words, the estimated probability of use for one or more authors was zero. Here lies the problem. The prediction function computes a linear combination of the frequencies of use and the logarithms of the estimated probability of use. If one such word appears in a paper, then an estimate of zero for author A eliminates A from

[6] If so, then the non-members may be offered segment-specific information and incentives.

being the author of the paper, no matter how similar the rest of the paper's word distribution is to the word distribution of author A. We had the luxury with the Federalist Papers of ignoring those zero-frequency words since there were more than 1100 words that were used by all three. With only three authors, enough common words were left to build an accurate prediction function. If there had been a larger number of authors, then ignoring words not common to all authors may have been costly with respect to the discriminative information carried by the words.

In the customer segmentation analysis, a group is a customer segment. There are thousands of items for sale (at least 12,890), and most of the segments are niche segments. Members of the niche segments are very exclusive in their selection of store items and some niche segments may not purchase a significant number of items. Our casual approach to dismissing items with zero frequency of purchase may negatively affect the prediction function.

Let's consider a more conservative approach to handling categories (a word or a store item) for which there are no observations on use (or purchase) for one or more the groups. The solution is to smooth the probability estimates. We recommend a technique known as additive smoothing (also known as Laplace smoothing). The idea is to add terms to both the numerator and denominator of a sample proportion so that if the numerator is zero, then the estimate will not be zero, but instead a small positive fraction.

Let d denote the number of groups or classes, and n denote the number of categories. In the Federalist Papers analysis, there were $d = 3$ groups of papers, each belonging to one of the authors, and $n = 1103$ categories, as there were 1103 non-stop words used at least once by all three authors. The probability of drawing item j from among the items bought by a customer belonging to segment k is denoted by $\pi_{k,j}$. The sample proportion estimator of $\pi_{k,j}$ is

$$p_{k,j} = \frac{x_{k,j}}{\sum_{i=1}^{n} x_{k,i}},$$

where $x_{k,j}$ is the frequency of purchases of item j by members of segment k. The sum of $p_{k,j}$ over items (indexed by j) is one since every purchase must be one of the n items. We found 12,890 unique items among the purchases of the members. The smoothed estimator of $\pi_{k,j}$ is a modification of the sample proportion given by

$$\widehat{\pi}_{k,j} = \frac{x_{k,j} + \alpha}{\sum_{i=1}^{n} x_{k,i} + d\alpha}, \tag{10.9}$$

where α is the smoothing parameter. A common choice is $\alpha = 1$ though smaller values are sometimes preferred. To illustrate, suppose that $\alpha = 1$ and $x_{k,j} = 0$ for some segment and item. Then, $\widehat{\pi}_{k,j} = 1/(\sum_{i=1}^{n} x_{k,i} + d)$. On the other hand, if every purchase from a segment k was of item j, then $x_{k,j} = \sum_{i=1}^{n} x_{k,i}$ and $\widehat{\pi}_{k,j} = (x_{k,j} + 1)/(x_{k,j} + d)$.

10.6.2 The Data

Returning to the data at hand, there are two data sets, one consisting of receipts from members and the other consisting of receipts from non-members. Both data sets were collected during the month of December 2015. The file named `member_transactions.txt` has contains 50,193 observations, each of which is best thought of as a receipt. Columns are tab-delimited and contain the following information:

1. **owner**: an anonymized identifier of the co-op member.
2. **transaction identifier**: a unique identifier for the transaction, or equivalently, the receipt.
3. **segment**: the segment membership of the owner.
4. **date**: the date of the transaction.
5. **hour**: the hour of the transaction.
6. **item list**: a variable-length list of the items that were purchased.

Table 10.3 shows a partial record. Note that one item, a bag of St. Paul bagels, appears twice in the list of items.

Table 10.3 A partial record from the data file. The entire receipt is not shown since 40 items were purchased

Variable	Value
Owner	11,144
Transaction identifier	m22
Segment	Primary
Date	2015-12-30
Hour	12
Item list	Tilapia `\t` St.Paul Bagels (Bagged) `\t` St.Paul Bagels (Bagged) `\t` Salmon Atlantic Fillet `\t` O.Broccoli 10oz CF `\t` . . .

A second file, `nonmember_transactions.txt`, contains transaction information for non-member customers. The non-members data file has the same columns as the previous data set except that the owner and customer segment identifiers are absent.

1. We begin by importing modules and defining the paths to the two data sets.

```
import sys
import numpy as np
from collections import defaultdict

working_dir = '../Data/'
mem_file_name = "member_transactions.txt"
non_mem_file_name = "nonmember_transactions.txt"
```

2. Initialize a dictionary of dictionaries named `segmentDict` to store the items that were purchased by the members of each segment. Initialize two more dictionaries to store the numbers of items purchased by each segment and the number of purchases of each item.

```
segmentDict = defaultdict(lambda: defaultdict(int))
segmentTotals = defaultdict(int) # A count of purchases by segment
itemTotals = defaultdict(int) # A count of purchases by item
```

   The instruction `defaultdict(lambda: defaultdict(int))` creates a dictionary of dictionaries. The outer dictionary is named `segmentDict`. The keys for this dictionary will be the groups—the customer segments. The inner dictionaries are not named, but the keys of these dictionaries are the item names and the values are the frequencies of occurrence of each item in the segment stored as integers. Because the dictionaries are `defaultdict` dictionaries, we do not need to test for and create an entry for each item if it’s not already in the inner dictionary. Likewise, the inner dictionaries (the segment dictionaries) are automatically created when a newly encountered segment is passed to `segmentDict`.
3. The member transactions data file is read in this code segment. The code builds `segmentDict`, the dictionary containing the frequencies of occurrence of each item in each segment. The total number of transactions for each segment is computed and stored in `segmentTotals`. The totals are used to estimate the prior probabilities. Lastly, the totals for each item are computed and stored in `itemTotals`.

```
path = working_dir + mem_file_name
print(path)
with open(path,’r’) as f :
    next(f) # skip header
    for record in f :
        record = record.strip().split("\t")
        segment = record[2]
        items = record[5:]
        segmentTotals[segment] += 1

        for item in items :
            item = item.lower()
            itemTotals[item] += 1
            segmentDict[segment][item] += 1
print(len(segmentDict))
print(segmentDict.keys())
```

   As each record is read, leading and trailing blanks are removed using the `.strip()` function.

The instruction `segmentDict[segment][item] += 1` increments the frequency of occurrence of the `item` in the segment dictionary. The `item` is a key for a particular segment dictionary, namely, `segmentDict[segment]`.

4. After all of the data is processed, calculate the natural logarithms of the prior probability estimates (the $\widehat{\pi}_k$'s) (Eq. (10.6)).

```
totalItems = sum([n for n in segmentTotals.values()])
logPriorDict = {seg : np.log(n/totalItems)
                     for seg, n in segmentTotals.items()}
nSegments = len(logPriorDict)
```

The calculation of `logPriorDict` makes use of dictionary comprehension. Segments are used as keys and the values are the natural logarithms of the prior probability estimates. Check that the number of segments is 20.

5. The next step is to build the prediction function, $f(\mathbf{x}_0|D)$. The function will consume a predictor vector $\mathbf{x}_0$ consisting of a receipt listing each item that was purchased by a customer. The prediction of y_0 is computed from the posterior probability estimates (the $\widehat{\pi}_{k,j}$'s) which in turn are computed from `segmentTotals`. We also use the prior probability estimates $\widehat{\pi}_k, k = 1, 2, \ldots, d$. The logarithms of these estimates are passed in as well.

   The function declaration is

```
def predictFunction(x0, segmentTotals, logPriorDict, segmentDict, alpha):
    ''' Predicts segment from x0
        Input: x0: a transaction list to classify
               segmentTotals: the counts of transactions by segment
               logPriorDict: the log-priors for each segment
               segmentDict: transaction by segment by item.
               alpha: the Laplace smoothing parameter.
    '''
```

6. Continuing with the `predictFunction`, the next code segment computes the logarithm of the posterior probability estimates of segment membership for the target $\mathbf{x}_0$. Recall that the prediction of group (or segment) membership is determined by the function (see Eq. (10.7))

$$f(\mathbf{x}_0|D) = \arg\max_k \left\{ \log(\widehat{\pi}_k) + \sum_{j=1}^{n} x_{0,j} \log(\widehat{\pi}_{k,j}) \right\},$$

where k indexes customer segments, $\widehat{\pi}_k$ is the estimated prior probability of membership in the kth segment, and $\widehat{\pi}_{k,j}$ is the estimated probability of purchasing the jth item by a customer belonging to segment k. Also, the logarithm of $\widehat{\pi}_{k,j}$ is (see Eq. (10.9))

$$\log(\widehat{\pi}_{k,j}) = \log(x_{k,j} + \alpha) - \log\left(\textstyle\sum_{i=1}^{n} x_{k,i} + d\alpha\right), \tag{10.10}$$

where $x_{k,j}$ is the total number of purchases of the jth item by members of customer segment k. The denominator $\sum_{i=1}^{n} x_{k,i} + d\alpha$ of the estimator $\widehat{\pi}_{k,i}$ is a constant—it doesn't depend on the item (indexed by j) so it can be computed once for each segment (indexed by k). Since the prediction function operates on the natural logarithm scale, we compute the logarithm of the denominator.

Be aware that customers often purchase several of the same items, in which case one might expect the count would be greater than one in $\mathbf{x}_0$. However, our item list is different: the item will have multiple occurrences in the list if the item was scanned more than once (see Table 10.3 for an example of a receipt).

We use nested `for` loops. The outer `for` loop iterates over the segments and the inner iterates over items in the transaction vector $\mathbf{x}_0$.

```
logProbs = defaultdict(float)
for segment in segmentDict :
    denominator = sum([itemTotals[item] for item in itemTotals])
                      + alpha * nSegments
    logDen = np.log(denominator)
    for item in x0 :
        logProbs[segment] += np.log(segmentDict[segment][item] + alpha)
                             - logDen
```

Because `segmentDict[segment]` is a `defaultdict`, it returns zero if there is no entry for `item` in `segmentDict[segment]` rather than a `keyError`.

Even if we pass $\alpha = 0$ and `segmentDict[segment][item]` is zero, an exception is not created. Instead, the `np.log` function returns `-inf`, mathematically, $-\infty$. The practical effect is that the segment without any purchases of the item cannot yield the maximum probability of segment membership since the estimated log-posterior probability of membership in the segment will be $-\infty$. This strategy is reasonable if the number of observations in the data set is large, as it is in this case with 50,192 transactions.

7. The last code segment in the `predictFunction` determines the segment associated with the maximum estimated posterior probability (on the logarithmic scale). The first line of code creates a list of two-element lists from the `logProbs` dictionary. The `Numpy` function `argmax` is used to extract the index of the largest element.[7]

[7] We used the function in instruction of 3 of the Sect. 10.5 tutorial.

```
    lstLogProbs = [[seg, logProb] for seg, logProb in logProbs.items()]
    index = np.argmax([logProb for _,logProb in lstLogProbs])
    prediction, maxProb = lstLogProbs[index]
    return(prediction)
```

The calculation of `index` uses list comprehension to build a list containing only the log-probability estimates. The underscore character `_` instructs the `Python` interpreter to ignore the first element of each pair in building the list.

8. Now that the prediction function has been programmed, we need to verify that it produces reasonable results. The first check is to apply the function to the membership data and determine the proportion of training observations that are assigned to the correct segment. We'll re-use the code from the beginning of the tutorial to process the data file. We create an empty list named `outcomes` to store the pairs $(y_i, \widehat{y}_i)$ as the `with open` loop iterates over records and computes the predictions.

```
outcomes = []
alpha = 0
with open(working_dir + mem_file_name,’r’) as f :
    next(f) # skip headers
    for idx, record in enumerate(f) :
        record = record.strip().split("\t")
        segment, item_list = record[2], record[5:]
        x0 = [item.lower() for item in item_list]
```

9. Call `predictFunction` and save the results by appending to the `outcomes` list:

```
prediction = predictFunction(x0, segmentTotals, logPriorDict,
                                segmentDict, alpha)
outcomes.append([segment,prediction])
```

This code segment immediately follows the translation of `item_list` to $\mathbf{x}_0$.

10. Every 100 records, compute an estimate of the overall accuracy of the function.

```
if idx % 100 == 0 :
    acc = np.mean([int(segment==prediction )
                   for segment, prediction in outcomes])
    print(str(idx),acc)
```

List comprehension is used to construct a list of boolean values (true or false) by the comparison `segment==prediction`. The conversion of a boolean to integer produces 1 if the boolean variable is true and 0 if the value is false. The mean of the binary vector is the proportion of observations that were correctly classified. With our choice of $\alpha = 0$, we obtained an estimated accuracy rate of .711.

11. The nonmember data set contains 23,251 receipts. We will compute predictions for each receipt and track the number of receipts that are predicted to belong to each of the 20 market segments.

    a. Change the input file name to read the non-members data set.
    b. Change the instruction that extracts the `item_list` and the record identifier. The new instruction should look like this:

```
recordID, item_list = record[0], record[3:]
```

    c. Initialize a `defaultdict` for which the values are sets before the non-members data file is processed. The keys of the dictionary will be segments, and the values will be a set of record identifiers that were predicted to belong to the segment.

```
segCounts = defaultdict(set)
```

    d. After `predictionFunction` returns `prediction`, add the record identifier to the dictionary. The key is the predicted segment.

```
segCounts[prediction].add(recordID)
```

    The function `add` adds a record identifier to a set.
    e. Run the script and print the contents of `segCounts`. Our results are shown in Table 10.4.

```
for seg in segCounts :
    total += len(segCounts[seg])
    print(" & ".join([seg,str(round(len(segCounts[seg])/idx,3))] ) )
```

10.6.3 Remarks

We have some concerns with the results of the prediction function. First, the accuracy estimate is not a great deal larger than the proportion of members that have been identified as members of the primary segment (.649). If we had

Table 10.4 Predicted customer segments for non-members. The tabled values are the proportion of predictions by customer segment. $N = 23{,}251$ receipts were classified to segment

Segment	Proportion
Primary	.873
Secondary	.077
Light	.0
Niche-juice bar	.003
Niche-frozen	.0
Niche-supplements	.0
Niche-meat	.001
Niche-bread	.0
Niche-personal care	.001
Niche-herbs & spices	.001
Niche-general merchandise	.0
Niche-beer & wine	.0
Niche-packaged grocery	.014
Niche-produce	.007
Niche-bulk	.001
Niche-cheese	.0
Niche-refrigerated grocery	.002
Niche-deli	.02

assigned every member to the primary segment, the accuracy rate estimate would be .649. We are not out-performing a simple baseline prediction function by a lot. Another item of some concern is the rate at which non-members are assigned to the primary segment—.873. This rate is substantially larger than the proportion of primary segment customers (.649). This observation raises a second concern that the prediction function is biased and assigns too many receipts to the primary segment. The problem is difficult to address and it may be that non-members tend to purchase items similar to those preferred by the primary segment customers in which case, the prediction function is not necessarily biased.

10.7 Exercises

10.7.1 Conceptual

10.1. The sample proportions were used as estimators of the conditional probabilities of membership $\pi_{k,i}, k = 1, 2, \ldots, d, i = 1, \ldots, n$ without justification for the multinomial prediction function. It ought to be established that the sample proportions are good estimators. We may instead determine a vector of estimators

$$\widehat{\boldsymbol{\pi}}_k = [\widehat{\pi}_{k,1} \ \cdots \ \widehat{\pi}_{k,n}]^T \tag{10.11}$$

that maximize the probability of obtaining the observation vector $\mathbf{x} = [x_1 \quad \cdots \quad x_n]^T$. The estimator $\widehat{\boldsymbol{\pi}}_k$ is optimal in the sense of making the estimates as consistent with the data as possible, and so we would choose to use $\widehat{\boldsymbol{\pi}}_k$ over any other estimator. Determine an estimator of $\boldsymbol{\pi}_k$ by maximizing the probability of obtaining $\mathbf{x}$ (formula (10.3)) as a function of $\boldsymbol{\pi}_k = [\pi_{k,1} \quad \cdots \quad \pi_{k,n}]^T$. For convenience, you may drop the subscript k.

10.2. Consider the two-word problem and prediction of authorship based on $\mathbf{x} = [25 \quad 37]^T$ (Eq. (10.2)). Suppose that $\pi_{1,1} = .5$, $\pi_{2,1} = .3$, and $\pi_{3,1} = .4$.

a. Compute $\Pr(\mathbf{x}_0|y_0 = A_k)$ for A_1, A_2 and A_3 and determine the prediction

$$\widehat{y}_0 = \arg\max\{\Pr(\mathbf{x}_0|y_0 = A_1), \Pr(\mathbf{x}_0|y_0 = A_2), \Pr(\mathbf{x}_0|y_0 = A_3)\}.$$

b. Suppose that the prior probabilities are $\pi_1 = .8$, $\pi_2 = \pi_3 = .1$. Compute the posterior probabilities $\Pr(y_0 = A_k|\mathbf{x}_0)$ for A_1, A_2 and A_3. Determine the prediction $\widehat{y}_0$.

c. Return to the multinomial naïve Bayes prediction problem and the tutorial of Sect. 10.5. The algorithm set the prior probability estimates to be the relative frequency of undisputed papers attributed to each author. Perhaps this is not the best scheme—there are limitations to the number of papers that any author may write in a year. Recompute the predictions of authorship for all papers using non-informative priors (Sect. 10.3.1). Is there a difference in the results?

10.7.2 Computational

10.3. Walmart sponsored a Kaggle competition to predict shopping trip type (https://www.kaggle.com/c/walmart-recruiting-trip-type-classification). This problem challenges the participants to classify a set of items referred to as a market basket purchased by a customer. The classification system consists of a set of 38 classes or types. The class labels are referred to as *shopping trip type*, or *trip type*. Walmart provides no information on the characteristics of the types, presumably to protect commercial interests.

They have provided a training set comprised of 647,053 purchased items from 95,674 visits. This exercise involves using one categorical variable, the department description of each purchased item, to predict trip type. There are 69 departments and a customer's purchases are distributed across the departments. The question to be answered in this analysis is: how much predictive information about trip type can be extracted from the departments?

More information is available from Kaggle. Some guidance toward answering the question is provided below.

a. Get the data file from Kaggle. The first record contains the names of the variables.

b. The first segment of code should map each record of a purchase to a visit. It's best to build a dictionary in which the key is the visit number (second column) and the value is a list consisting of departments from which the items were purchased. For instance, the three records from visit 9 are

```
8,9,"Friday",1070080727,1,"IMPULSE MERCHANDISE",115
8,9,"Friday",3107,1,"PRODUCE",103
8,9,"Friday",4011,1,"PRODUCE",5501
```

The trip type is 8 and the first department from which an item was purchased from is IMPULSE MERCHANDISE.[8] The dictionary entry for visit 9 is

```
[3, ['IMPULSE MERCHANDISE', 'PRODUCE', 'PRODUCE']]
```

The trip type in our dictionary entry is 3, not 8, as it is in data file because we have relabeled trip type using the integers $0, 1, \ldots, 37$.
Iterate over the data file and build the visit dictionary by aggregating all of the purchases from a single trip. It's a good idea to relabel trip types as consecutive integers $0, 1, \ldots, 37$. If you relabel trip types, then you will need to build a dictionary in which the keys are the Walmart trip type codes and the values are the integer-valued labels. You can build both dictionaries as the script iterates over the data file.

c. Construct a dictionary, we'll call it `dataDict`, in which the keys are visits and values are lists of length $p = 69$ and contain the number of items purchased from each department. The lists correspond to the predictor vectors $\mathbf{x}_1, \ldots, \mathbf{x}_n$, where $n = 95{,}674$. To build `dataDict`, note that the keys are the same as for the visit dictionary. Iterate over the keys and for each entry in the visit dictionary, count the number of items from each department and store the count in a list. Save the list as the value associated with the key. You can count the number of times a particular department `d` occurred in a list `x` using the `count` function:

```
counts = [0]*p  # p is the number of departments.
for i,d in enumerate(depts):
    counts[i] += x.count(d)
```

The variable `depts` is the list of departments from which items were purchased in the course of the visit. Store the count list *and* the trip type as the value associated with visit.

[8] The UPS code 1070080727 is a barcode symbol that specifically identifies the item.

d. The frequency distributions of each trip type (similar to the `count` list) may be constructed using the same code structure. The difference is that the dictionary keys are the trips.

e. Build a dictionary that contains the log-relative frequencies of occurrence of each department for each trip (the $\log(\widehat{\pi}_{k,i})$'s where k indexes trip and i indexes department). Also compute the log-priors (the $\log(\widehat{\pi}_k)$'s).

f. Compute an estimate of the accuracy of the multinomial naïve Bayes prediction function. Iterate over `dataDict` and extract each test vector. Compute the sum of log-probabilities as shown in formula (10.7) and determine the most-likely trip:

```
yhat = np.argmax(postProbList)
```

where the list `postProbList` contains the sums for each trip.

g. While iterating over `dataDict`, tabulate the prediction outcomes in a confusion matrix. The confusion matrix may be a `Numpy` array initialized according to

```
confusionMatrix = np.zeros(shape = (nTrips,nTrips))
```

If the visit in question has been labeled as trip `y0`, then increment the entry in the confusion matrix in row `y0` and column `yhat`. (Relabeling the trip classes as integers from 0 to $g-1$ has made it easy to build the confusion matrix in this manner.) While iterating, compute and print the overall estimate of accuracy:

```
acc = sum(np.diag(confusionMatrix))/sum(sum(confusionMatrix))
```

h. Report the estimated overall accuracy and the confusion matrix.

10.4. The `member_transactions.txt` data file is large enough that we can simply split it into a training and test set for rule evaluation. Randomly split the data into a training set containing 80% of the observations and a test set containing the remaining observations. Produce a confusion matrix comparing the actual segment to the predicted segment. Compute the estimated accuracy rate $\widehat{\alpha}$.

10.5. The `member_transactions.txt` data file contains dates and hours of the day associated with each transaction. Build a contingency table that tabulates the numbers of transactions by day of the week and hour of the day for the primary segment, and then again for all other segments combined. The `Python` module `datetime` has a function that will translate a date to day of the week. Use the `weekday` function from the module to identify the day of the week. Import the module using the instruction `from datetime import datetime`. The `datetime.weekday()` function labels a Monday as 0, a Tuesday as 1, and so on. Does it appear that the two segments differ with respect when they shop? What are the implications for using day of the week and hour of the day for prediction?

Chapter 11
Forecasting

Abstract This chapter provides an introduction to time series and foundational algorithms related to and for forecasting. We adopt a pragmatic, first-order approach aimed at capturing the dominant attributes of the time series useful for prediction. Two forecasting methods are developed: Holt-Winters exponential forecasting and linear regression with time-varying coefficients. The first two tutorials, using complaints received by the U.S. Consumer Financial Protection Bureau, instruct the reader on processing data with time attributes and computing autocorrelation coefficients. The following tutorials guide the reader through forecasting using economic and stock price series.

11.1 Introduction

Health insurers annually set premiums for their customers so that claims are covered and premiums are competitive. Accurately forecasting future costs is essential for competitiveness and profitability. Information for forecasting health insurance costs originates from a variety of sources, but most notably, number and cost of claims, number of insured clients, and institutional costs. All of these variables are subject to trend over time and ought to be forecasted to anticipate next-year costs. Forecasting is also prominent in other arenas, for example, public health, business, and of course, climatology. Forecasting, then, is an essential skill of the data scientist. The data analytic algorithms for forecasting are not terribly dissimilar from those of the more general class of algorithms for predictive analytics. There are, however, aspects of forecasting related to time that ought to be understood and exploited if one is to engage in forecasting.

The term *forecasting* refers to predicting an outcome that will be realized, or observed, in the future. *Prediction* is a more general term for predicting an unobserved value for which there may or may not be a time component.

B. Steele et al., *Algorithms for Data Science*,
DOI 10.1007/978-3-319-45797-0_11

Broadly speaking, forecasting functions are constructed by estimating the mean level of a process at a future time. Since the estimated mean level becomes the forecast, the central aim is to estimate the expected value, or the mean, of the random variable of interest at a future time step. In this discussion, the current time step is n so that the estimated mean level $\widehat{\mu}_{n+\tau}$ is a forecast τ time steps in the future.

The data of interest are traditionally referred to as time series because the process generating the data possesses a chronological attribute. As a result, the data are generated and observed in chronological order, and the chronological ordering is presumed to be useful for understanding the process and for forecasting. *Streaming data* are time series data observed in real-time, and the aim often is to compute forecasts as new observations arrive—also in real-time.[1] Effective algorithms for extracting information from time series data ought to exploit the chronological ordering, but only if chronologically near observations are not independent but instead are *autocorrelated*. If autocorrelation is present, then data that are most recently observed are most useful for forecasting and the forecasting function ought to place greater value on more recent observations at the expense of observations observed in the more distant past.

It's assumed in this discussion that the process generating the data stream is at drift. A process is at drift if the mean level and possibly the variance are not constant but instead are changing, and the change is predominantly toward larger, or smaller, values. The direction of trend and rate of change may vary with time as well. Figure 11.1 provides an example. In this figure, the number of consumer complaints related to mortgages and filed with the U.S. Consumer Financial Protection Bureau are plotted against the date that the complaint was received. Many more complaints were made on weekdays compared to weekends. Weekend counts show little trend over time, but weekday counts show several spikes and some downward trend preceding the end of a year. Since trend in weekday counts varies with time, trend at time step n must be estimated by data not too distant from n to be reflective of the current process.

The visual evidence of trend in the weekday counts imply that past observations are predictive of future observations and that the data are autocorrelated. The term *serial correlation* is used interchangeably with autocorrelation but it is more to the point since autocorrelation may originate from other sources such as spatial proximity. In any case, a claim that these data are independent is indefensible and it would be foolish to analyze the data for forecasting purposes as if they were independent.

If trend is present in a data series, then the mean of all observations is not terribly informative about what is happening to the process generating the series. If the aim is to forecast future values as data arrive in a stream, then a sensible approach is to update a previously-made forecast by incorporating

[1] Chapter 12 discusses computational aspects of processing streaming data.

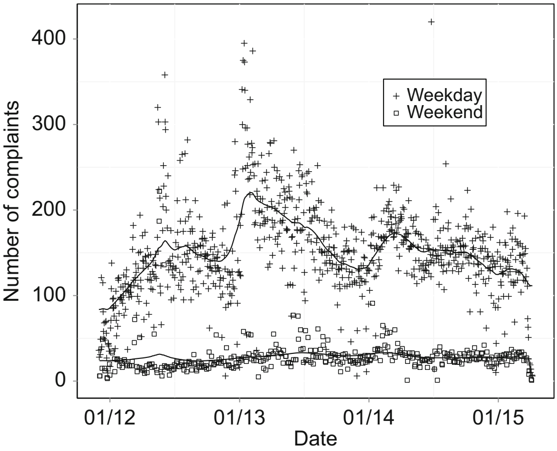


Fig. 11.1 Number of consumer complaints about mortgages plotted against date. Complaints were filed with the U.S. Consumer Financial Protection Bureau

the most recent observation. When a new observation is incorporated, then the importance of each of the preceding observations is reduced to some extent. The extent to which the recent observations dominate the forecast at the expense of past observations ought to be tied to the rate of change in trend. If the process changes slowly, then old observations are more useful than if the process is changing rapidly.

11.2 Tutorial: Working with Time

Before proceeding further with the statistical aspects of forecasting, we address a computational issue. Many data sets created by instrumentation or automatic data logging contain time stamps that must be translated from character string such as ‘Feb042015’ or ‘2015:02:04:23:34:12’ to a numerical representation. For analytical purposes, it’s efficient to construct a variable that is ordered chronologically such as the number of days or hours elapsed since 12 p.m., December 31, 2000. The `Python` modules `time`, `datetime`, and `calendar`, among others, are useful for this task. However, it’s sometimes better to have greater control over the translation if the data are non-standard or contain errors. We take the do-it-yourself approach in this tutorial.

The reader is introduced to working with time variables by investigating trend and autocorrelation in the numbers of complaints submitted to the U.S. Consumer Financial Protection Bureau.[2] The data consist of complaint records received by the Consumer Financial Protection Bureau from individuals. A rich set of attributes are recorded in these data. In particular, complaints are classified as related to a specific product or service such as mortgages, credit reporting, and credit cards. Figure 11.1 was constructed

[2] We used these data in Sect. 6.5.

using data from this source. While the data from the Consumer Financial Protection Bureau is an excellent publicly available resource for studying the quality of service provided by companies to consumers, it's not perfectly suited for our objective of investigating trend and estimating autocorrelation. Therefore, we'll have to restructure the data.

Since we are interested in analyzing counts over time, we'll map individual complaint records to the day that the complaint was received. The mapping produces a dictionary in which the keys are day (represented as a character string) and the values are lists of three items: a chronologically ordered integer representation of day (e.g., $1, 2, \ldots$), a label identifying the day as a weekday or a weekend day, and the number of complaints that were submitted on the day in question.

The dictionary will be written to a second file and used as the data source for the tutorial on computing autocorrelation coefficients. When writing the file, days will be chronologically ordered for plotting and computing autocorrelation coefficients. The task at hand, then, is to count the number of complaints for each day, label each day as a weekend or weekday day, and order the days chronologically.

1. Go to https://catalog.data.gov/dataset/consumer-complaint-database and retrieve the data file `Consumer_Complaints.csv`.
2. Create an empty dictionary `dataDict` to contain the counts of complaints for each day in the file. Dictionary keys will be a string in the format *month/day/year* and dictionary values will be three-element lists containing the number of elapsed days between the date that the complaint was received and December 31, 2009, a label identifying whether the day is a weekday or weekend day, and the number of complaints received on the day.
3. Before processing the data set, read the first line of the data file using the `readline()` function and inspect the column positions and names of the variables:

```
path = '../Consumer_Complaints.csv'
with open(path, encoding = "utf-8") as f:
    variables = f.readline()
    for i,v in enumerate(variables.split(',')):
        print(i,v)
```

   The variable named `Product` identifies the product or service and the variable `Date received` contains the date that the complaint was received by the Consumer Financial Protection Bureau. Be aware that the column identifiers for these variables may change if the Bureau restructures the data file.

4. Process each line of the data file but save only those complaints of a particular type, say credit reporting. The format for dates are the form

month/day/year where month and day are both represented by two characters and year is represented by four characters. Extract day, month, and year from lines with entries for the chosen product or service.

```
for record in f:
    line = record.split(’,’)
    if line[1] == ’Credit reporting’:
        received = line[0]
        month = int(received[0:2])
        day =  int(received[3:5])
        year = int(received[6:10])
```

The product entry may be missing from some of the records so it’s best to use an exception handler. The exception handler wraps around the code as follows:

```
try:
    if line[1] == ’Credit reporting’:
        ...
        n += 1
except:
    pass
```

The variable `n` will count the number of valid records. Initialize `n` before the `for` loop.

5. Count the number of days elapsed between the time that a complaint was received and December 31, 2009. First, define a list `daysPerMonth` with the numbers of days in each of the months January through November.

```
daysPerMonth = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30]
```

In leap years, there is an additional day in February. We will handle the leap year in the code with a conditional statement. Define `daysPerMonth` before the `for` loop.

6. Compute the number of days elapsed between December 31, 2009 and the day that the complaint was received as follows. Sum over past years and past months and the number of days elapsed since the beginning of the current month. We’ll need the number of days since the beginning of the year through the months preceding the current month. The function call `sum(daysPerMonth[:month-1])` computes the sum according to

$$\texttt{sum(daysPerMonth[:month-1])} = \begin{cases} 0, & \texttt{if month} = 1, \\ 31, & \texttt{if month} = 2, \\ \vdots & \vdots \\ 344, & \texttt{if month} = 12. \end{cases} \qquad (11.1)$$

Compute the total number of elapsed days:

```
elapsedDay = (year - 2010)*365 +  sum(daysPerMonth[:month-1]) + day \
              + int(year == 2014 and month >= 3) + int(year > 2014)
```

We're taking into account leap years by adding an additional day to the count of elapsed days if the complaint was received after the last day of February, 2014. The terms `int(year == 2014 and month >= 3) + int(year > 2014)` will add an additional day to the count if necessary. The most recent leap day occurred in 2014 and the next leap day will occur in 2018. Adapt the statement if the current date is later than March 1, 2018.

Indent the statement so that it executes only if the product matches the selected type of product.

7. Use the `weekday` function from the `datetime` module to identify the day of the week. Import the module using the instruction `from datetime import datetime` at the beginning of the script.

   The `datetime` function `weekday()` labels a Monday as 0, a Tuesday as 1, and so on. Compute the day of the week and store it as a variable named `dOfWeek`.

```
dOfWeek = datetime.strptime(received, '%m/%d/%Y').weekday()
```

   The function call `datetime.strptime(received, '%m/%d/%Y')` translates `received` to a `datetime` object from which the day of the week can be computed.

8. Translate the integer-valued day of the week to a label identifying weekdays and weekend days:

$$\texttt{label} = \begin{cases} \text{'Weekend'}, & \text{if } \texttt{dOfWeek} \in \{5, 6\}, \\ \text{'Weekday'}, & \text{if } \texttt{dOfWeek} \notin \{5, 6\}. \end{cases}$$

9. Create an entry if the received date is not in the dictionary `dataDict`. Otherwise increment the number of complaints received on the date:

```
if dataDict.get(received) is None:
    dataDict[received] = [elapsedDay,label,1]
else :
    dataDict[received][2] += 1
```

10. Import the module `operator` at the beginning of the script. When all of the records have been processed, sort the dictionary according to elapsed day. Store the sorted dictionary as a list.

```
lst = sorted(dataDict.items(), key=operator.itemgetter(1))
```

    The argument `dataDict.items()` instructs the `sorted` function to return the dictionary keys and values as a two-tuple, each containing the key and the value, for example, `(’12/31/2014’, [1826, ’Weekday’, 78])`. The `key` argument specifies which variable to use for sorting, sometimes called the sorting key (not to be confused with the dictionary key). It is to be the zeroth argument in the *value* since we must sort on days elapsed since December 31, 2009. If the dictionary were to be sorted by the number of complaints, then the instruction would be

```
sorted(dataDict.items(),key=lambda item: item[1][2])
```

11. We’ll use the data in the next tutorial so save the list by creating a binary file containing `lst` that can be loaded back into memory later.

```
import pickle
path = ’../Data/lst.pkl’
with open(path, ’wb’) as f:
   pickle.dump(lst, f)
```

    We used the pickle module in instruction 12 of the Chap. 8, Sect. 8.4 tutorial.

12. Check that the pickle operation has successful saved the data.

```
with open(path, ’rb’) as f:
    lst = pickle.load(f)
print(lst)
```

13. Plot the count of complaints against elapsed day:

```
import matplotlib.pyplot as plt
day = [value[0] for _,value in lst]
y = [value[2] for _,value in lst]
plt.plot(day, y)
```

14. Plot a sub-series, say days 500 through 1000.

```
indices = np.arange(500,1001)
plt.plot([day[i] for i in indices],[y[i] for i in indices])
```

We turn now to analytical methods.

11.3 Analytical Methods

11.3.1 Notation

For the time being, we will consider univariate data and denote the observed data as an ordered arrangement of n values

$$D_n = (y_1, y_2, \dots, y_t, \dots, y_n),$$

where y_t is a realization of the random variable Y_t. The indexing system orders the observations chronologically so that y_t is observed after y_{t-1} for every $0 < t \leq n$. It won't be assumed that the inter-arrival times of observations are constant, and so the index t orders the data chronologically but is not equivalent to clock time. Instead, t denotes a time step.

If forecasting is carried out at the same time that the data arrive, then the data are referred to as streaming data, and the analytical methods are often called real-time analytics.

11.3.2 Estimation of the Mean and Variance

If streaming data are generated by a process that is believed to be stationary, and so, not at drift, then all observations ought to be used for estimating μ, σ^2, and any other relevant parameters that describe the process. The only relevant issue posed by streaming data is data storage. Storing a large number of observations is neither practical nor necessary since an associative statistic can be updated as the data arrive. For example, an associative statistic for estimating the mean and variance was discussed in Sect. 3.4. We'll add the subscript n to the elements of the associative statistic and write them as

$$\begin{aligned}\mathbf{s}(D_n) &= (s_{1,n}, s_{2,n}, s_{3,n}) \\ &= \left(\sum_{t=1}^{n} y_t, \sum_{t=1}^{n} y_t^2, n\right).\end{aligned} \tag{11.2}$$

The estimators of μ and σ^2 at time step n are functions of $\mathbf{s}(D_n)$ given by

$$\begin{aligned}\widehat{\mu}_n &= s_{1,n}/s_{3,n}, \\ \widehat{\sigma}_n^2 &= s_{2,n}/s_{3,n} - (s_{1,n}/s_{3,n})^2.\end{aligned} \tag{11.3}$$

An algorithm for computing $\widehat{\mu}_n$ and $\widehat{\sigma}_n^2$ at time step n updates $\mathbf{s}(D_{n-1})$ by computing

$$\begin{aligned}s_{1,n} &\leftarrow s_{1,n-1} + y_n \\ s_{2,n} &\leftarrow s_{2,n-1} + y_n^2 \\ s_{3,n} &\leftarrow s_{3,n-1} + 1.\end{aligned}$$

Upon completion, $\widehat{\mu}_n$ and σ_n^2 are computed using formulas (11.3).

Another approach is necessary if it is believed that the mean level is not constant but rather is at drift. Since older observations contain less information about the current level, older observations should be omitted or down-weighted when estimating the mean level. A *moving window* is an elementary approach to estimating the mean level in the presence of drift. A moving window is a set consisting of the most recently observed m observations. The moving window changes with each new observation as if you were looking in a fixed direction from the window of a moving vehicle. As each new observation arrives, it replaces the oldest observation in the set. Hence, a constant number of observations are contained in the moving window. An estimate of μ_n computed from the moving window is called a simple moving average.[3] The moving average estimator of μ_n is

$$\widehat{\mu}_n^{\mathrm{MA}} = m^{-1} \sum_{t=n-m+1}^{n} y_t.$$

There are limitations to moving windows. If m is too small, the mean will be sensitive to random deviations and as a forecasting tool, it will be imprecise. If m is too large, then a change in the mean may not be detectable until undesirably many time steps have passed. Furthermore, a simple average of m observations treats each observation equally rather than assigning the largest weights to the most recent observations. Our premise is that the information value of y_t decreases regularly as n increases and so the two-weight system (either $1/m$ or zero) is crude.

[3] Exercise 9.5 of Chap. 9 asked the reader to program a moving average.

11.3.3 Exponential Forecasting

It's possible to use the complete set of observations $y_1, \ldots, y_n$ and assign every observation a weight that depends on the time step at which it was observed. We can use the exponential weights of the k-nearest-neighbor prediction function of Chap. 9 for this purpose. The exponentially weighted mean estimator of $\mathrm{E}(Y_n|y_1, \ldots, y_n) = \mu_n$ is

$$\widehat{\mu}_n = \sum_{t=1}^{n} w_t y_t \tag{11.4}$$

where $0 \leq w_t \leq 1$ and $1 = \sum_{t=1}^{n} w_t$ [27]. The weights are defined according to

$$w_t = \alpha(1-\alpha)^{n-t}, t \in \{1, 2, \ldots, n\}.$$

Since $0 < \alpha < 1$, the weights increase to α as t increases from 1 to n. If it is the case that $\sum_{t=1}^{n} w_t < 1$, then we use a set of scaled weights. The scaled weights are $v_t = w_t / \sum_{t=1}^{n} w_t, t = 1, \ldots, n$.

The effect of different choices of α on the weights are shown in Fig. 11.2. Values of α greater than .25 lead to estimators that reflect the behavior of the most recent observations since the weights diminish rapidly as t recedes from n to zero. In contrast, values of α less than .05 lead to estimators that reflect the values of a larger and older set of observations.

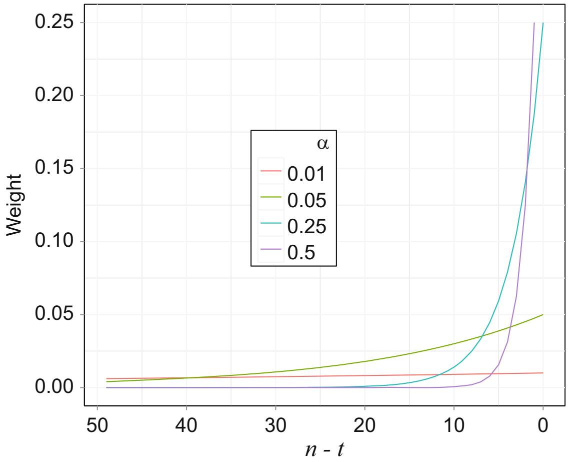


Fig. 11.2 Exponential weights w_{n-t} plotted against $n - t$, where n is the current time step and t is a preceding time step

If w_t is replaced with $\alpha(1-\alpha)^{n-t}$ in formula (11.4), then it is readily deduced that $\widehat{\mu}_n$ is a linear combination of the previous step estimate $\widehat{\mu}_{n-1}$ and the most recent observation y_n. Specifically,

$$\widehat{\mu}_n = (1-\alpha)\widehat{\mu}_{n-1} + \alpha y_n. \tag{11.5}$$

Exercise 11.1 asks the reader to verify Eq. (11.5). Equation (11.5) provides some guidance towards choosing α since it's apparent that the most recent observation receives the weight α whereas the older observations collectively receive the weight $1 - \alpha$. Further insight toward choosing α can be obtained from an investigation of autocorrelation, the subject of the next section. Equation (11.5) also shows that $\widehat{\mu}_n$ can be computed simply and quickly by updating the previous step estimate $\widehat{\mu}_{n-1}$ and hence, the past observations need not be stored.

Exponentially weighted mean estimators are sometimes used for visually representing the drift in a series. If there's no interest in forecasting, then values observed both before and after a time step t are used in the computation of the exponential mean $\widehat{\mu}_t$. The term *exponential smoothing* is used to describe the process.

Updating formulas such as Eq. (11.5) are useful for streaming data because the computational effort of computing $\widehat{\mu}_n$ is trivial. The exponentially weighted mean $\widehat{\mu}_n$ can be used to forecast a future value $Y_{n+\tau}$, where τ counts the number of steps forward from time step n. We'll refer to forecasting using Eq. (11.5) as *exponential forecasting.* The subject is discussed in detail in Sect. 11.6.

11.3.4 Autocorrelation

Effective algorithms for extracting information from time series data exploit the tendency of observations that are near in time to be similar in value, or serially correlated. The statistical term for these observations is *dependent*, and the exponentially weighted mean is an example of exploiting serial correlation. However, there aren't any gains to using exponential forecasting if observations are not dependent. Instead, there's a loss of precision because early observations are lost (or nearly so) from the estimator. It's desirable, then, to be able to assess the degree of intra-series correlation. The terms *autocorrelation* and serial correlation are used to describe intra-series correlation in an ordered arrangement of observations $(y_1, \ldots, y_n)$. In Sect. 6.8.3, we used sample autocorrelation coefficients to assess serial correlation in an arrangement of linear regression residuals. This discussion develops the ideas of serial correlation and autocorrelation coefficients in detail.

Autocorrelation is quantified by a set of process or population parameters $\rho_0, \rho_1, \ldots, \rho_r$ where r is a positive integer. The parameter ρ_τ is commonly referred to as the lag-τ autocorrelation coefficient since it measures the degree of linear association between one set of observations that lag behind another set by τ time steps. A definition is

$$\rho_\tau = \frac{\mathrm{E}[(Y_t - \mu)(Y_{t-\tau} - \mu)]}{\sigma^2}, \tag{11.6}$$

where $\sigma^2 = \mathrm{E}[(Y_t - \mu)^2]$ is the variance of Y_t [27]. Formula (11.6) assumes that the process is stationary since there is single population mean μ and variance σ^2 rather than a series of means, say $\mu_1, \mu_2, \ldots$, and a series of variances. Note that Eq. (11.6) implies $\rho_0 = 1$.

An estimate of the lag-τ autocorrelation coefficient is

$$\widehat{\rho}_\tau = \frac{\sum_{t=1}^{n-\tau}(y_t - \widehat{\mu})(y_{t+\tau} - \widehat{\mu})}{(n-\tau)\widehat{\sigma}^2}. \tag{11.7}$$

Formula (11.7) provides appropriate estimates of ρ_τ if the mean level and variance of the process are constant and hence, not at drift. The estimator $\widehat{\rho}_\tau$ is the Pearson correlation coefficient computed from the $n - \tau$ data pairs

$$\{(y_1, y_{\tau+1}), (y_2, y_{\tau+2}), \ldots, (y_{n-\tau}, y_n)\}. \tag{11.8}$$

A set of estimates $\{\widehat{\rho}_1, \widehat{\rho}_2, \ldots, \widehat{\rho}_\tau\}$ that are mostly large in magnitude suggest the process is predictable in the sense that the most recently observed values will be similar in magnitude to future values provided that the future values are not too far ahead of the most recent observation. A process subject to drift will manifest autocorrelation because when y_t is substantially greater (or smaller) than μ, then observations near in time step are also likely to be greater (or smaller) than the mean and more similar than a random selection of observations. Typically, the autocorrelation coefficients will be positive and will diminish slowly towards zero as τ tends to larger values.

Figure 11.3 shows two sets of estimated autocorrelation coefficients computed from the numbers of mortgage-related complaints received by the U.S. Consumer Financial Protection Bureau. Notice that the autocorrelation coefficients computed from both weekday and weekend counts reveal substantive and persistent autocorrelation. The successive weekend days are either 1 day apart, e.g., Saturday then Sunday, or 6 days apart when Sunday is followed by the Saturday of the next weekend.[4] Note that larger values of $\widehat{\rho}$ are associated with lags 5, 10, 15, and 20 in the weekday series. The implication is that the counts observed on a particular day of the week (e.g., Mondays) are more similar compared to counts received on different days of the week. Therefore, we conclude that the numbers of received complaints depends on the day of the week.

11.4 Tutorial: Computing $\widehat{\rho}_\tau$

The tutorial of Sect. 11.2 guided the reader through the reduction of the consumer complaints data file and the creation of a `Python` list in which each item consisted of the data recorded on a particular day. The list was ordered

[4] This is an example of a time series with time steps of distinctly different lengths.

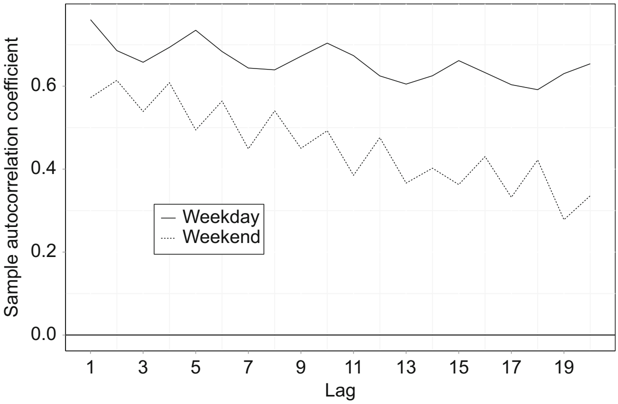


Fig. 11.3 Estimates of the autocorrelation coefficient for lags $1, 2, \ldots, 20$. The target variable, number of consumer complaints regarding mortgages received by the U.S. Consumer Financial Protection Bureau, have been grouped by weekday and weekend

chronologically. The list items contain the date, the number of days elapsed between the date and December 31, 2009, an identifier of the date as weekday or weekend, and the number of complaints received on the date. The task of this tutorial is to estimate autocorrelation coefficients $\widehat{\rho}_1, \widehat{\rho}_2, \ldots, \widehat{\rho}_{20}$ for the weekday observations and again for the weekend data. The coefficients will then be plotted against lag to produce a figure analogous to Fig. 11.3.

Computing the lag-τ autocorrelation coefficient requires the variance estimate $\widehat{\sigma}^2$ which in turn requires the sample mean $\widehat{\mu}$. These terms are computed from Eq. (11.2). Section 11.3.2 provides guidance. In addition to $\widehat{\sigma}^2$, the calculation of $\widehat{\rho}_\tau$ requires the sum

$$C_n = \sum_{t=\tau+1}^{n} (y_t - \widehat{\mu})(y_{t-\tau} - \widehat{\mu}) = \sum_{t=\tau+1}^{n} c_t. \tag{11.9}$$

The term

$$c_t = (y_t - \widehat{\mu})(y_{t-\tau} - \widehat{\mu})$$

is referred to as the cross-product.

Suppose that C_n is computed for successive time steps, from $t = \tau + 1$ to $t = n$.[5] At time step t, C_{t-1} is available, having been computed for the last step. We need to compute $C_t = C_{t-1} + c_t$. The computation of c_t at time step t requires the past observation $y_{t-\tau}$. Therefore, it's necessary to store some of the data values. For this data set, there's no problem storing all of the data, but if the computations are being carried out as the data arrive in a stream at great velocity, we may not be able to store all observations. The solution is to maintain a set (a `Python` list to be precise) containing the necessary past observations. At time t, past observations are stored in the set $P_t = \{y_{t-\tau}, y_{t-\tau+1}, \ldots, y_{t-1}\}$. When y_t is to be processed, we extract $y_{t-\tau}$

[5] As would be done if the data were arriving in a stream.

from P_t, and compute c_t and the update $C_t = C_{t-1} + c_t$. Then, y_t replaces $y_{t-\tau}$ in P_t, thereby creating P_{t+1} and t is advanced to $t + 1$.

Table 11.1 shows the contents of the past-values storage list and the location of $y_{t-\tau}$ in the list as t advances. The modulo function provides a simple means of extracting and replacing the oldest values in a list of τ observations. The list is updated at time step t by replacing $y_{t-\tau}$ in position $t \bmod \tau$ with y_t. For example, suppose that $t = 10$ and $\tau = 5$. Then $10 \bmod 5 = 0$ and y_5 is stored in position 0. We extract that value and replace it with y_{10}. On the next step, $11 \bmod 5 = 1$, so y_6 is extracted from position 1. Lastly, y_{11} stored in position 1. We continue advancing through the five positions of the list until $t = 15$. At that point, we're back to the beginning of the cycle and y_{10} is extracted from position 0.

Table 11.1 Contents of the past-values storage list for $\tau = 5$ for time steps $t, t+1$ and $t + 4$ when t is a multiple of τ (and hence, $t \bmod \tau = 0$). The boxed value, $y_{t-\tau+i}$, is used in the calculation of the cross-product. The contents of the list are shown at the start of the iteration and after the list has been updated. Note that the position of the replaced value is $(t + i) \bmod \tau$, for $i \in \{0, 1, 2, 3, 4\}$

		Position				
i	Time step	0	1	2	3	4
0	t start	$\boxed{y_{t-5}}$	y_{t-4}	y_{t-3}	y_{t-2}	y_{t-1}
	t end	y_t	y_{t-4}	y_{t-3}	y_{t-2}	y_{t-1}
1	$t+1$ start	y_t	$\boxed{y_{t-4}}$	y_{t-3}	y_{t-2}	y_{t-1}
	$t+1$ end	y_t	y_{t+1}	y_{t-3}	y_{t-2}	y_{t-1}
	$\vdots$ $\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
4	$t+4$ start	y_t	y_{t+1}	y_{t+2}	y_{t+3}	$\boxed{y_{t-1}}$
	$t+4$ end	y_t	y_{t+1}	y_{t+2}	y_{t+3}	y_{t+4}

The following code demonstrates. The list storing the past data is named `pastData`. We use t as the replacement value instead of y_t:

```
tau = 5
pastData = [0]*tau
for t in range(16) :
    print(’step =’, t, ’ start. position =’, t%tau, ’value =’,
             pastData[t%tau])
    pastData[t%tau] = t
    print(’step =’, t, ’ end.   position =’, t%tau, ’value =’,
             pastData[t%tau])
```

This code block may be executed in a console.

The tutorial instructs the reader to compute $\widehat{\rho}_\tau$ using two iterations. The first iteration computes the sample mean $\widehat{\mu}$ and variance $\widehat{\sigma}^2$. The second iteration computes the sum of the cross-products.

1. Read the pickle file and store the contents.

```
import pickle
path = '../Data/lst.pkl'
with open(path, 'rb') as f:
    lst = pickle.load(f)
print(lst[:20])
```

2. Initialize lists `sDay` and `eDay` to store the associative statistics $\mathtt{s}(D_n)$ for weekdays and weekends (Eq. (11.2)). For instance, `sDay = [0]*3` initializes a three-element list containing zeros.
3. Iterate over `lst` and extract the data.

```
for item in lst:
    date, triple = item
    elapsed, dayOfWeek, y = triple
```

4. As the script iterates over `lst`, update `sDay` and `eDay`. We'll build a function `updateStat` to update the statistics. Put the function definition before the `for` loop. The function is

```
def updateStat(y,assocStat):
    assocStat[0] += y
    assocStat[1] += y**2
    assocStat[2] += 1
    return assocStat
```

5. Use a conditional statement so that `sDay` is updated if the day of the week is a weekday (see instruction 8 of Sect. 11.2). Otherwise, update `eDay`. The function call `sDay = updateStat(y, sDay)` updates `sDay`. Update the statistics after extracting the elements of `triple`.
6. Process the entire data set. Then, compute $\widehat{\mu} = n^{-1}\sum y_i$ and $\widehat{\sigma}^2 = n^{-1}\sum y_i^2 - \widehat{\mu}^2$ (Eq. (11.2)) for the weekday counts. Name them `meanEst` and `varEst` in the `Python` code.
7. The next step is to compute the sum of the cross-product terms $C_n = \sum_{t=\tau+1}^{n} c_t$ for the weekday counts. Initialize a τ-length list `pastData` as described above. Build a `for` loop that iterates over `lst` to compute the sum. Extract elapsed days and the count on each iteration.

```
crossSum = 0
weekdayCounter = 0
for item in lst:
    date, triple = item
    elapsed, dayOfWeek, y = triple
```

8. Update the storage list `pastData`. Update C_n if $t \geq \tau$.

```
if dayOfWeek == 'Weekday':
    if weekdayCounter >= tau:
        crossSum += (y-meanEst)*(pastData[weekdayCounter%tau]-meanEst)
    pastData[weekdayCounter%tau] = y
    weekdayCounter += 1
```

   Note that the index `weekdayCounter%tau` points to a value, namely, $y_{t-\tau}$, that was stored earlier.
9. At the completion of the `for` loop, compute $\widehat{\rho}_\tau$:

```
rhoTau = crossSum/((weekdayCounter - tau)*varEst)
```

10. The next task is to compute estimates of ρ_τ over a range of values for τ, say $\tau \in \{1, 2, \ldots, 20\}$, once using the weekday data and again using the weekend data. This can be accomplished by wrapping the existing `for` loop that processes `lst` in an outer `for` loop that iterates over $\{1, 2, \ldots, 20\}$. At the completion of each iteration of the outer loop, save $\widehat{\rho}_\tau$ in a list. Be sure to recompute the sample mean and variance (instruction 6) whenever changing the data source from weekend data to weekday data, and vice versa.
11. We'll suppose that the weekend data was used to fill a list of estimated autocorrelation coefficients named `WeekendRho` and that the weekday data filled `WeekdayRho`. With these lists, plot the estimated autocorrelation coefficients against lag using the code segment below.

```
import matplotlib.pyplot as plt
lag = np.arange(1,21)
plt.plot(lag, WeekdayRho)
plt.plot(lag, WeekendRho)
```

11.4.1 Remarks

It should be kept in mind that the time steps vary with respect to the elapsed time between steps. For the weekday series, the interval between the beginning of one step and the next step is approximately 1 day except when 1 day is Friday and the following weekday is Monday. For the weekend data series, the variation in time step length is greater: the interval between the beginning of one step and the next is 1 day or 5 days. Logically, the similarity between counts ought to be greater for shorter time step intervals and so a comparison of autocorrelation estimates between the weekday and weekend series is somewhat muddied by the differences in time steps. We could have maintained uniform intervals of 24 h between time steps if the data series had not been split into weekday and weekend series. But, the difference between weekday and weekend sample means provides very convincing evidence that two distinct processes are generating counts: weekday and weekend processes, and failing to separate the data implies that analysis and forecasting will either be more complicated or less accurate than analysis and forecasting using the two series.

Despite the variation in time step length, it's clear that there is a significant degree of autocorrelation in the underlying complaint generation processes. Time step length variation does not explain why the weekend series of sample autocorrelation coefficients has a sawtooth appearance (Fig. 11.3). The sawtooth phenomenon suggests that the number of complaints made on different Saturdays are more alike than complaints made on different days of the week. The same statement holds for Sundays. The weekday series similarly suggests that Mondays are similar to other Mondays, Tuesdays similar to other Tuesdays, and so on. From these observations, it's deduced that day of the week differences are persistent and substantive. This suggests that a forecasting model ought to account for differences between day of the week and that a linear model containing day of the week as a factor is a good starting point for developing a forecasting algorithm. Figure 11.1, the mortgage complaint series, suggests that the mean level drifts over time in a somewhat predictable fashion. A forecasting model ought to account for drift as well.

11.5 Drift and Forecasting

The level of the consumer complaints series shown in Fig. 11.1 changes over time. This phenomenon has been referred to as *drift*, an admittedly imprecise term synonymous with trend. Our intent in using the term drift rather than trend is to articulate the tendency for the rate of change and direction of the trend to morph over time. Often, the observed changes in trend are in response to factors that are either unidentifiable or unquantifiable. We think of trend as is more regular and predictable than drift. Yet drift is distinctly different from random movements since there is a clear direction of movement. Though drift is often difficult to explain using other variables or series, referred to as exogenous variables in economics, it is possible to at least partially account for drift using past observations. The degree of success in this regard depends on the consistency of the drift.

The objective of forecasting is to predict an observation $Y_{n+\tau}$ that will be realized τ time steps from the current time step n. We use the observed data $D_n = (y_1, , \ldots, y_n)$ to forecast $Y_{n+\tau}$. If it were assumed that the process generating the data was not at drift, then a best predictor of $Y_{n+\tau}$ is the mean level $\mathrm{E}(Y_t) = \mu$. Without knowledge of μ, the sample mean is used as the forecast of $Y_{n+\tau}$. Furthermore, because μ is not expected to change, we use the same estimate as a forecast for all values of τ. The prediction of $y_{n+\tau}$ is

$$\widehat{y}_{n+\tau} = \overline{y}_n.$$

We are rarely comfortable with the assumption that the process is not subject to drift, and so turn now to methods that accommodate drift.

Our interest centers on *nonstationary* processes, that is, processes that are at drift. We have already been introduced to exponentially weighted forecasting, a technique that computes a weighted mean placing greater weight on recent observations and less weight on older observations. Not only does this estimator capture drift by responding to recent observations, but it is also computationally efficient and it easily accommodates different rates of change by manipulating of the tuning constant α. Computations are carried out using the updating formula given in Eq. (11.5).

While the series of estimates $\widehat{\mu}_1, \ldots, \widehat{\mu}_n$ obtained from the exponentially weighted mean will reflect trend provided that α is appropriately chosen, the estimator of $\mu_{n+\tau}$ and the forecast of $y_{n+\tau}$ is unfortunately, $\widehat{\mu}_n$ for all values of $0 \leq \tau$. For small values of τ and a slowly drifting mean, the exponentially weighted mean may yield accurate estimates. However, even with obvious trend in the recent estimates of μ_n, the forecasts do not account for trend. To accommodate trend, we turn to Holt-Winters forecasting.

11.6 Holt-Winters Exponential Forecasting

The Holt-Winters method is an extension of exponential weighting that includes an additional term for trend. The Holt-Winters exponential forecast $\widehat{y}_{n+\tau}$ extends exponential forecasting (Sect. 11.3.3) by introducing a rate of change parameter β_n. The estimated mean level is updated according to

$$\widehat{\mu}_n = (1 - \alpha_h)(\widehat{\mu}_{n-1} + \widehat{\beta}_n) + \alpha_h y_n, \tag{11.10}$$

where $0 < \alpha_h < 0$ is a smoothing constant analogous to α in exponential forecasting. The difference between exponential forecasting (Eq. (11.5)) and

Holt-Winters forecasting is the inclusion of the coefficient $\widehat{\beta}_n$. The coefficient $\widehat{\beta}_n$ is the estimated rate of change in the mean level between time step $n-1$ and n. The coefficient $\widehat{\beta}_n$ may be described as a time-varying slope estimate. The forecast of $Y_{n+\tau}$ made at time step n for τ steps in the future is

$$\widehat{y}_{n+\tau} = \widehat{\mu}_n + \widehat{\beta}_n \tau. \tag{11.11}$$

Equation (11.11) explicitly accounts for trend in the forecast through the term $\widehat{\beta}_n \tau$. The forecast takes the same form as a linear regression model $\mathrm{E}(Y|x) = \beta_0 + \beta_1 x$. However, the slope coefficient is fixed in the linear regression model whereas Holt-Winters allows the slope to vary over time.

The next matter to address is an algorithm for computing the coefficients $\widehat{\mu}_1, \ldots, \widehat{\mu}_n, \widehat{\beta}_1, \ldots, \widehat{\beta}_n$. Formula (11.10) is used for $\widehat{\mu}_1, \ldots, \widehat{\mu}_n$. Once again, we use exponential weighting to compute $\widehat{\beta}_1, \ldots, \widehat{\beta}_n$. In other words, $\widehat{\beta}_n$ will be a linear combination of the past estimate $\widehat{\beta}_{n-1}$ and the change between $\widehat{\mu}_n$ and $\widehat{\mu}_{n-1}$. Another tuning constant is needed to control the rate of change in the slope coefficients. The tuning constant is $0 < \alpha_b < 1$, and the updating formula for the slope coefficients is

$$\widehat{\beta}_n = (1 - \alpha_b)\widehat{\beta}_{n-1} + \alpha_b(\widehat{\mu}_n - \widehat{\mu}_{n-1}). \tag{11.12}$$

Equation (11.12) incorporates the observed change in estimated mean level, $\widehat{\mu}_n - \widehat{\mu}_{n-1}$, which may be thought of as an elementary estimate of the current rate of change in the mean level. Large values of α_b allow the slope estimates to change rapidly in response to large changes in the estimated mean level whereas small values of α_b suppress the effect of large changes in the estimated mean level.

At time step n, $\widehat{\mu}_n$ is computed using Eq. (11.10) and then $\widehat{\beta}_n$ is computed (Eq. (11.12)). Reasonable choices for the initial values are $\widehat{\mu}_1 = y_1$ and $\widehat{\beta}_1 = 0$. The updating equations begin at time step $n = 2$ since the updates of $\widehat{\mu}_n$ and $\widehat{\beta}_n$ require the previous values $\widehat{\mu}_{n-1}$ and $\widehat{\beta}_{n-1}$.

Figure 11.4 illustrates Holt-Winters and exponential forecasting. A short series of price quotations for Apple stock circa 2013 are shown along exponentially weighted and Holt-Winters forecasts for $\tau = 20$ time steps in the future. The smoothing constant for exponential smoothing was set to be $\alpha_e = .02$ and for Holt-Winters forecasting, we set $\alpha_h = .02$ and $\alpha_b = .015$. The forecasts appear to be ahead of the observed values. It's not the case, however, if we consider a single time step, say 26,600, and look at the forecasts and actual values. The observed value is greater than the forecast because the forecast was made 20 time steps earlier and at that time the level of the series was less than at time step 26,600. By setting α_h and α_b to be larger, the forecasts will change more rapidly with the data series; however, rapid changes in the data series induce large, and often inaccurate changes in the forecast series. In any case, there's relatively little difference between exponentially weighted and Holt-Winters exponential forecasting. We found it difficult to significantly improve on exponential smoothing using Holt-Winters forecasting for this problem.

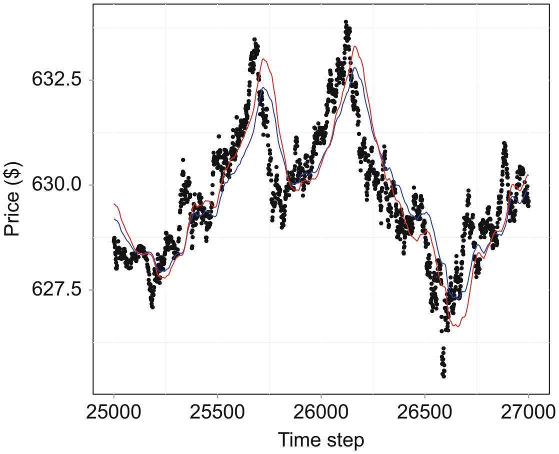


Fig. 11.4 Apple stock prices circa 2013 (points). Forecasts obtained from exponential smoothing are shown in *blue* and Holt-Winters forecasts are in *red* for $\tau = 20$. The root mean square forecasting error was estimated to be \$.710 and \$.705 for exponential smoothing and Holt-Winters, respectively

11.6.1 Forecasting Error

Forecasting error is estimated by comparing observations to forecasts. Whenever a new observation is observed, the difference between the forecast of the new value and the actual value provides a data point for error estimation. From the differences, the median absolute deviation or the root mean square error may be computed. How far in the future the forecasts are made will affect the accuracy, and so our estimator explicitly defines the mean square error as a function of τ:

$$\operatorname{Err}(\tau) = \frac{\sum_{n=\tau+2}^{N} (y_n - \widehat{y}_n)^2}{N - \tau - 1}. \tag{11.13}$$

The denominator is the number of terms in the sum. In this formulation, $\widehat{y}_n$ is a forecast of y_n *made* at time step $n - \tau$. Said another way, $\widehat{y}_{n+\tau}$ is a forecast of $y_{n+\tau}$ made at time step n. The first possible forecast (assuming that Holt-Winters forecasting is used) is made at time step $n = 2$ and the first evaluation of prediction error may be computed at time step $n = \tau + 2$ when the target value $y_{\tau+2}$ comes into view. Early forecasts are likely to be less accurate than later forecasts since the information content of few observations is less than a substantive number of observations. It's usually better to evaluate forecasting error using the root mean square error $\sqrt{\operatorname{Err}}(\tau)$ since the root mean square error is measured in the same units as the observations.

In a real-time application of Holt-Winters exponential forecasting, it's often desirable that the error estimates reflect recent performance rather measure average performance over a long history. Therefore, older forecasts may be given less weight in the error estimate by computing the error estimate using exponential weights. For example, the time step n estimator may be computed as

$$\mathrm{Err}_n(\tau) = \begin{cases} (y_n - \widehat{y}_n)^2, & \text{if } n = \tau + 2, \\ \alpha_e(y_n - \widehat{y}_n)^2 + (1 - \alpha_e)\mathrm{Err}_{n-1}(\tau), & \text{if } \tau + 2 < n. \end{cases} \tag{11.14}$$

where $0 < \alpha_e < 1$ is a tuning constant.

Forecasting errors produced by the competing forecasting functions (to be discussed in Sect. 11.8) are shown in Fig. 11.5. The figure gives the impression that the average forecast

$$\overline{y}_{n+\tau} = \frac{\widehat{y}^{\mathrm{A}}_{n+\tau} + \widehat{y}^{\mathrm{B}}_{n+\tau}}{2}$$

would out-perform both forecasting function with respect to forecasting error.

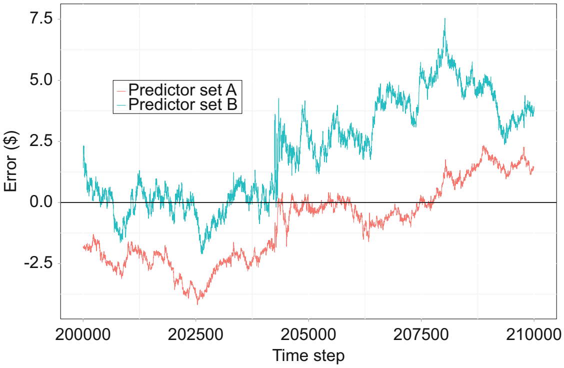


Fig. 11.5 Forecasting errors for a sequence of 10,000 time steps obtained from two linear regression prediction functions. Predictor set A consists of past Apple and Alphabet values ($R^2_{\text{adjusted}} = .924$) and set B consists of 18 stocks besides Apple ($R^2_{\text{adjusted}} = .952$)

11.7 Tutorial: Holt-Winters Forecasting

The National Association of Securities Dealers Automated Quotations (NASDAQ) is an online securities exchange, more commonly referred to as a stock market. The NASDAQ data stream is a sequence of updates on price quotations, bid and asked, that may be consumed by the public possessing the appropriate technology.[6] NASDAQ streams are widely used by automated trading systems. A key component to these systems are the algorithms that produce intra-day and sometimes intra-hour forecasts of future prices.

The objective of this tutorial is to implement the exponential and Holt-Winters forecasting algorithms for forecasting stock market prices. The reader is guided through the process using a static set of observations collected

[6] We'll do just this in the Chap. 12.

from the Yahoo Finance NASDAQ price quotation stream. The data were collected by requesting an update on the most recent bid price roughly every 30 s over the course of 2 months. The file name is `QuotesLong.txt`. It is space-delimited and contains prices for 27 well-known technology companies. Ticket symbols are shown below. As the data set is not large, `R` is used in this tutorial.

We will compute forecasts of bid price at each time step beginning with $n = 2$ using three functions: the sample mean, exponential weighting, and Holt Winters exponential forecasting. The sample mean represents the simplest of all data based forecasting functions and serves as a baseline against which to compare the other two methods.

1. Retrieve the data file `QuotesLong.txt` from the textbook website.
2. Read the data into a data frame named `D`. Assign column names to the data frame.

```
fileName = '../QuotesLong.txt'
D = read.table(fileName)
names = 'AAPL,ADBE,ADSK,AMAT,BMC,CA,CSCO,CTXS,DELL,GOOG,GRMN,
  INFY,INTC,INTU,LLTC,LRCX,MCHP,MSFT,MU,NVDA,ORCL,QCOM,SNDK,
  STX,SYMC,TXN,YHOO'
colnames(D) = unlist(strsplit(names,','))
```

   Each of the names is a symbol for a NASDAQ stock; for instance, AAPL is the symbol for Apple Inc. Each row in `D` is viewed as a vector of bid prices with one value for each of the 27 stocks. The object `strsplit(names,',')` is a `R` list of length one; the function `unlist` converts the list to a character vector of length $p = 27$. You can determine the internal structure of an `R` object using the function `str` (not be confused with the `Python` function by that name[7]).
3. Initialize matrices to store the estimated means, forecasts, and errors at each time step $n \in \{1, 2, \ldots, N\}$, where $N = 28,387$ denotes the number of time steps (and observation vectors in the dataframe `D`). Choose a stock, extract the corresponding column, and store the values in a vector `y` of length N.

[7] The `Python` function with the same purpose is named `type`.

```
N = dim(D)[1]
F = matrix(0,N,3)
muEst = matrix(0,N,3)
errMatrix = matrix(0,N,3)
colnames(F) = c('Mean','Exponential','Holt-Winters')
colnames(muEst) = c('Mean','Exponential','Holt-Winters')
colnames(error) = c('Mean','Exponential','Holt-Winters')
y = D$AAPL  # Our choice of stocks is Apple Inc
```

The instruction `F = matrix(0,N,3)` initializes the matrix for storing the forecasts.

4. Set the initial estimates to be y_1 for each method using the instruction `muEst[1,] = y[1]`.
5. The updating formula for the sample mean is

$$\overline{y}_n = n^{-1}\left[(n-1)\overline{y}_{n-1} + y_n\right].$$

Iterate over time step and store each update:

```
tau = 20
for (n in 2:(N - tau)){
  muEst[n,1] = ((n - 1)*muEst[n-1, 1] + y[n])/n
}
```

At the completion of the `for` loop, you'll have a series of estimates and observed values.

6. Plot a short sequence of the data and the sample means:

```
n = 200:1200
plot(n,y[n],pch=16,cex = .4)
lines(n,muEst[n,1])
```

The sample means do not form a straight line since the estimates are updated as n increases. Visually, $\overline{y}_n$ appears to be a poor technique for estimating the mean in the presence of drift.

7. Compute the exponentially weighted means after setting a value for α_e. The instruction for updating the estimator at time step n is

```
muEst[n,2] = a.e*y[n] + (1 - a.e)*muEst[n-1,2]
```

We use the `R` object `a.e` to store α_e. Set α_e to be a value between .01 and .05, add the calculation to the `for` loop (item 5) and execute the code.

8. Add the exponentially weighted estimates to the plot by executing the instructions in item 6 and the instruction `lines(n,muEst[n,2], col='red')`. You might repeat these calculations for a different value of α_e and plot with a different color.
9. In the next step, the Holt-Winters forecasting algorithm will be programmed. For now, choose the two tuning constants for Holt-Winters by setting the tuning constant for the mean (α_h) to be the same value as for exponential smoothing. Set the tuning constant for slope equation (α_b) to 0. By setting $\alpha_b = 0$, the Holt-Winters estimate will be the same as exponential smoothing, thereby providing a check on your code. Initialize a vector of length N to store the $\widehat{\beta}_n$'s, say, `beta= rep(0,N)`.
10. Code the updating formulas for Holt-Winters. This code segment immediately follows the computation of the exponentially weighted forecast (item 7).

```
muEst[n,3] = a.h*y[n]+ (1- a.h)*(muEst[n-1,3] + beta[n-1])
beta[n] = a.b*(muEst[n,3] - muEst[n-1,3]) + (1 - a.b)*beta[n-1]
```

    Compute the means for $n = 2, \ldots, N$ using Holt-Winters and add the values to your plot. Holt-Winters and exponential smoothing estimates should coincide. If the estimates are different, then there is an error in your code. When the estimates are the same, reset α_b to .01, recompute the estimates and plot again.
11. Set τ to be a value between 5 and 50. Compute and store the forecasts for a choice of τ. Change the termination step of the `for` loop so that it completes before the index n exceeds $N - \tau$. The `for` loop should look like this:

```
for (n in 2:(N - tau)){
    ...
    F[n+tau,1:2] = muEst[n,1:2]
    F[n+tau,3] = muEst[n,3] + beta[n]*tau
```

    Be sure to update `muEst` and `beta` before computing the forecasts.
12. Add the Holt-Winters forecasts to your plot using the instruction

```
lines(n,F[n,3],col='magenta')
```

    Larger values of τ result in poorer performance since there is less information in the observed data $y_1, \ldots, y_n$ about $y_{n+\tau}$.
13. Initialize `errMatrix` as a $N \times 3$ matrix of zeros to store the forecasting errors.

14. Compute the estimated root mean forecasting error for the three methods. We'll compute the terms contributing to the sum in Eq. (11.13) as the `for` loop iterates over n. So, within the `for` loop, compute and store the squared errors provided that $\tau + 2 \leq n \leq N - \tau$.

    ```
    if ((tau+2 <= n)&(n <= N-tau)) errMatrix[i,] = (y[n] - F[n,])^2
    ```

    Since we've stored the forecasts for time step $n+\tau$ at position $n+\tau$ in `F`, `y[n]` is the target of `F[n,]` and the forecasting errors are `y[n] - F[n,]`.
15. Upon completion of the `for` loop, compute and print the estimated root mean prediction error for the three methods, say,

    ```
    print(sqrt(colSums(errMatrix)/(N-2*tau))
    ```

16. Change the tuning constant associated with $\widehat{\beta}_n$ to a small but positive number, say .02. Try to reduce the estimated error for the Holt-Winters estimator by manipulating the tuning constants.
17. Keep the tuning constants fixed, compute $\sqrt{\text{Err}}(\tau)$ for $\tau \in \{5, 10, 20, 40, 80\}$, and plot $\sqrt{\text{Err}}(\tau)$ against τ for the three forecasting functions.

11.8 Regression-Based Forecasting of Stock Prices

Linear regression provides a somewhat different approach to forecasting compared to Holt-Winters and exponential forecasting. Rather than attempt to exploit recent trends in the underlying process to forecast the target variable, we'll simply use a prediction function constructed from concomitant predictor variables. The essential new aspect of the algorithm is that the forecasting function is fit to a data set in which the data pairs are staggered with respect to time step. For example, $(y_{n+\tau}, \mathbf{x}_n)$ is a staggered data pair, where n is a time step and τ is a positive integer. We can also describe the predictor vector as *lagged* relative to the target variable. Because the forecasting function is trained on targets observed τ time steps after the predictor vector was observed, the forecasting function yields predictions of the target τ time steps after the time step when the input predictor vector was realized. For example, if the current time step is n, then $f(\mathbf{x}_n|D) = \mathbf{x}_n^T\widehat{\boldsymbol{\beta}} = \widehat{y}_{n+\tau}$ is a forecast of $Y_{n+\tau}$. The vector $\widehat{\boldsymbol{\beta}}$ is computed as a solution to the problem of minimizing the sum of the squared differences between forecasts made at time step n and targets observed at time step $n + \tau$. The k-nearest-neighbor forecasting function built in Sect. 9.10 for forecasting the S&P 500 index used similarly staggered data pairs to train the prediction function.

The data set, therefore, consists of staggered pairs $(y_{n+\tau}, \mathbf{x}_n), n = 1, \ldots, N - \tau$, where $\mathbf{x}_n$ is a vector of variables measured at time step n. The rationale for the linear regression approach is the supposition that an underlying process is generating the realized target values. We postulate that there is information contained in the predictor vector $\mathbf{x}_n$ that's informative with respect to the future price of the target. Mathematically, this statement translates to saying that the conditional mean $\mathrm{E}(Y_{n+\tau}|\mathbf{x}_n)$ is approximately equal to $\mu_{n+\tau}$. If these ideas are reasonable, then a regression-based approach to the forecast problem is promising.

Stock market forecasting is a good application for the method because a collection of stocks usually can be identified that are similar in price movement to the target stock. For instance, we hypothesize that technology stocks prices tend to respond similarly to events in the financial world. We'll use a collection of technology stock prices observed at time step n to forecast one of the stock prices at time step $n + \tau$ in the following tutorial.

In the linear regression context, $\mathbf{x}_n^T\widehat{\boldsymbol{\beta}}$ is an optimal estimator of the expected future value, i.e., $\mathrm{E}(Y_{n+\tau}|\mathbf{x}_n) = \mathbf{x}_n^T\boldsymbol{\beta}$. Theoretically, the expected value is an optimal predictor of $Y_{n+\tau}$ having minimal mean square prediction error provided a number of conditions are met. For the problem at hand, we don't believe that the conditions hold or that $\mathrm{E}(Y_{n+\tau}|\mathbf{x}_n)$ is equivalent to the process mean, but instead speculate that a linear combination of past values may overcome some of the random variation in the target series and estimate the process mean with more accuracy than exponential or Holt-Winters forecasts. Certainly, the linear regression forecasting function involves more data and a forecasting function optimized to minimize the forecasting errors. In contrast, Holt-Winters involves two tuning constants α_h and α_b that are quite difficult to fit. It's reasonable to postulate that the information carried by the predictor vectors will translate to more information for forecasting.

11.9 Tutorial: Regression-Based Forecasting

This tutorial guides the reader through the development of a forecasting algorithm based on linear regression. The forecasting function is used to predict future prices of Apple Inc. from prices observed on $p = 27$ technology stocks. The data set consists of nearly 372,267 quotations on the technology-related stocks collected between October 2012 and January 2013. The data are less than perfect because on a few occasions the recording device did not suspend when the market closed. There were apparently a few occasions when the device did not restart when the market reopened. Consequently, there are sub-series exhibiting very abrupt changes.

Observations have been arranged in the data file so that an observation pair $(y_n, \mathbf{x}_{n-\tau}), n = \tau + 1, \tau + 2, \ldots, N$ is composed of y_n, an asking price for a share of Apple Inc. stock recorded $\tau =$ 10,000 time steps (or approximately 2.75 h) after the 27 stock quotations contained in the vector $\mathbf{x}_{n-\tau}$. By staggering the elements of the data pairs, the forecasting function computed from these data will predict the Apple asking price 10,000 steps ahead of the time step at which time the predictor vector was observed. Therefore, if the prediction function consumes a predictor vector observed at the current time or very nearly so, then the prediction will be for a price in the future.

In more detail, suppose that the current time step is n and a forecast of the price τ time steps ahead of n is to be computed. The forecast of the future price $Y_{n+\tau}$ is $\widehat{Y}_{n+\tau} = \mathbf{x}_n^T\widehat{\boldsymbol{\beta}}$ since the function has been trained on a data set for which each predictor vector $\mathbf{x}_n$ has been paired with target value $y_{n+\tau}$ observed τ time steps in the future.

If we think of each row of the data file as a vector, then the data set consists of the set of row vectors. Row vector $\mathbf{r}_n$ contains the elements of $(y_n, \mathbf{x}_{n-\tau})$, and so the data file consists of the following row vectors:

$$\begin{aligned}
\mathbf{r}_{1+\tau} &= [y_{1+\tau},\ \underbrace{x_{1,1},\ \ x_{1,2},\ \ \cdots x_{1,p}}_{\mathbf{x}_1}] \\
&\ \,\vdots \qquad\qquad\qquad \vdots \\
\mathbf{r}_n &= [y_n,\ \underbrace{x_{n-\tau,1},\ \ x_{n-\tau,2},\ \ \cdots x_{n-\tau,p}}_{\mathbf{x}_{n-\tau}}] \\
&\ \,\vdots \qquad\qquad\qquad \vdots \\
\mathbf{r}_N &= [y_N,\ \underbrace{x_{N-\tau,1},\ \ x_{N-\tau,2},\ \ \cdots x_{N-\tau,p}}_{\mathbf{x}_{N-\tau}}].
\end{aligned} \tag{11.15}$$

We'll include the past price of Apple Inc. in the predictor vector as it is presumed that the past Apple prices will be informative even with 26 other stocks included in the predictor vector.

Recall from Sect. 3.9.2, formulas (3.30) and (3.31), that the matrix form of $\widehat{\boldsymbol{\beta}}$ can be expressed as

$$\widehat{\boldsymbol{\beta}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} = \mathbf{A}^{-1}\mathbf{z},$$

where $\mathbf{A} = \mathbf{X}^T\mathbf{X}$ and $\mathbf{z} = \mathbf{X}^T\mathbf{y}$.

Let's introduce a subscript n since we will compute a forecasting function at time step n using the matrices $\mathbf{A}$ and $\mathbf{z}$ computed from the data pairs $(y_{1+\tau}, \mathbf{x}_1), \ldots, (y_n, \mathbf{x}_{n-\tau})$. At time step n, we compute $\mathbf{A}$ according to

$$\mathbf{A}_n = \sum_{i=1}^{n-\tau} \mathbf{x}_i\mathbf{x}_i^T. \tag{11.16}$$

The vector $\mathbf{z}_n$ is computed according to

$$\mathbf{z}_n = \sum_{i=1}^{n-\tau} \mathbf{x}_i y_{i+\tau}. \tag{11.17}$$

The last step of computing the forecasting function computes $\widehat{\boldsymbol{\beta}}_n = \mathbf{A}_n^{-1}\mathbf{z}_n$. This computation is carried out by passing $\mathbf{A}_n$ and $\mathbf{z}_n$ to the `Numpy` function `linalg.solve()`. The matrix $\mathbf{A}_n$ will not be full rank unless n is substantially larger than $p = 27$. Thus, $\widehat{\boldsymbol{\beta}}_n$ is not computed until $n > 1000$.

1. Retrieve `NASDAQ.csv` from the textbook website. The file is comma-delimited and contains no missing values. The first row contains the variable names. There are n = 372,267 rows and 28 columns. Each row is of the form shown in formulae (11.15). Set `path` to be the name of the file including the full path to the containing folder. Read the first line containing the stock ticker symbols of each (potential) predictor variable and print the names.

```
path = '../NASDAQ.csv'
with open(path) as f:
    variables = f.readline().split(',')
    for i, name in enumerate(variables):
        print(i, name)
```

   You can look up the company names at http://www.nasdaq.com/symbol/.
2. Import `Numpy` as `np`.
3. Choose any set of p variables and set up an index vector. This discussion assumes that the specification is `predictorIndex = [1, 10]`. Selecting the first and tenth predictor variables yields the past values of Apple Inc. and Alphabet Inc. (previously known as Google).
4. Before computing the matrices $\mathbf{A}$ and $\mathbf{z}$, it is necessary to initialize them as zero matrices. Initializations must take place before any records are processed. Set

```
p = len(predictorIndex)
A = np.matrix(np.zeros(shape=(p+1, p+1)))
x = np.matrix(np.ones(shape=(p+1, 1)))
z = np.matrix(np.zeros(shape=(p+1, 1)))
s = 0
```

   The variable `s` will store $\sum y_i^2$ so that the variance estimate $\widehat{\sigma}^2 = n^{-1}\sum y_i^2 - \overline{y}^2$ can be computed.
5. Process the data file by iterating over lines. Extract $y_{n+\tau}$ and fill the predictor vector $\mathbf{x}_n$ from the list `data`.[8]

[8] It's not necessary to stagger the data because the data set has been constructed from staggered data pairs.

```
for line in f:
    data = line.split(',')
    for k, pIndex in enumerate(predictorIndex):
        x[k+1] = float(data[pIndex])
    y = float(data[0])
```

The function `enumerate` instructs the `Python` compiler to generate an index `k` as the variable `pIndex` is drawn from the `Numpy` array `predictorIndex`. Whereas `pIndex` will take on the values stored in `predictorIndex`, `k` will take on the values $0, 1, \ldots, p$.

6. Update $\mathbf{A}$, $\mathbf{z}$, and s. The time step n is stored in $a_{1,1}$, the first row and first column element of $\mathbf{A}$. Extract n.

```
A += x*x.T
z += x*y
s += y**2
n = A[0,0]
```

7. If the time step n exceeds 1000, then compute the least squares estimator of $\boldsymbol{\beta}_n$. Print the parameter estimate associated with the past values of Apple assuming that the first element of `predictorIndex` is 1.

```
if n > 1000:
    betaHat = np.linalg.solve(A,z)
    if n%1000 == 0:
        print(int(n), round(float(betaHat[1]),2))
```

Again, assuming that the second column of $\mathbf{X}_n$ contains the past Apple prices, then the coefficient $\widehat{\beta}_{n,1}$ appears in the linear forecasting equation

$$\widehat{y}_{n+\tau} = \widehat{\beta}_{n,0} + \widehat{\beta}_{n,1} x_{n,\texttt{AAPL}} + \widehat{\beta}_{n,2} x_{n,\texttt{GOOL}}$$

as a multiplier of the past Apple price $x_{n,\texttt{AAPL}}$. Thus, a one unit change in $x_{n,\texttt{AAPL}}$ results in a $\widehat{\beta}_{n,1}$ change in the forecasted price.

8. Compute R^2_{adjusted} (formula (3.36)) as the program iterates over the file by coding the following equations:

$$\widehat{\sigma}^2 = n^{-1}\sum_i y_i^2 - \overline{y}^2$$
$$\widehat{\sigma}^2_{\text{reg}} = \frac{\sum y_i^2 - \mathbf{z}^T\widehat{\boldsymbol{\beta}}}{n},$$
$$\text{and } R^2_{\text{adjusted}} = \frac{\widehat{\sigma}^2 - \widehat{\sigma}^2_{\text{reg}}}{\widehat{\sigma}^2}.$$

9. Print R^2_{adjusted} and $\widehat{\sigma}_{\text{reg}}$ every hundred iterations.

```
if n%100 == 0 and n < 372200:
    s2 = s/n - (float(z[0])/n)**2
    sReg = s/n - z.T*betaHat/n
    rAdj = 1 - float(sReg/s2)
    print(int(n), round(float(betaHat[1]), 2),
         round(np.sqrt(float(sReg)), 2), round(rAdj, 3))
```

Allow the script to execute until all records in the data file have been processed.

10. The last step of operations will compute and save a subset of forecasts, target values, and forecasting errors from which a couple of plots will be constructed. The forecasting error $e_{n+\tau}$ associated with the time step n forecast is the difference between the future value $y_{n+\tau}$ and the forecast $\widehat{y}_{n+\tau} = \mathbf{x}_n^T\widehat{\boldsymbol{\beta}}_n$. Specifically, the forecasting errors are

$$e_{n+\tau} = y_{n+\tau} - \mathbf{x}_n^T\widehat{\boldsymbol{\beta}}_n, n = k, k+1, \ldots, N, \tag{11.18}$$

where k is the time step of the first forecast.

Create a dictionary to store the errors computed over some range of time steps, say 190,000 through 210,000.

```
indices = np.arange(190000,210000)
errDict = dict.fromkeys(indices,0)
```

The dictionary keys correspond to the time steps for which the errors are going to be computed and saved. The dictionary must be created before the data file is processed since it will be filled as the data are processed.

11. Insert the following code segment in the `for` loop.

```
try:
    errDict[n] = (y, float(x.T*betaHat), y - float(x.T*betaHat))
except(keyError):
    pass
```

If n is not a key in the dictionary, then the attempt to store the three-tuple will cause a `KeyError` exception that is trapped by the exception handler.

In a real-time application, the code segment should execute after extracting $\mathbf{x}_n$ and $y_{n+\tau}$ from the data file and before $\widehat{\boldsymbol{\beta}}_n$ is computed. This order simulates what would happen in practice: the forecast is made before the forecasting target comes into view and the error is computed after the forecasting target comes into view. Because we cannot compute $\widehat{\boldsymbol{\beta}}_n$ without the target $y_{n+\tau}$, we would use the previous time step estimate of $\boldsymbol{\beta}$ and compute $\widehat{y}_{n+\tau} = \mathbf{x}_n^T\widehat{\boldsymbol{\beta}}_{n-1}$.

12. After the data file is completely consumed, create a figure showing the forecasts and the target values.

```
import matplotlib.pyplot as plt
forecastLst = [errDict[key][1] for key in indices]
targetLst = [errDict[key][0] for key in indices]
plt.plot(indices, forecastLst)
plt.plot(indices, targetLst)
```

Figure 11.5 is an example comparing errors generated by two different prediction functions.

11.9.1 Remarks

An essential aspect of the forecasting function is the use of staggered data for training. By staggering the data pairs, the forecasting function has been optimized for forecasting τ time steps in the future. Linear regression insured that the forecasting function possessed the optimality property of least squares. The mean squared difference between the forecasted and observed target values was as small as possible because each predictor vector $\mathbf{x}_n$ has been paired with a target value $y_{n+\tau}$ observed τ time steps in the future. Yet, the root mean squared forecasting error was relatively large.

To understand why, let's suppose that the model consists of a single predictor containing the past values of the target, say $\mathrm{E}(Y_{n+\tau}|\mathbf{y}_n) = \beta_0 + \beta_1 y_n$. If the data series exhibits positive trend, then $\widehat{\beta}_1$ ought to be greater than one, and if the trend is downward, then $\widehat{\beta}_1$ ought to be less than one. In the Apple price series, the mean level appears to drift in both directions (though not at the same time). At a time step n distant from the start of the series, the time step n estimate of $\widehat{\beta}_1$ will reflect overall trend, and not necessarily recent trend. As a result, the forecasting function is not terribly accurate. The root mean square error $\widehat{\sigma}_{\text{reg}}$ manifested the problem—it grew larger as n grew larger. The problem is drift—the price-generating process changed over time and consequently, the relationship between the future value of Y

and the present value of $\mathbf{x}$ changed. A forecasting function built from a long series of data pairs often is inaccurate.

A solution to the drift problem is to allow the regression parameters to vary in response to recent changes in the underlying process. The implementation of this approach is a remarkably simple fusion of exponential weighting and regression. We develop the approach in the next section.

11.10 Time-Varying Regression Estimators

Suppose that the process generating the data is subject to drift. Not only may the mean level of the target variable Y drift, but the relationship between the expected value of $Y_{n+\tau}$ and the associated predictor vector $\mathbf{x}_n$ may change. Therefore, the model $\mathrm{E}(Y_{n+\tau}|\mathbf{x}_n) = \mathbf{x}_n^T\boldsymbol{\beta}, n = 1, 2, \ldots,$ is incorrect. A different linear model describes $\mathrm{E}(Y_{n+\tau}|\mathbf{x}_n)$ for different values of n, let's say,

$$\mathrm{E}(Y_{n+\tau}|\mathbf{x}_n) = \mathbf{x}_n^T\boldsymbol{\beta}_n, n = 1, 2, \ldots, \tag{11.19}$$

Conceptually, Eq. (11.19) is preferable to the model with a single parameter vector $\boldsymbol{\beta}$. From a practical standpoint, Eq. (11.19) is difficult to work with since it's impossible to fit a separate linear model using a single pair of observations $(y_{n+\tau}, \mathbf{x}_n)$.

To formulate an approximation of the relationship between $\mathrm{E}(Y_{n+\tau}|\mathbf{x}_n)$ and $\mathbf{x}_n$ given drift, we presume that the parameter vector, or the model, changes slowly relative to the time step. In other words, the rate of incoming pairs $(y_i, \mathbf{x}_i)$ is rapid enough that chronologically near observation pairs $(y_i, \mathbf{x}_i)$ and $(y_j, \mathbf{x}_j)$ are similar with respect to the parameter vectors $\boldsymbol{\beta}_i$ and $\boldsymbol{\beta}_j$ describing the expectations of Y_i and Y_j as functions of $\mathbf{x}_{i-\tau}$ and $\mathbf{x}_{j-\tau}$. Therefore, we may use chronologically near observations to estimate $\boldsymbol{\beta}_n$, albeit with some small degree of error. Logically, the observations that are nearest chronologically to $(y_n, \mathbf{x}_n)$ ought to have the largest influence in the computation of $\boldsymbol{\beta}_n$.

Once again, we turn to exponential weighting to assign the largest weight to $(y_n, \mathbf{x}_{n-\tau})$ at time n and successively smaller weights to $(y_t, \mathbf{x}_{t-\tau})$ as the time step t recedes in the past. Recall that Eq. (11.5) expressed the exponentially weighted estimator of μ_n as a linear combination of the most recent observation and the previous estimate of μ, say, $\widehat{\mu}_n = \alpha y_n + (1-\alpha)\widehat{\mu}_{n-1}$.

We allow the parameter estimator $\widehat{\boldsymbol{\beta}}_n = \mathbf{A}_n^{-1}\mathbf{z}_n$ to vary in response to the most recent observation pair by modifying the definition of $\mathbf{A}_n$ and $\mathbf{z}_n$. Specifically, the new terms, $\mathbf{A}_n^*$ and $\mathbf{z}_n^*$, are computed as linear combinations of the most recently observed pair $(y_n, \mathbf{x}_{n-\tau})$ and the previous time step values. The *exponentially weighted* versions of $\mathbf{A}$ and $\mathbf{z}$ at time step n are

$$\begin{aligned} \mathbf{A}_n^* &= (1-\alpha)\mathbf{A}_{n-1}^* + \alpha\mathbf{x}_{n-\tau}\mathbf{x}_{n-\tau}^T \\ \mathbf{z}_n^* &= (1-\alpha)\mathbf{z}_{n-1}^* + \alpha\mathbf{x}_{n-\tau}y_n, \end{aligned} \tag{11.20}$$

where $0 < \alpha < 1$ is the tuning constant controlling the importance of the most recent pair towards the time-varying estimator $\widehat{\boldsymbol{\beta}}^*_n$. Lastly, the estimate $\widehat{\boldsymbol{\beta}}^*_n$ is computed according to

$$\widehat{\boldsymbol{\beta}}^*_n = \mathbf{A}^{*\,-1}_n \mathbf{z}^*_n. \tag{11.21}$$

The forecast of $Y_{n+\tau}$ is $\widehat{y}_{n+\tau} = \mathbf{x}_n^T \widehat{\boldsymbol{\beta}}^*_n$.

11.11 Tutorial: Time-Varying Regression Estimators

We continue with the Apple price forecasting problem of Sect. 11.9. There's only a few modifications to be made of the Sect. 11.9 Python script. The only substantive change is to replace the updating equations for $\mathbf{A}_n$ and $\mathbf{z}_n$ with the updating equations for $\mathbf{A}^*_n$ and $\mathbf{z}^*_n$. The revised updating formulae are Eq. (11.20).

We won't be able to compute the variance estimate $\widehat{\sigma}^2 = n^{-1} \sum y_i^2 - \overline{y}^2$ from the terms in $\mathbf{A}^*$ and $\mathbf{z}^*$. For example, $a_{1,1}$ (the element in row one and column one of $\mathbf{A}$) contained the numbers of data pairs used in the calculation of $\widehat{\boldsymbol{\beta}}$. Now $a^*_{1,1}$ contains the sum of the exponential weights, and hence has the value 1. Instead, we'll use the data stored in the dictionary `errDict` to compute error estimates. The dictionary will contain the target values and the forecasts. The mean squared forecasting error $\widehat{\sigma}^2_{\text{reg}}$ and R^2_{adj} will be computed at the completion of the `for` loop using the terms stored in `errDict`.

There's a undesirable side-effect of exponential weighting for this problem. The matrix $\mathbf{A}^*_n$ may be singular, in which case, the call to the `Numpy` function `linalg.solve` returns an error instead of the solution to the equation $\mathbf{A}^*_n \boldsymbol{\beta} = \mathbf{z}^*_n$. The solution, had it existed, would be $\widehat{\boldsymbol{\beta}}^*_n$. Larger values of α increase the likelihood of the matrix being singular.

1. Initialize the tuning constant α, `errDict`, and the predictor variable indexes.

```
a = .005
errDict = {}
predictorIndex = [1,10]
```

   We're using past Apple values and Google (now Alphabet Inc) for forecasting.

2. Add a counter to the `for` loop:

```
for counter, line in enumerate(f):
    data = line.split(',')
    for k, pIndex in enumerate(predictorIndex):
        x[k+1] = float(data[pIndex])
```

3. Replace the variable `n` with the counter throughout the program. Remove the instruction `n = A[0,0]`.
4. Replace the updating equations for `A` and `z` with

```
A = a*x*x.T + (1 - a)*A
z = a*x*y +(1 - a)*z
```

5. Begin computing forecasts when at least 5000 observations have been processed.

```
if counter >= 5000:
    try:
        yhat = float(x.T*betaHat)
        errDict[counter] = (y, yhat)
    except:
        pass
    betaHat = np.linalg.solve(A,z)
    if counter%1000 == 0 and counter < 327200:
        print(counter, round(float(betaHat[1]), 2))
```

   The exception handler avoids the error of attempting to compute a forecast before $\widehat{\boldsymbol{\beta}}^*$ as been computed.
6. When the `for` loop has finished executing, compute $\widehat{\sigma}^2$ and $\widehat{\sigma}^2_{\text{reg}}$. Compute R^2_{adj} and print the results.

```
s2 = np.var([v[0] for v in errDict.values()])
sReg = np.mean([(v[0] - v[1])**2 for v in errDict.values()])
rAdj = 1 - float(sReg/s2)
print(s2,sReg,rAdj)
```

7. Plot a sequence of 10,000 forecast values and observations, say

```
indices = np.arange(300000,310000)
forecastLst = [errDict[key][1] for key in indices]
targetLst = [errDict[key][0] for key in indices]
plt.plot(indices, forecastLst)
plt.plot(indices, targetLst)
```

11.11.1 Remarks

The accuracy estimates obtained from the time-varying regression forecasting estimates are remarkably good, $\widehat{\sigma}_{\text{reg}} = \$.402$ and $R^2_{\text{adj}} = .999$ versus $\widehat{\sigma}_{\text{reg}} = \$4.82$ and $R^2_{\text{adj}} = .919$ obtained from the conventional linear regression forecasting function. Such a large difference deserves explanation.

The time-varying regression approach to forecasting incorporates the best components of the least squares and exponential weighting forecasting algorithms. The exponential weighting component introduces recency into the least squares objective function. The regression component of the solution provides a tractable method of optimizing the forecasting function. The objective function is

$$S(\boldsymbol{\beta}_n) = \textstyle\sum_{i=1}^{n} w_{n-i} \left(y_{i+\tau} - \mathbf{x}_i^T \boldsymbol{\beta}_n\right)^2, \tag{11.22}$$

where $w_{n-i} = \alpha(1-\alpha)^{n-i}$ for $i \in \{1, 2, \ldots, n\}$. The objective is to minimize $S(\boldsymbol{\beta}_n)$ by the choice of $\boldsymbol{\beta}_n$. Minimizing $S(\boldsymbol{\beta}_n)$ yields the smallest possible sum of the weighted forecasting errors. Because the largest weights are assigned to the most recent errors, the solution $\widehat{\boldsymbol{\beta}}_n^* = \mathbf{A}_n^{*\,-1}\mathbf{z}_n^*$ yields the smallest possible sum of recent forecasting errors.[9] All of the data are used in the calculation of the forecasting function $f(\mathbf{x}_n|D) = \mathbf{x}_n^T\widehat{\boldsymbol{\beta}}_n^*$. At the same time, the forecasting function is determined to a greater extent by recent history at the expense of the more distant history.

The algorithm is computationally efficient because $\mathbf{A}_n^*$ and $\mathbf{z}_n^*$ are rapidly updated when a new observation pair comes into view, and $\widehat{\boldsymbol{\beta}}_n^*$ is easily computed from $\mathbf{A}_n^*$ and $\mathbf{z}_n^*$. Consequently, the algorithm may be used to process high-speed data streams in real-time. Real-time analytics are the subject of the next chapter.

11.12 Exercises

11.12.1 Conceptual

11.1. Verify that Eq. (11.5) can be derived from Eq. (11.4).

11.2. Consider exponential smoothing. Argue that if α is small, then the weights decrease to 0 approximately linearly as $t \to 0$.

11.3. Determine the vector $\boldsymbol{\beta}$ that optimizes Eq. (11.22).

[9] This statement is mathematically vague—the influence of recent errors on the determination of $\widehat{\boldsymbol{\beta}}_n^*$ depends on α.

11.12.2 Computational

11.4. Download data on the urban Consumer Price Index from the US. Bureau of Labor Statistics. The URL is

https://research.stlouisfed.org/fred2/series/CPIAUCSL/downloaddata.

These data are monthly values of percent change in a measure of the cost of food, clothing, shelter, and fuels relative to a initial date. Large changes in the Consumer Price Index indicate economic inflation or deflation. Using these data, compare the performance of exponential and Holt-Winters prediction. Also, compare these prediction functions to a time-varying least squares regression function that uses time as a predictor variable. To get started, read that data and convert the character representation of month to the date format:

```
D = read.table('ConsumerPriceIndex.txt',header=TRUE)
D = data.frame(as.Date(D[,1]),D[,2])
colnames(D) = c("Year",'Index')
```

Don't make any predictions using the regression predictor until at least 50 observations have been processed and accumulated in the matrix $\mathbf{A}_n^*$ and the vector $\mathbf{z}_n^*$. The `for` loop should begin as follows:

```
x = matrix(0,2,1)
for (n in (tau+1):N){
  x[1] = n - tau
```

a. Compute and plot the auto-correlation function using the R function `acf`. Describe and interpret the pattern of autocorrelation. What does the pattern suggest with respect to trend in Consumer Price Index?
b. Compute the root mean prediction error for $\tau \in \{12, 24\}$ months. Good initial values for the smoothing constants are between .01 and .1. Find good values of each smoothing constant. Construct a table showing the root mean prediction error realized by the three methods and for the two values of τ. Report the values that were used as smoothing constants.
c. Construct a figure showing the Consumer Price Index and the three sets of predictions plotted against month. The following code may be helpful:

```
library(ggplot2)
interval = 51:N
df = data.frame(D[,'Year'],y,pred)[interval,]
colnames(df) = c('Year','Index',colnames(pred))
plt = ggplot(df, aes(Year,Index)) + geom_point()
plt = plt + geom_line(aes(Year,Exponential.smooth), color = 'blue')
plt = plt + geom_line(aes(Year,Holt.Winters), color = 'red')
plt = plt + geom_line(aes(Year,LS),color = 'green')
```

11.5. Compute the root mean square forecasting error for three predictive models using the `NASDAQ.csv` data set:

$$\begin{aligned} \mathrm{E}(Y_{n+\tau}) &= \beta_1^* n \\ \mathrm{E}(Y_{n+\tau}) &= \beta_1^* x_{n,\texttt{AAPL}} \\ \mathrm{E}(Y_{n+\tau}) &= \beta_1^* x_{n,\texttt{GOOG}} \end{aligned} \tag{11.23}$$

where $x_{n,\texttt{AAPL}}$ and $x_{n,\texttt{GOOG}}$ denotes the Apple Inc. and Alphabet (Google) Inc. price for step n. Determine a good choice of α for each of the three models.

Chapter 12
Real-time Analytics

Abstract Streaming data are data transmitted by a source to a host computer immediately after being produced. The intent of real-time data analytics is to analyze the data as they are received and at a rate sufficiently fast to keep up with the stream. Analyses of streaming data center about characterizing the current level of the data-generating process, forecasting future values, and determining whether the process is undergoing unexpected change. In this chapter, we focus on computational aspects of real-time analytics. Methods were discussed in Chap. 11. The tutorials guide the reader through the analysis of data streams originating from two public and very different sources: tweets originating from the Twitter API and stock quotations originating from the NASDAQ.

12.1 Introduction

A data stream is a series of observations that are received by a device (for example, a computer) for a sustained period of time. The stream is generated by a device that automatically produces or sends the data to the receiver. Self-driving cars are fantastic examples of the generation and use of streaming data. A variety of sensors collect information regarding physical location, speed, position on the roadway, and nearby objects and their movement. The data are sent from the sensors to the guidance system as data streams. As much as a gigabyte of data per second may be sent by the sensors. Consequently, data processing must be rapid, and saving all of the data is neither necessary nor practical.

The Twitter stream is accessible to the public with only the minor inconvenience of requesting authorization from Twitter Inc. A Twitter data stream consists of a series of text messages sent by the Twitter application programming interface (API) to a host computer. A tweet appearing in the Twitter stream consists of the familiar short text message and, less familiar to most

B. Steele et al., *Algorithms for Data Science*,
DOI 10.1007/978-3-319-45797-0_12

people, a remarkably large number of other variables related to the origin of the tweet. A connection to the public Twitter stream will receive perhaps 100 tweets per second depending on the time of day and the size of the geographic area from which tweets are directed.

Streaming data, or *data in motion* differ from static data, or *data-at-rest*, by origin and objective. The goal of real-time analytics is to analyze data streams at a rate equal to or greater than the arrival rate. In most cases, analysis involves re-computing a few simple statistics whenever an observation is received, or better, updating the statistics of interest whenever new observations arrive. The goal of this continual analysis is to make decisions and take actions as rapidly as possible, say at the same rate as the data are received.

Another publicly accessible data stream are stock prices retrieved from the Yahoo Financial application programming interface. The data are quotations originating from any of the approximately 3100 companies listed with the NASDAQ stock exchange. The stream is generated by the local or host computer by repeatedly sending requests to the Yahoo Financial API for a price quotation. The analytics applied to the Yahoo Finance stream are aimed at computationally fast forecasting as would be appropriate for intra-day trading and automated stock trading.

This chapter discusses real-time analytics by example. The first example, the NASDAQ quotation stream, consists of quantitative values. The second example, the Twitter stream, consists of text messages. The objectives and the analytics applied to the streams are, of course quite different.

12.2 Forecasting with a NASDAQ Quotation Stream

We begin with a quantitatively-valued data stream of stock prices generated from the Yahoo Finance API (http://finance.yahoo.com/). Yahoo Finance collects stock quotations from the NASDAQ stock market and provides the information as a public service through their website. Most users interact with the website by point-and-click navigation. Computer-generated requests for stock quotations are also handled by the website. When the requests are correctly formatted, responses are returned to the requesting computer. The following tutorial uses a `Python` script to repeatedly ask for price quotations. The series of requests generates a stream of responses in the form of price quotations.

A trader may buy shares of a stock from a seller by paying the asking price and may sell the stock to buyer for the bid price. The data sent by the Yahoo Financial API in response for a quotation contains the most recent bid price and asked price and a number of other attributes. Of course, the bid and asked prices of a stock fluctuates and drifts over time. The rate of change in the most recent price depends on traders' interest level in the stock.

Quotations originating from the NASDAQ source are received by and rapidly updated by Yahoo Finance when the market is open for trading. There is however, a delay of perhaps 15 min between the time that a stock price changes on the NASDAQ and the time that its price is updated on Yahoo Finance. Consequently, the data stream received by the `Python` script is not a real-time stream from NASDAQ. However, the application of forecasting algorithms to the delayed NASDAQ quotation data stream does approximate a real-time analysis and is useful for developing a forecasting function for use with a real-time stream. Since multiple requests for quotations may be sent in a single second to the Yahoo Finance, the velocity of the generated stream of responses is reasonably good.

The `Python` script developed in the following tutorial will repeatedly send Hypertext Transfer Protocol (HTTP) requests to Yahoo Finance for quotations on a stock of interest to generate a stream of asking prices. We'll use a third-party module `ystockquote` [22] to make the request and extract the price from the JSON object (JavaScript Object Notation) returned from Yahoo Finance. Keep in mind that new quotations can be received only when the market is open. Most trading takes place during the weekday market hours of 9:30 a.m. to 4:00 p.m. Eastern time; the remainder occurs during pre- and post-market trading periods spanning 4:00 a.m. to 8:00 p.m.

The tutorial assists in developing `Python` code to support the following hypothetical enterprise. A trader is holding stocks $S_1, \ldots, S_k$ at time step τ. Forecasts are made of each stock at time step $n + \tau$. The stock with the smallest forecasted change in price is sold at time step n and the proceeds are used to buy shares of the stock with the largest forecasted increase in asking price.

12.2.1 Forecasting Algorithms

Two algorithms will be used for forecasting. We'll use Holt-Winters exponential forecasting and linear regression with time-varying coefficients.

Recall from the discussion of Sect. 11.6 that at time step n, the Holt-Winters forecast of the target $y_{n+\tau}$ is

$$\widehat{y}^{\text{HW}}_{n+\tau} = \widehat{\mu}_n + \tau \widehat{\beta}^{\text{HW}}_n. \tag{12.1}$$

The term $\widehat{\mu}_n$ is the estimated mean level of the stock, free, in principle, of short-term variation, and $\widehat{\beta}^{\text{HW}}_n$ is the estimated rate of change of the process generating the sequence of prices $\ldots, y_{n-1}, y_n, \ldots$. Whenever a price quotation is received from the Yahoo Financial API, the estimates $\widehat{\mu}^{\text{HW}}_n$ and $\widehat{\beta}^{\text{HW}}_n$ are updated according to Eqs. (11.10) and (11.12). Then, the forecast

$$\widehat{y}^{\text{HW}}_{n+\tau} = \widehat{\mu}^{\text{HW}}_n + \tau \widehat{\beta}^{\text{HW}}_n \tag{12.2}$$

is computed. When $y_{n+\tau}$ arrives at τ steps after the present step, the forecast error $y_{n+\tau} - \widehat{y}_{n+\tau}^{\mathrm{HW}}$ will be computed.

The second forecasting algorithm is linear regression with time-varying coefficients (Sect. 11.10).

The time-varying slope $\boldsymbol{\beta}_n^{\mathrm{LR}}$ is estimated by computing

$$\widehat{\boldsymbol{\beta}}_n^{\mathrm{LR}} = \mathbf{A}_n^{*\,-1}\mathbf{z}_n^*$$

(Eq. (11.21)). The statistics $\mathbf{A}_n^*$ and $\mathbf{z}_n^*$ are updated by computing a weighted sum of the past time step value of the statistics and terms computed from the most recent observations (Eq. (11.20)).

The linear regression forecasting function uses time step as the predictor variable in the tutorial. At time step n, the predictor is simply $x_n = n$, and the forecasting model is

$$\mathrm{E}(Y_{n+\tau}|\mathbf{x}_n) = \beta_{n,0} + n\beta_{n,1}.$$

The data pair used to fit the model at time step n is $(y_n, x_n) = (y_n, n - \tau)$ and the predictor vector at time step n is $\mathbf{x}_n = [1 \quad n]^T$. A comparison of Holt-Winters to time-varying linear regression forecasting is on more level ground without the advantage of exogenous information carried by others stocks. The time-varying linear regression forecasting method still has the advantage of being optimized for prediction via least squares whereas Holt-Winters depends on a wise choice of tuning parameters α_e and α_b. In practice, an analyst will compute forecasting error using each pair (α_e, α_b) in a set of candidate pairs to find a good pair of tuning constants. The search for best tuning constants should be conducted on a regular basis.

12.3 Tutorial: Forecasting the Apple Inc. Stream

The `Python` script will repeatedly send requests to the Yahoo Financial API for the current quote of Apple Inc. If the returned value has changed from the previous response, then the forecasts are updated. If there's no change in value, the program waits for a short time, and sends another request. When a forecasted value comes into view, the time step n forecast errors are computed and estimates of forecasting error are computed. (Recall that the time step n forecast was made at time step $n - \tau$.) When $n = 1000$ values have been received, the program terminates.

We'll use Corey Goldberg's `Python` module `ystockquote` [22] to send quotation requests to Yahoo Finance. The first task, then, is to install the module by submitting the instruction `pip install ystockquote` from within a Linux terminal or a Windows command prompt window.

The vast majority of trading takes place during the market hours of 9:30 a.m. to 4:00 p.m., Eastern, on weekdays. Trading also occurs during pre- and post-market trading periods spanning 4:00 a.m. to 8:00 p.m. No data is streamed outside of the interval 4:00 a.m. to 8:00 p.m. The script will only generate a stream between 4 a.m. and 8 p.m. on trading days.

Upon arrival of a price y_n not equal the previous price y_{n-1}, the two forecasts of the future value at time step $n+\tau$ will be computed. To assess the performance of the algorithm, at time step n, for $n = \tau+1, \ldots, N$, the target price y_n and forecasts of the future target $y_{n+\tau}$ will be saved in a dictionary. The forecasts of $y_{n+\tau}$ are $\widehat{y}^{\text{HW}}_{n+\tau}$ and $\widehat{y}^{\text{LR}}_{n+\tau}$. Time step is used as the dictionary key and so y_n is stored with the key n. The forecasts $\widehat{y}^{\text{HW}}_{n+\tau}$ and $\widehat{y}^{\text{LR}}_{n+\tau}$ are stored with the key $n+\tau$.

When program execution has completed, the dictionary entries are used to compute the root mean square forecasting errors $\widehat{\sigma}_{\text{HW}}$ and $\widehat{\sigma}_{\text{LR}}$. A plot of the three series provides visual information on the source of differences between the two estimates of forecasting error. For example, Figure 12.1 shows that the Holt-Winters and time-varying linear regression forecasts appear to be very similar. However, there is some difference in the estimated forecasting error since $\widehat{\sigma}_{\text{HW}} = \$.0766$ and $\widehat{\sigma}_{\text{LR}} = \$.0585$. We made no attempt to find better tuning constants than the first choice of $\alpha_r = \alpha_e = \alpha_b = .1$.

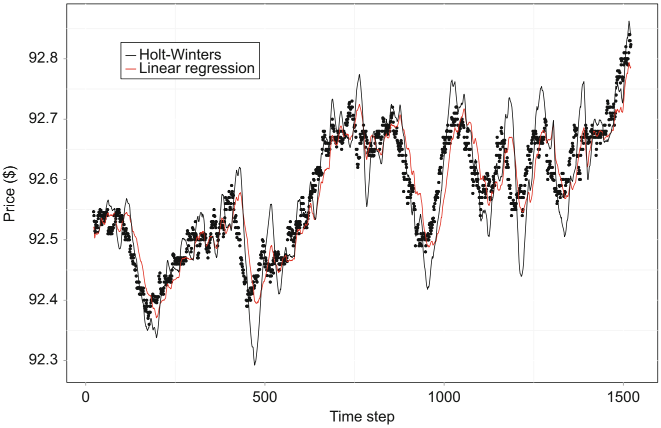


Fig. 12.1 Observed and forecasted prices of Apple Inc. stock. Forecasts were computed for $\tau = 20$ time steps ahead of the current time step. The tuning constants were set to the same value, $\alpha_r = \alpha_e = \alpha_b = .1$

1. Create a `Python` script and import the modules `sys`, `time`, `numpy` and `ystockquote`.
2. Choose a stock for forecasting and set the variable `symbol` to be the ticket symbol, say `symbol = 'AAPL'`. A list of the most actively trading stocks can be viewed at the webpage http://www.nasdaq.com/symbol/.

3. Initialize the tuning constants according to $\alpha_r = \alpha_h = \alpha_b = .1$. Set $\tau = 5$ for now. Once the code appears to work, set $\tau = 20$. Initialize a dictionary `sDict` to store the forecasts and the target values. Initialize a variable to store the price of the previous quotation. Set the time step counter `n` to be zero.

```
ar = ae = ab = .1
tau = 5
sDict = {}
previousPrice = 0
n = 0
```

4. Initialize the terms used in the computation of the time-varying linear regression forecast.

```
p = 1
A = np.matrix(np.zeros(shape=(p+1, p+1)))
z = np.matrix(np.zeros(shape=(p+1, 1)))
x = np.matrix(np.ones(shape=(p+1, 1)))
```

5. Use a `while` loop to request quotations.

```
while n <= 1000 :
    try:
        print('Sending request .. ')
        data  = ystockquote.get_price(symbol)
        dataReceived = True
        print(data)
    except:
        print('Error')
        time.sleep(10)
        dataReceived = False
```

   Transmission errors are possible. An exception handler is used to catch transmission errors. If there is an error, then the program idles for 10–20 s before resuming execution.

6. If the request for a quote has returned a price, then translate the string value to a floating point value. If the time step variable is zero, then initialize the Holt-Winter estimates of the mean to be the received value and set $\beta_0^{\text{HW}} = 0$. Test whether the received value `y` has changed since the last request. If it has, then the forecasting operations will commence with the next instruction.

```
if dataReceived:
    y = float(data)
    if n == 0:
        mu = y
        muOld = y
        beta = 0
    if y != previousPrice:
        n += 1
        previousPrice = y
```

This code block executes for every iteration of the `while` loop, so indent the segment accordingly.

The variable n counts the time step. The first time step is $n = 1$ since time step is advanced before computing the forecasts.

7. We have not yet programmed the updating algorithms for the calculation of the forecasts $\widehat{y}_n^{\text{HW}}$ and $\widehat{y}_n^{\text{LR}}$. However, we will create the code for appending the new value y_n to the dictionary entry $[\widehat{y}_n^{\text{HW}}, \widehat{y}_n^{\text{LR}}]$ at this point.

   Keep in mind that forecasts are stored with the future time step as the key. However, an entry with the key n will exist only if $n > \tau$ because the entry would have been created at time step $n - \tau$ (instruction 10 below). So, if $n > \tau$, then a dictionary entry with the current time step n exists in `sDict` and we should append `y` to the dictionary entry with key n.

```
    try:
        sDict[n].append(y)
    except(KeyError):
        pass
```

   The code segment follows directly after the assignment `previousPrice = y`. The code segment and the assignment must have the same indentation.

8. Update $\widehat{\mu}_n^{\text{HW}}$ and $\widehat{\beta}_n^{\text{HW}}$ and compute the Holt-Winters forecast of $y_{n+\tau}$.

```
mu = ae*y+ (1- ae)*(mu + beta)
beta = ab*(mu - muOld) + (1 - ab)*beta
HWforecast = mu + beta*tau
muOld = mu
```

Only update $\widehat{\mu}_n^{\text{HW}}$ and $\widehat{\beta}_n^{\text{HW}}$ if `y` has changed since the previous time step so take care with indentation.

9. Compute the predictor vector $\mathbf{x}_n$ by updating the second element. Update the statistics $\mathbf{A}_n^*$ and $\mathbf{z}_n^*$.

```
x[1] = n - tau
A = ar*x*x.T + (1 - ar)*A
z = ar*x*y + (1 - ar)*z
```

10. Compute the least squares estimator of $\boldsymbol{\beta}_n^*$ if $n > 1$. At least two predictor vectors must be accumulated in $\mathbf{A}_n^*$ before it is nonsingular. If $\mathbf{A}_n^*$ is singular, the attempt to compute the estimator will produce an error.

```
if n > 1:
    betaHat = np.linalg.solve(A, z)
    LRforecast = float(x.T*betaHat)
    x[1] = n
    sDict[n + tau] = [HWforecast, LRforecast]
```

The list `[HWforecast, LRforecast]` is stored with the future time step key $n+\tau$ since each forecast is a prediction of $y_{n+\tau}$. We'll have to wait τ time steps until the actual value $y_{n+\tau}$ comes into view before we can compute the forecasting errors associated with the two forecasts. At that time step $(n+\tau)$, we attach the price to the `sDict` entry just created (instruction 7). The next item computes the estimates of root mean square prediction error.

11. Compute the mean square forecasting errors for the two forecasting functions. For example, the error estimate for Holt-Winters is

$$\widehat{\sigma}_{\text{HW}} = \frac{\sum_{i=\tau+1}^{n}(y_i - \widehat{y}_i^{\text{HW}})^2}{n-\tau}.$$

The calculation is simplified by having stored the triples $(\widehat{y}_i^{\text{HW}}, \widehat{y}_i^{\text{LR}}, y_i)$ for $i \in \{\tau+1, \ldots, n\}$ in `sDict` using i as the key.

Initialize a list to store the error estimates at the top of the script, say `err =[0]*2`. We'll compute error estimates if $n > \tau$. This is accomplished by extracting the forecast and the actual value pairs from `sDict`. A list of the squared differences is constructed using list comprehension. The mean of the list is computed, and then the square root of the mean. The result is the estimated root mean square forecasting error.

```
if n > tau:
    for j in range(2):
        errList = [(sDict[i][2] - sDict[i][j])**2
              for i in sDict if len(sDict[i]) == 3]
        err[j] = np.sqrt(np.mean(errList))
```

The test `len(sDict[i]) == 3` insures that a forecast has been computed for the target price. The first τ prices have no associated forecasts and so the length of the associated dictionary entries is 1.

12. Write the error estimates to the console. Write the slope estimates obtained from the two forecasting methods.

```
print('Error = ', round(err[0], 5), round(err[1], 5))
print('LR slope = ', round(float(betaHat[1]), 5),
      ' HW slope = ', round(beta,5))
```

13. Upon completion of the `while` loop, extract and plot the three series and note the error estimates for the two forecasting functions.

```
import matplotlib.pyplot as plt
plt.plot(x,y)
plt.plot(x,lr)
plt.plot(x,hw)
```

12.3.1 Remarks

Using time step as the predictor variable produced a time-varying linear regression forecasting function very similar to the Holt-Winters forecasting function. Both functions forecast the future value $y_{n+\tau}$ using an estimate of $\mathrm{E}(Y_{n+\tau}|D_n) = \mu_{n+\tau}$, where $D_n = (y_1, \ldots, y_n)$ represents the data stream. Both models describe the future process mean as the sum of the current process mean and an increment determined by the current rate of change in $\mu_1, \ldots, \mu_n$. For instance, the Holt-Winters model is

$$\mu_{n+\tau} = \mu_n + \tau\beta_n.$$

where β_n is the current rate of change in the process mean. Both parameters μ_n and β_n are allowed to vary with time step. Both estimators are either weighted averages of the previous step estimates and the most recent observation or are computed from weighted averages of the previous step estimates and the most recent observation.[1]

There's not much more that can be done easily to forecast $Y_{n+\tau}$ without more data. One way to bring in more data in this situation is extend the linear regression predictor vector to include past prices of the target and other stocks as we did in Sect. 11.8. Extending the predictor vector is relatively easy. The primary hurdle is that $\mathbf{A}_n^*$ may be singular and so an alternative

[1] The time-varying linear regression estimator of $\boldsymbol{\beta}_n$ is computed from the statistics $\mathbf{A}_n^*$ and $\mathbf{z}_n^*$ which are weighted averages of the previous step estimates and the most recent observation.

forecasting algorithm should be available to compute the forecast in the case that $\mathbf{A}_n^*$ is singular. Another approach is the pattern matching approach of Sect. 9.10. The pattern matching approach requires more effort as a set of recent patterns needs to be updated at regular intervals.

We turn now to a substantially different source of streaming data, the Twitter stream.

12.4 The Twitter Streaming API

Twitter is an online social networking service that enjoys enormous popularity. As of the first quarter of 2016, more than 300 million users were broadcasting in excess of 500 million tweets daily [8]. Twitter users send and read text-based messages of up to 140 characters, referred to as *tweets*. Unregistered users can read tweets, while registered users can post tweets through the website interface and mobile devices. An analyst may tap the Twitter stream and receive some or all of the messages broadcast for public consumption and a considerable amount of associated information regarding the senders. The Twitter stream has been used for a variety of political and research purposes. For example, Twitter was used by government opponents to disseminate information during the Arab Spring rebellions in Egypt and Tunisia [36]. Perhaps as a consequence, it has been proposed that Twitter be used for monitoring group dynamics in real-time for purposes such as early detection of disease outbreaks [55]. Opportunities for exploiting the Twitter stream for financial gain surely exist but we have not yet determined what they are.

Tweets possess 30 or more attributes besides the text message sent by the user [52, 61]. The attributes, which may be blocked by the user, include user name, a short self-description, and geographic origin of the tweet (latitude and longitude and the country of origin). The description of the sender, when it exists, has been written by the user to describe herself, and so provides potentially useful information about the sender. Tweets are JSON-encoded, and consequently, the attributes are easily extracted. Two streams may be tapped through the Twitter API. The streams are the public stream, which requires a set of access credentials that are obtained by registering with Twitter Inc. through their API, and the Twitter Firehose [62] which requires special permissions. The public stream consists of a small sample of all tweets transmitted by the Twitter Firehose. It is however, adequate for learning how to process the Twitter stream and, in some cases, testing prototypes.

Given the massive volume of messages that are broadcast within the Twitter sphere, an individual that wishes to communicate with others regarding a specific topic faces the challenge of getting their messages to others interested in the topic. A popular solution is to include one or more identifying hashtags in the message. A hashtag is a word preceded by the hashtag symbol: #. Because Twitter users can search for and view tweets containing specific words or phrases, users that include a hashtag within their messages are providing an identifier so that others with similar interests can find their message. Consequently, hashtags lead to the formation of transitory groups of users united by a common topical interest. For the analyst, the senders of hashtag-labeled messages may be thought of as members of a group, and the division of a Twitter-sphere into groups of similar individuals is often advantageous in the study of the population. The possibility of collecting very large volumes of data on the thoughts of participants imply that Twitter is an attractive data source for the study of social networks and human behavior. For example, reaction to external stimuli such as political events can be measured by sentiment analysis of tweets. However, Twitter users differ from the general population to an unknown degree, and the analyst must recognize that the appropriate scope of inference is most likely limited to Twitter users.[2]

12.5 Tutorial: Tapping the Twitter Stream

In this tutorial, the reader will construct a dictionary of hashtags from a live Twitter stream and compute the relative frequency of occurrence for each observed hashtag. A dictionary with hashtags as keys and counts as values will be built so that the hashtags with the greatest amount of traffic can be determined.

There are two preliminary steps that must be carried out before the Twitter stream may be accessed using a `Python` script. The steps are to gain permission to tap the Twitter stream and to install a module that will interact with the Twitter API. Instructions for obtaining credentials are described in Appendix B.

The Twitter API consists of a set of programs and services that enables registered users to post and retrieve tweets and gain access to a live stream of tweets. A third-party `Python` module will create a communication link called a *socket* between a local host (computer) and the Twitter API. The socket allows the Twitter API to write to a reserved memory location called a *port* in the host. The `Python` program awaits new tweets, and whenever a tweet is received, it is scanned for hashtags. We will use the `TwitterAPI` module [20] to open the port and receive the stream.

1. Create a Twitter account if you do not already have one. Obtain the credentials necessary to unlock access to the API. Appendix B provides instructions.

[2] The scope of inference of a study is the target population to which the conclusions apply. A representative sample must be obtained from the target population for inferences to be statistically defensible. Individuals that send tweets are self-selected and it is hard to identify a larger population of which they are representative of.

2. Install `TwitterAPI`. Submit the instruction `pip install TwitterAPI` from within a Linux terminal or a Windows command prompt window.
3. Open a new `Python` script. The first action is to send a request to the Twitter API to stream tweets to the host computer. Pass your credentials, that is the four codes obtained from the Twitter API. The codes are named Consumer Key (`a1`), Consumer Secret (`a2`), Access Token (`a3`), and Access Token Secret (`a4`). Code the following instructions.

```
import sys
import time
from TwitterAPI import TwitterAPI

a1 = ’www’   # Replace ’xxx’ with your credentials.
a2 = ’xxx’
a3 = ’yyy’
a4 = ’zzz’
api = TwitterAPI(a1, a2, a3, a4)
```

4. Create an iterable object (it’s named `stream`) that will request tweets from the Twitter API. Pass a string of coordinates that defines a sampling region. The tweets that are received from the Twitter API originate from within the region.

```
coordinates = ’-125, 26, -68, 49’ # Roughly the continental U.S.
stream = api.request(’statuses/filter’, {’locations’:coordinates})
```

The instruction `api.request(...)` sends a request to the Twitter API to open the stream. If the request is accepted, then the `api` function will handle the stream sent by the Twitter API. As the stream is received, each tweet is written to a port.[3]

The list `coordinates` defines the source region from which tweets are streamed to the host port of your computer. The four values in the coordinate list define the southwestern and northeastern corners of a rectangle that roughly covers the continental United States. The southwestern corner is defined by longitude $= -125$ and latitude $= 26$.

5. Create a generator that will yield a series of items from the object `stream`.[4] The generator is a function that when called, generates a series of returns without terminating. A conventional function returns an object and then terminates. In a sense, the instruction `yield` replaces

[3] Port 80 is the standard port for non-secure web traffic.

[4] The object `stream` may itself be a generator. Creating the generator `tweetGenerator` may not be necessary.

the function instruction `return`. The generator does not terminate after executing the instruction `yield` but instead continues to return items. Since the object `stream` created by the `api.request` call is iterable, the generator will yield a stream of tweets when called in a `for` loop.

```
def generator(iterable):
    for item in iterable:
        yield item
tweetGenerator = generator(stream)
```

6. Iterations begin with the statement `for tweet in tweetGenerator`. A tweet is produced by the generator whenever a tweet is received from the Twitter API. The `tweet` produced by the `TwitterAPI` module is a dictionary containing a lot of information besides the message. The key `’text’` extracts the message.

```
stopAt = 5000
for tweet in tweetGenerator:
    try:
        txt =  tweet[’text’].lower()
        print(txt)
    except:
        print(time.ctime(),tweet)
    if n > StopAt:
        break
```

The `.lower()` attribute translates the characters to lowercase.

If tweets are received by the host computer more rapidly than they can be processed by the program, then `tweet` will consist only of a limit notice and the number of tweets that were lost. In this case, the `text` attribute will be missing from `tweet` and an exception will be raised. It is for this reason that an exception handler is invoked. The exception handler will trap other errors as well. If you have problems with the code and cannot see the error, then suppress the exception handler. You may replace the statement `try` with `if len(tweet) > 1` and `except` with `else` so that an error message is written to the console. If, at some point, the program was capturing tweets and then fails and reports a 401 error, then try restarting the kernel. If that fails, reboot the computer before restarting the program.

7. Temporarily replace the instruction `txt = tweet[’text’]` with `desc = tweet[’user’][’description’]`. The object `tweet[’user’]` is a dictionary. The contents of `desc` is a self-authored description of the user. Examining the keys of `tweet[’user’]` will reveal 38 attributes associated with the user, one of which is the self-authored description. The instruction `print(tweet[’user’].keys())` shows the keys.

Execute the script. After examining some of the user descriptions, change the script so that the messages are assigned to txt.

8. Import the re module at the top if the script. Immediately after extracting the message from tweet, extract the hashtags, if any, from txt.

```
hashtags = re.findall(r"#(\w+)", txt)
```

The instruction re.findall(r"#(\w+)", txt) uses the re module (regular expressions) to extract the strings in txt that follow a hashtag symbol (#). A list is returned. If no hashtags are found, then the returned value is an empty list.

9. We'll build a frequency table in the form of a dictionary in which the keys are hashtags and the values are the frequencies of occurrence. Use a counter dictionary to count the occurrences of hashtags. First, import the collections module. Initialize a dictionary named hashtagDict to store the hastags and frequencies of occurrence before consuming the Twitter stream.

```
import collections
hashtagDict = collections.Counter()
```

10. Iterate over the list of hashtags extracted from the message. Increment the count of incidences of each tag in hashtag using the update attribute of the counter dictionary.

```
for tag in hashtags:
    hashtagDict.update([tag])
print(n,len(hashtagDict), hashtagDict[tag])
```

If hashtags is empty, then the for loop will not execute.

11. The for tweet in tweetGenerator loop will terminate when n > StopAt. At that point in the code, create a list containing the sorted hashtag dictionary entries. Sort according to frequency of occurrence and extract the twenty-most common.

```
sortedList = sorted(hashtagDict.items(), key=operator.itemgetter(1))
shortList = sortedList[len(hashtagDict)-20:]
```

12. Construct a horizontal barchart showing the frequency of occurrence of the 20 most common hashtags. An example is shown in Fig. 12.2.

```
y = [n for _,n in shortList]
tags = [name for name, _ in shortList]
index = [i+1 for i in range(len(tags))]
plt.barh(index, y, align='center')
plt.yticks(index, tags)
```

You'll have to execute the instruction `import matplotlib.pyplot as plt` before attempting to use the plotting functions.

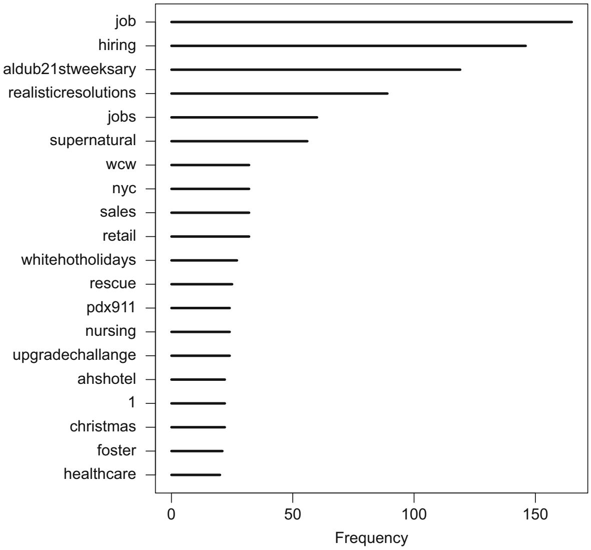


Fig. 12.2 Frequencies of the 20 most common hashtags collected from a stream of 50,000 tweets, December 9, 2015

12.5.1 Remarks

With a real-time data stream, we may begin to ask questions about the population from which the stream is originating. For instance, we may view Twitter users that use a particular hashtag, say, `#NeverTrump`, as a population of interest and attempt to quantify the response of the population to events of the day. Quantifying the response of individuals toward an event or topic can be approached by measuring the sentiment of tweets containing the hashtag. In this situation, we're interested in determining whether there is a prevalent attitude, positive or negative, toward the event or person. We won't pursue a deeper level of sophistication here, but it's also possible to measure the expression of several broad emotional states such as happiness, sadness, fear, disgust, and anger.[5] In any case, the stream of tweets containing a hashtag are a source of data from which information may be gleaned. The next topic is the measurement of sentiment contained in textual data.

12.6 Sentiment Analysis

We will broaden the discussion beyond streaming data and Twitter and discuss the process of extracting sentiment from textual data. The general objective of sentiment analysis is to extract information from a passage of text about the sentiment expressed within, usually positive or negative, but also with respect to emotional state. Sentiment analysis is used in various arenas, for instance, in marketing for assessing product perception and customer satisfaction and in politics for assisting in strategizing and resource allocation [34]. Presently, data sources are almost entirely textual. Sources include Twitter, blogs, social network sites, and discussion forums. The sentiment of topical subjects may change rapidly, particularly if important events are taking place at the time that the data is observed. Because the Twitter API generates a data stream in real-time, Twitter is a unique resource for near-real-time assessment of sentiment.

Sentiment analysis is complicated by the use slang, abbreviations, incorrect spellings, and emoticons. The problem of incorrect speech is most acute in the Twitter sphere. The 140 character limitation apparently encourages non-standard spellings and rampant use of abbreviations. Automatic spelling correction algorithms are of limited help since purposeful spelling deviations are common and difficult to correct (e.g., *gonna*). Another complication is that sentiment analysis is built upon assigning sentiment to individual words but often a phrase is used to express sentiment. Linguistic analyses of multi-word phrases is very difficult if only because of the vast number multi-word phrases and variations used in common speech. Negation is also a problem; for instance, a simplistic algorithm for measuring sentiment is likely to generate a positive sentiment for the phrase *I'm not at all happy* because the word happy is present. The negation is difficult to detect.

The simplest real task in sentiment analysis is determining the polarity of a text (or a block of text such as a sentence). Polarity is usually described as positive, negative, or neutral. A finer level of measurement attempts to assign a numerical score, say between -1 and 1, or perhaps assigns an emotive state such as *disgusted* or *happy* to the text, or a polarity score for one or more emotive states.

[5] Exercise 12.5 provides an opportunity to work with emotional states.

The sentiment analysis of a text or a corpus[6] begins with splitting the text into units such as tweets, sentences, or articles. The units are character strings which must be split again into words, or tokens, using a *tokenizer* that splits a character string whenever a white space is encountered. We refer to these sub-strings as *tokens* because they not necessarily words but generally represent something of meaning.

The next step assigns a sentiment score to each token contained in a textual unit. This operation requires a sentiment dictionary listing words and sentiment values. The values may consist of polarity (positive or negative) and perhaps the strength of the sentiment voiced by the word. For example, Table 12.1 shows four entries from a popular sentiment dictionary [67]. A pair is assigned to each word showing the strength (weak or strong) and the direction (positive or negative). Words encountered in the text that are not listed in the sentiment dictionary are assigned a neutral score or simply ignored because no information is available regarding sentiment. Words that occur with high frequency and are neutral in sentiment are sometimes called *stopwords*. Examples of stopwords are pronouns (e.g., *he*, *it*), prepositions (e.g., *on*, *at*), and conjunctions (e.g., *and*, *when*).

Table 12.1 A few dictionary entries showing polarity strength and direction. There are 6518 entries in the dictionary [31, 67]

Word	Strength	Direction
Abandoned	Weak	Negative
Abandonment	Weak	Negative
Abandon	Weak	Negative
Abase	Strong	Negative

The classification of words by strength and direction illustrated by Table 12.1 leads to four classes, or groups of sentiment values: (strong, positive), (strong, negative), (weak, positive), and (weak, negative). Polarity may be measured in other ways; in particular, numerical values can be assigned to each sentiment. Table 12.2 shows our codification of strength and direction of sentiment. Textual units can be assigned numerical scores based on the values associated with each token in the textual unit. The analyst may search for each token in the sentiment dictionary, determine a sentiment score of the token from the sentiment dictionary, and compute a total or mean sentiment value. In the following tutorial, the sentiment associated with a hashtag is measured by the mean sentiment of all words that co-occur with the hashtag and are entered in the sentiment dictionary.

[6] A *corpus* is a collection of writings; for example, a collection of tweets containing a particular hashtag.

Table 12.2 Numerical scores assigned to sentiment classes

	Direction	
Strength	Positive	Negative
Weak	.5	−.5
Strong	1	−1

12.7 Tutorial: Sentiment Analysis of Hashtag Groups

The objective of this tutorial is to compute the sentiment of a hashtag group. A hashtag group is a collection of hashtags related to a common subject. For example, the hashtags *jobs* and *hiring* belong to a group of hashtags that are used by prospective employers to advertise job openings. Our objective is to measure the sentiment expressed in messages containing a hashtag group representative. The sentiment of the hashtag group is estimated in real-time.

In this tutorial, we’ll extract messages from the Twitter stream that contain a hashtag of interest. A measure of the message sentiment is computed for each of these messages. Specifically, the sentiment score of each token in the message is determined from the sentiment dictionary. The message sentiment score is defined to be the total of the sentiment scores. The sentiment of a hashtag group is estimated by the mean sentiment score of all words co-occurring with a hashtag in the hashtag group.

The tutorial begins by building a sentiment dictionary and defining a hashtag group. Then, a port to the Twitter stream is opened and the stream is processed.

1. Build a list containing one or more popular hashtags. We’re using a group that’s related to jobs and hiring. We’ve chosen the hashtags by inspecting Fig. 12.2. Our hashtag group is the following set.

```
hashtagSet = set([’job’,’hiring’,’careerarc’,’jobs’,’hospitality’])
```

2. Retrieve the sentiment dictionary [67] from Tim Jurka’s github webpage https://github.com/timjurka. The file is named `subjectivity.csv` and can be accessed by navigating to the `sentiment` repository. A few dictionary entries are shown in Table 12.1.
3. Read the file `subjectivity.csv` and extract the word, strength, and polarity.

```
path = ’../Data/subjectivity.csv’
sentDict = {}
with open(path, mode= "r",encoding=’latin-1’) as f:
    for record in f:
        word, strength, polarity = record.replace(’\n’,’’).split(’,’)
```

We've initialized an empty dictionary named `sentDict` to store the sentiment words and sentiment values.

4. Compute a numerical score for each sentiment word in the file using the codification scheme shown in Table 12.2. Store the word as the key and the sentiment score as the value in `sentDict`.

```
score = 1.0
if polarity == "negative":
    score = -1.0
if strength == "weaksubj":
    score *= 0.5
sentDict[word] = score
```

5. Request access to the Twitter stream using the calls `TwitterAPI(...)` and `api.request(...)` described in instructions 3 and 4 of Sect. 12.5.
6. Include the definition of the `generator` function (Sect. 12.5, instruction 5) in your script.
7. Initialize a list for the computation of the mean sentiment of messages containing a hashtag in the hashtag group of interest. Create the generator.

```
meanSent = [0]*2
tweetGenerator = generator(stream)
```

8. Initialize a counter by setting `counter = 0`.
9. Build a `for` loop using the `tweetGenerator` object. If possible, extract the hashtags from each tweet and increment the number of times that a hashtag belonging to the hashtag group was encountered in the Twitter stream.

```
for n, tweet in enumerate(tweetGenerator):
    try:
        txt =  tweet[’text’].lower()
        hashtags = set(re.findall(r"#(\w+)", txt)) & hashtagSet
    except:
        print(time.ctime(),tweet)
```

The command `set(re.findall(r"#(\w+)", txt)) & hashtagSet` computes the intersection of `hashtagSet` and the set of hashtags appearing `txt`.

10. If there are hashtags of interest in the message, then compute the sentiment of the message. To accomplish this, iterate over the tokens contained in the message and look up each token in the sentiment dictionary. Increment the sum of the sentiment values and the count of messages that contain a hashtag from the hashtag group.

```
    if len(hashtags) > 0:
        sent = 0
        for token in txt.split(' '):
            try:
                sent += sentDict[token]
            except(KeyError):
                pass
        meanSent[0] += 1
        meanSent[1] += sent
        print(n,meanSent[0],round(meanSent[1]/meanSent[0],3))
```

The test `len(hashtags) > 0` determines whether there are hashtags of interest in the message.

The code segment immediately follows the statement `hashtags =` $\cdots$.

11. Collect 1000 messages that contain hashtags from the hashtag group and terminate the program.
12. Create a different hashtag group and run the script again. The website https://www.hashtags.org/trending-on-twitter.html provides a list of popular hashtags. It appears that hashtags with commercial purposes have been omitted from their lists.

12.8 Exercises

Problem 12.1. Return to instruction 13 of the Sect. 12.3 tutorial and plot the observations as points and the forecasts as lines using `ggplot2`. Use different colors for the two types of forecasts.

Problem 12.2. Emoticon are often used to express emotion in the text-based social media. The set $A = \{$`:-)`, `:)`, `:D`, `:o)`, `:]`, `:3`, `:>`, `:}`, `:^`$\}$ contains some commonly-used emoticons used to express happiness or amusement, and the emoticons in the set $B = \{$`:-(`, `:(`, `:-c`, `:C`, `:-{`, `:-[`, `:[`, `:{`, `:-{`$\}$ are used to convey unhappiness.

a. Compute the mean sentiment of tweets that contain emoticons from set A and again for set B.
b. Compute the mean sentiment of all tweets.
c. Compute the standard error of the mean $\widehat{\sigma}(\overline{y}) = \widehat{\sigma}/\sqrt{n}$ for the three sets A, B, and all tweets.

Problem 12.3. Returning to the Tutorial of Sect. 12.3, investigate relationship between forecasting error, τ, and the tuning constants α_r, α_e, and α_b. Calculate estimates of forecasting error for the Holt-Winters forecasting function and linear regression with time-varying coefficients forecasting function using $\tau \in \{10, 50, 100\}$ and $\alpha \in \{.05, .1, .2\}$. Set $\alpha_r = \alpha_e = \alpha_b = \alpha$.

Problem 12.4. In the Sect. 12.5 tutorial, you were asked to construct a barchart using `pyplot` (instruction 12). Build the barchart using `ggplot`.

Problem 12.5. Investigate sentiment associated with the emotions anger, disgust, fear, joy, sadness, and surprise. A list of words and their classification as expressions of one of the five emotions can be obtained from Jurka's github webpage https://github.com/timjurka. Use the data file containing the words and emotions, `emotions.csv`, to compute the mean sentiment of messages containing a token expressing a particular emotion. Summarize your results in a table.

Appendix A
Solutions to Exercises

Chapter 2

2.1. Show

$$n^2 = \sum_{i=1}^{n} \sum_{j=1}^{i-1} 1 + \sum_{i=1}^{n} \sum_{j=i+1}^{n} 1 + n,$$

and solve for $\sum_{i=1}^{n} \sum_{j=i+1}^{n} 1$.

2.2. Formulas for combinations.

a.

$$\begin{aligned} C_{n,3} &= \frac{n(n-1)(n-2)}{3 \cdot 2 \cdot 1} = \frac{n!}{3!(n-3)!}, \\ C_{5,3} &= 10, \\ C_{100,3} &= 161,700. \end{aligned}$$

b. Python code to generate and print the three-tuples:

```
i = j = k = 0
threeTuples = [(i, j, k)]
for i in range(3):
    for j in range(i+1, 4):
        for k in range(j+1, 5):
            print(i, j, k)
```

2.4.

```
import operator
l = [(1,2,3),(4,1,5),(0,0,6)]
#sort by second coordinate
sortedList = sorted(l, key = operator.itemgetter(1))
```

B. Steele et al., *Algorithms for Data Science*,
DOI 10.1007/978-3-319-45797-0

2.6.

```
import timeit
lcStart = timeit.default_timer()
lcSetpairs ={(i,j) for i in range(n) for j in range(i+1,n+1)}
lcTime=timeit.default_timer() - lcStart

joinStart = timeit.default_timer()
setJoin=set()
for i in range(n):
    for j in range(i+1,n+1):
        pairSet =set({(i,j)})
        setJoin=setJoin.union(pairSet)
        joinTime =timeit.default_timer() - joinStart
```

Chapter 3

3.1. Suppose that for some $n \in \{1, 2, \ldots, \}$,

$$s(\cup_{i=1}^{n}) = \sum_{i=1}^{n} s(D_i).$$

Then,

$$\begin{aligned} s(\cup_{i=1}^{n+1} D_i) &= s\left(D_{n+1} \cup [\cup_{i=1}^{n} D_i]\right) \\ &= s\left(D_{n+1}\right) + s\left([\cup_{i=1}^{n} D_i]\right). \end{aligned}$$

Hence, $s(\cup_{i=1}^{n}) = \sum_{i=1}^{n} s(D_i) \Rightarrow s(\cup_{i=1}^{n+1}) = \sum_{i=1}^{n+1} s(D_i)$. Therefore, the statement is true for every integer $n > 0$.

3.3. Let $\mathbf{y} = [y_1 \quad \cdots \quad y_n]^T$. Then,

$$\begin{aligned} s(\boldsymbol{\beta}) &= (\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^T(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}) \\ \Rightarrow \frac{s(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} &= -2\mathbf{X}^T(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}). \end{aligned}$$

Set the vector of partial derivatives equal to $\mathbf{0}$ and solve for $\boldsymbol{\beta}$.

3.5. Let $\underset{n\times p}{\mathbf{X}} = \begin{bmatrix} \underset{n\ \times 1}{\mathbf{X}_1} & \cdots & \underset{n\times 1}{\mathbf{X}_p} \end{bmatrix}$.

a. Since $\mathbf{j}^T = [1 \quad \cdots \quad 1]$, $\mathbf{j}^T\mathbf{x} = \sum x_i$.
b. $(\mathbf{j}^T\mathbf{j})^{-1}\mathbf{j}^T\mathbf{X} = n^{-1}\left[\mathbf{j}^T\mathbf{X}_1 \quad \cdots \quad \mathbf{j}^T\mathbf{X}_p\right]$.
c. `j.T.dot(X)/j.T.dot(j)`

Chapter 4

4.1. Let E denote the event of a cluster failure. $\Pr(E) = 1 - \Pr(E_1^c) \times \cdots \times \Pr(E_n^c)$. If $p = .001$ and $n = 1000$, then $\Pr(E) = .6323$.

4.4. The key elements of the mapper program are shown below.

```
for record in sys.stdin:
    variables = record.split(’\t’)
    try:
        allowed = round(float(variables[22]), 2)
        print(str(n%100) + ’\t’ + provider + ’|’ +
            str(payment)+ ’|’ + str(submitted)+ ’|’ + str(allowed))
    except(ValueError):
        pass
```

The key elements of the reducer program are shown below.

```
A = np.zeros(shape= (q, q))
w = np.matrix([1,0,0,0]).T
for record in sys.stdin:
    _,data = record.replace(’\n’,’’).split(’\t’)
    numerics = data.split(’|’)
    for i,x in enumerate(numerics[1:]):
        w[i+1,0] = float(x)
    A += w*w.T
for row in A:
    print(’,’.join([str(r) for r in row]))
```

After computing $\mathbf{A} = \sum_i^r \mathbf{A}_i$, the correlation matrix is computed using the algorithm of Chap. 3 and formula (3.21).

```
n = A[0,0]
mean = np.matrix(A[1:,0]/n).T
CenMoment  = A[1:,1:]/n - mean.T*mean
s = np.sqrt(np.diag(CenMoment))
D = np.diag(1/s)
corMatrix = D*CenMoment*D
```

Chapter 5

5.1. See Fig. A.1.
5.4. See Fig. A.2.
5.4. See Fig. A.3.

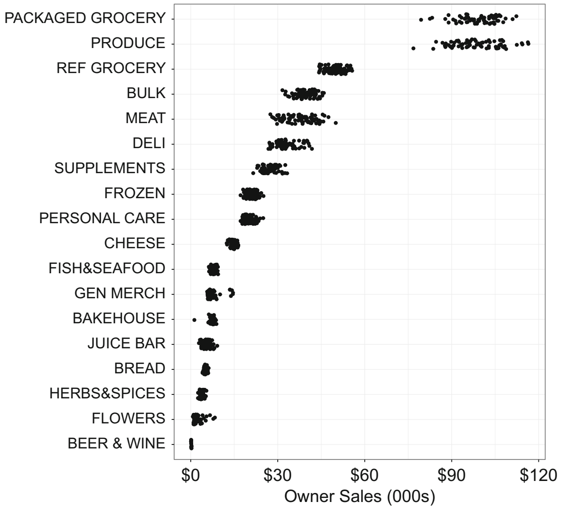


Fig. A.1 Dotchart of monthly sales by department

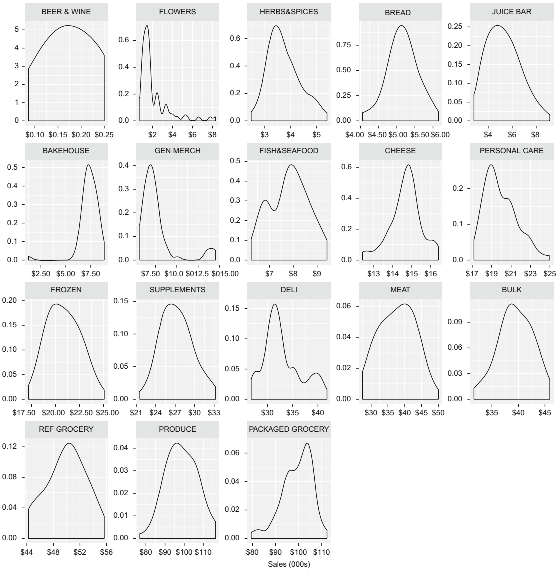


Fig. A.2 A faceted graphic showing the empirical density of sales per month by department

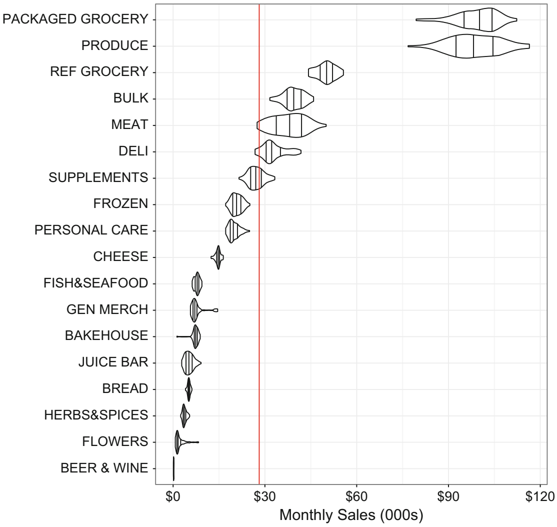


Fig. A.3 Monthly sales by department

Chapter 6

6.1. $$\frac{\sum y_i^2/n - \sum(y_i - \widehat{y}_i)^2/(n-p)}{\sum y_i^2/n} = \frac{47007.17 - 10746.03}{47007.17} = R^2.$$

6.3. The model without x_{female} can be expressed conditionally as

$$\begin{aligned} \mathrm{E}(Y_i|\mathbf{x}_i) &= \beta_0 + \beta_1 x_{\text{ssf},i} + \beta_3 x_{\text{interaction},i} \\ &= \begin{cases} \beta_0 + \beta_1 x_{\text{ssf},i}, & \text{if } x_{\text{female}} = 0, \\ \beta_0 + (\beta_1 + \beta_3) x_{\text{ssf},i} & \text{if } x_{\text{female}} = 1. \end{cases} \end{aligned}$$

6.5.

a. Figure A.4 shows the fitted models and data.
b. Table A.1 shows the fitted models. The slopes are not much different in a practical sense.
c. Table A.2 shows confidence intervals. The confidence intervals overlap to a substantial extent and so the data do not support the contention that the true slopes are different.

6.7. Anorexia data set.

a. $n_{\text{CBT}} = 29$, $n_{\text{Cont}} = 26$, $n_{\text{FT}} = 17$.

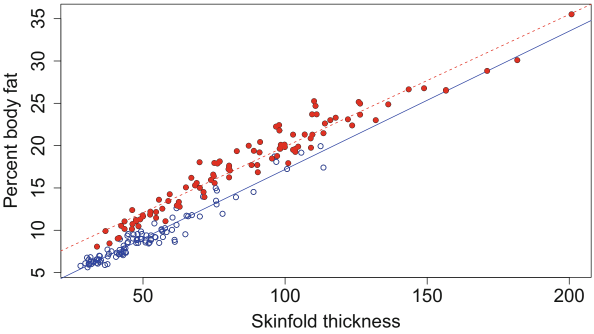


Fig. A.4 Percent body fat plotted against skinfold thickness for 202 Australian athletes. Separate regression lines are shown for each gender. Males are shown as *open circles* and females are shown as *filled circles*

Table A.1 Fitted models for males and females

Group	$\widehat{\beta}_0$	$\widehat{\beta}_1$	$\widehat{\sigma}(\beta_1)$
Females	4.26	.156	.0040
Males	.849	.163	.0042

Table A.2 Confidence intervals for β_1 for males and females. Separate regression lines

	95% confidence interval bounds	
Group	Lower	Upper
Females	.148	.164
Males	.155	.172

b. See Fig. A.5.
c. See Fig. A.6.
d. See Fig. A.6. The black line consists of points with equal pre- and post-experiment weights. Because the fitted regression lines for the two treatments are positioned above the diagonal line, most of these patients have gained weight. For the control, there's no apparent relationship between pre- and post-experiment weight. There's no explanation in the data for why there's no relationship.
e. See Fig. A.7.
f. There's evidence of a relationship for the two treatments (p-value = .026 for the family therapy treatment and p-value = .007 for the cognitive behavioral therapy treatment). The p-values are obtained from a two-sided test of the alternative hypothesis $H_a : \beta_1 \neq 0$. There's insufficient evidence to conclude that a relationship exists for the control group.
g. Table A.3 shows the estimates and 95% confidence intervals.

   The intercepts are the estimated mean post-treatment weight of a patient given that their pre-treatment weight is equal to the mean pre-treatment weight of all patients. By centering, we have adjusted for, and removed

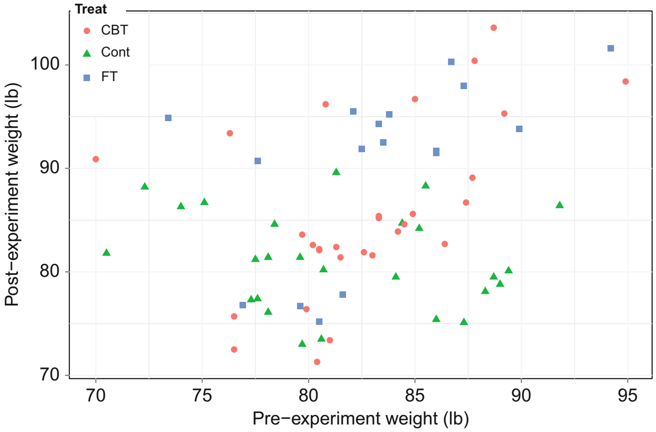


Fig. A.5 Pre- and post-experiment weights for $n = 72$ anorexia patients. Points are identified by treatment group

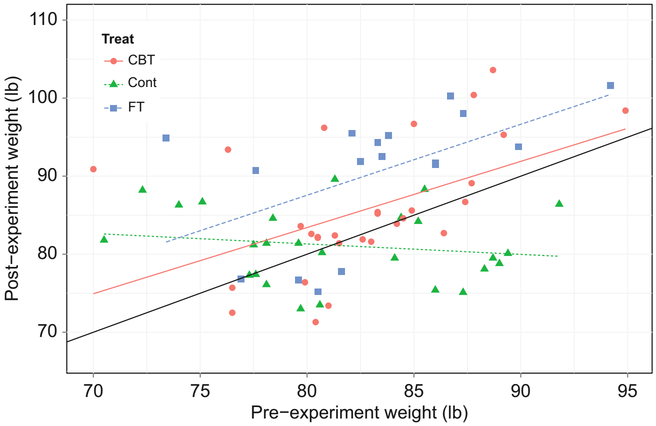


Fig. A.6 Pre- and post-experiment weights for $n = 72$ anorexia patients. Separate regression lines are shown for each treatment group

Table A.3 Confidence intervals for the centered intercepts. The centered intercept is the estimated mean post-treatment weight after adjusting for differences in pre-treatment weight. Separate regression lines were fit for each treatment group

		95% confidence interval bounds	
Treatment	Estimate	Lower	Upper
Control	80.99	79.04	82.95
Cognitive behavioral therapy	85.46	82.63	88.28
Family therapy	89.75	85.88	93.16

the effects of differences among groups with respect to the pre-experiment weights. The estimates in Table A.3 can be compared to assess the efficacy of the two treatments.

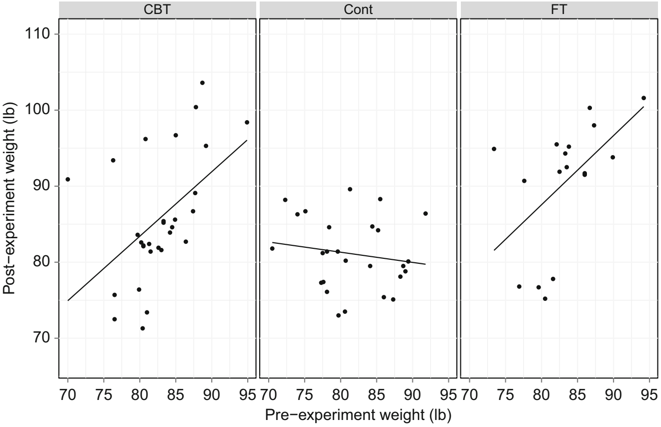


Fig. A.7 Pre- and post-experiment weights for $n = 72$ anorexia patients. Data are plotted by treatment group along with a regression line

h. There appears to be a significant difference between control and the cognitive behavioral therapy means (the $\mathrm{E}(Y|\mathbf{x})$'s) since the confidence intervals do not overlap. The estimated difference is 4.47 lbs for patients that have a pre-treatment mean weight of $\overline{y}_{\text{pre-treatment}} = 82.40$ lbs. The same statement holds for the family therapy versus control comparison. Moreover, the cognitive behavioral therapy and family therapy confidence intervals do not overlap, and so we conclude that the family therapy treatment mean is larger than the cognitive behavioral therapy mean. Both therapies are concluded to be effective and promoting weight gain, and family therapy is better than cognitive behavioral therapy. Our conclusion are based on the confidence interval principle: if a value is not contained in an interval, then the conjecture that the value is the group mean is inconsistent with the data. The approach of comparing confidence intervals may not always yield a firm conclusion. The analysis of variance test does yield and indisputable conclusion.

i. $f = 5.41$, $\mathrm{df}_{\mathrm{num}} = 2$, $\mathrm{df}_{\mathrm{den}} = 66$, p-value $= .0067$. There's strong evidence that treatment affects the expected post-experiment weight and that the effect of the treatment depends on the pre-treatment weight.[1] The statement that the effect depends on the pre-treatment weight is unremarkable since it's obvious that the post-treatment weight strongly depends on the patient's weight at the beginning of the treatment.

[1] This statement is supported by the outcome of the analysis of variance test for interaction.

Chapter 7

7.1.

a. Setting $x_0 = \overline{x}$ implies that the second term in the sum is zero.
b. Suppose that the observations are equidistant and ordered as $x_1 < \cdots < x_n$. The midpoint is the point equi-distant from x_1 and x_n. Suppose that n is even and that m is the midpoint. Then, $x_n - m = m - x_1$, $x_{n-1} - m = m - x_2$, and so on. Therefore, $\sum_{i=1}^{n/2} m - x_i = \sum_{i=n/2+1}^{n} x_i - m \Rightarrow m = \overline{x}$. Therefore, the midpoint is the mean and $m = x_0$ minimizes the variance of $\text{var}[\widehat{\mu}(x_0)]$.

7.3

a. $\mu = N^{-1} \sum_{i=1}^{N} y_i$.
b. $\pi_j = nN^{-1}$.
c. Let Z_j be an indicator of the event that y_j is in the sample. Then, $\overline{Y} = n^{-1} \sum_{j=1}^{N} Z_j y_j = n^{-1} \sum_{i=1}^{n} y_i \Rightarrow \text{E}(\overline{Y}) = n^{-1} \sum_{j=1}^{N} nN^{-1} y_j = \mu$.
d. Use $\text{E}\left(\overline{Y}_w\right) = \text{E}\left(N^{-1} \sum_{j=1}^{N} w_j Z_j y_j\right)$.

7.5 Use `zip(x, x0)` and list comprehension. The question of which code is best, the function with three lines of code, or the single-line code, is difficult to answer as we must weigh the compactness of the one-liner against the clarity of the three-liner function.

7.7 Using exercise in place of education reduces the predictive accuracy of the rule. If we compare sensitivity and specificity for $p = .1$, we see that the new rule has a lower level of sensitivity (.780 versus .873). Specificity is larger, (.661 versus .548), but sensitivity is more important (Table A.4).

Table A.4 Values of sensitivity and specificity for five choices of the threshold p

p	Specificity	Sensitivity
.2	.879	.458
.19	.867	.483
.18	.851	.516
.17	.833	.55
⋮	⋮	⋮
.1	.661	.780

Chapter 8

8.1. Expand $\sum(a_i - b_i)^2$ and use the fact that $0 \le a_i \le 1 \Rightarrow a_i^2 \le a_i$.

Chapter 9

9.1. Use the fact that $\sum_{i=0}^{\infty} \alpha^i = (1-\alpha)^{-1}$, for $0 < \alpha < 1$.
9.3. Table A.5 shows the possible values of $\widehat{\Pr}(y_0 = l|\mathbf{x}_0)$ and of $\max \widehat{\Pr}(y_0 = l|\mathbf{x}_0)$.

Table A.5 Estimated probabilities of group membership from the conventional k-nearest-neighbor prediction function

	Possible values	
k	$\widehat{\Pr}(y_0 = l\|\mathbf{x}_0)$	$\max \widehat{\Pr}(y_0 = l\|\mathbf{x}_0)$
1	$0, 1$	1
3	$0, 1/3, 2/3, 1$	$2/3, 1$
5	$0, 1/5, 2/5, 3/5, 4/5, 1$	$3/5, 4/5, 1$

9.4.

```
i = np.argmax(counts)
if sum([counts[i] ==count for count in counts ]) > 1:
    counts[nhbr]+=1
```

9.5 The root mean square errors are 16.9 and 25.6.
9.7.

a. Table A.6 shows some of the estimates.

Table A.6 Estimates of root mean square prediction error $\widehat{\sigma}_{\text{kNN}}$ as a function of d and α

		d	
α	1	3	5
.05	15.7	24.5	30.7
.1	15.7		
.3	16.1	25.6	

b. For $p = 10$, $\alpha = .1$, and $d = 1$, $\widehat{\sigma}_{\text{kNN}} = 16.9$ and for $p = 10$, $\alpha = .05$ and $d = 4$, $\widehat{\sigma}_{\text{kNN}} = 30.5$. These estimates differ only slightly from the results using $p = 5$. The pattern length doesn't appear to have much effect. However, as might be expected, the analysis indicates that shorter lengths are better when the forecasted number of days ahead is small versus large.

Chapter 10

10.1 Maximize the log-probability given the constraint $\sum_{i=1}^{n} \pi_k = 1$ by introducing a Lagrange multiplier. The objective function is

$$f(\pi, \lambda) = \sum_{i=1}^{n} x_i \log \pi_i + \lambda(1 - \sum_{i=1}^{n} \pi_i).$$

Setting $\partial f / \partial \pi_i = 0$ implies $x_i = \lambda \pi_i \Rightarrow \lambda = \sum x_i$.

Chapter 11

11.1. Expand $\widehat{\mu}_n = \sum_t^n w_t y_t$:

$$\begin{aligned} \widehat{\mu}_n &= \alpha y_n + \textstyle\sum_{t=0}^{n-1} w_t y_t \\ &= \alpha y_n + \alpha(1-\alpha) y_{n-1} + \alpha(1-\alpha)^2 y_{n-2} + \cdots \\ &= \alpha y_n + (1-\alpha)[\alpha y_{n-1} + \alpha(1-\alpha) y_{n-2} + \cdots]. \end{aligned}$$

11.3 Let $\mathbf{W}$ be a diagonal matrix with the weights $w_0, \ldots, w_{n-1}$ arranged on the diagonal such that

$$\begin{aligned} S(\boldsymbol{\beta}_n) &= \textstyle\sum_{t=1}^{n} w_{n-t} \left(y_{t+\tau} - \mathbf{x}_t^T \boldsymbol{\beta}_n\right)^2 \\ &= (\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^T \mathbf{W} (\mathbf{y} - \mathbf{X}\boldsymbol{\beta}). \end{aligned} \tag{A.1}$$

Differentiating $S(\boldsymbol{\beta}_n)$ with respect to $\boldsymbol{\beta}$ and setting the vector of partial derivatives equal to zero leads to the normal equations

$$\mathbf{X}^T \mathbf{W} \mathbf{y} = \mathbf{X}^T \mathbf{W} \mathbf{X} \boldsymbol{\beta}. \tag{A.2}$$

Solve for $\boldsymbol{\beta}$ and argue that $\mathbf{X}^T \mathbf{W} \mathbf{X} = \sum w_i \mathbf{x}\mathbf{x}^T$ and $\mathbf{X}^T \mathbf{W} \mathbf{y} = \sum w_i \mathbf{x}_i y_i$.

11.5. Table A.7 shows estimates of $\widehat{\sigma}^2_{\text{reg}}$ for some choices of α.

Table A.7 Estimates of σ^2_{reg} for three choices of α and predictor variable

		α	
Predictor	.02	.05	.1
n	.639	.624	.999
AAPL	.652	.623	.622
GOOG	.651	.629	.949

Chapter 12

12.2. The code for computing the sentiment of messages containing happy emoticons follows.

```
happySent = [0]*2
for n,tweet in enumerate(tweetGenerator):
    try:
        txt =  tweet[’text’].lower()
        tokens = set(txt.split(’ ’))

        if len(tokens & A) > 0:
            sent = 0
            for token in tokens:
                try:
                    sent += sentDict[token]
                except(KeyError):
                    pass
            happySent[0] += 1
            happySent[1] += sent
            happySent[2] += sent**2
            se  = np.sqrt((happySent[2]/happySent[0]
                 - (happySent[1]/happySent[0])**2)/happySent[0])
            print(’Happy ’,n,happySent[0],
                 round(happySent[1]/happySent[0],3),round(se,3))
    except:
        print(time.ctime(),tweet)
    if 1000 < happySent[0]:
        sys.exit()
```

12.5. The tutorial of Sect. 12.7 can be used with a few modifications.

1. Read the data file `emotions.csv` and build a dictionary that uses token as a key and the emotion as the value. It’s named `emotionDict`. Build a dictionary `emotionSentDict` that uses emotion as a key and two-element lists as values. The lists will contain the number of occurrences of a particular emotion and the associated sum of sentiment values. Build a set containing the five emotions.

```
    emotionDict = {}
    path = ’../Data/emotions.csv’
    with open(path,mode= "r",encoding=’utf-8’) as f:
        for record in f:
            token, emotion = record.strip(’\n’).split(’,’)
            emotionDict[token] = emotion
    keys = emotionDict.keys()
    emotionSentDict = dict.fromkeys(emotionDict.values(),[0]*2)
    emotions = set(emotionDict.values())
```

2. After extracting the tokens from the message (`txt`), increment the entries in `emotionSentDict[emotion]` for those emotions appearing in the message.

```
emotionTokens  = tokens & keys
if len(emotionTokens) > 0:
    sent = 0
    for token in tokens:
        try:
            sent += sentDict[token]
        except(KeyError):
            pass
        for token in emotionTokens:
            emotion = emotionDict[token]
            n,Sum = emotionSentDict[emotion]
            n += 1
            Sum += sent
            emotionSentDict[emotion] = [n,Sum]
```

Our results are shown in Table A.8.

Table A.8 Mean sentiment of tweets containing a particular emotion

Emotion	Mean
Joy	1.115
Surprise	.312
Anger	−.005
Sadness	−.471
Disgust	−.553
Fear	−.593

Appendix B
Accessing the Twitter API

There are two preliminary steps to tapping the Twitter API: getting a Twitter account and getting credentials. The credentials unlock access to the API and you need an account to get credentials.[1]

1. Get a Twitter account if you do not have one. Navigate to the URL https://twitter.com/signup. Open an account and make note of your user name and password.
2. Navigate to https://apps.twitter.com/ and login with your user name and password.
3. Navigate to the URL https://apps.twitter.com/app/new and fill in the required details. In essence, you're informing Twitter that you're going to create an account and agree to their conditions in exchange for access to the Twitter stream. You will not actually create an app in the course of the tutorials, but registering an app releases the credentials that are necessary for accessing the Twitter stream.

   a. For the website entry, enter the URL of a website that is accessible by the public. We will not attempt to access the website.
   b. You may leave the *Callback URL* blank.
   c. Agree to the Twitter Development Agreement and click on *Create your Twitter application.*

4. The next page will contain a tab to *Keys and Access Tokens*. Click on the tab. There's a button that will create the access tokens at the bottom of the page that opens. Click on the button.
5. The next page that opens shows the credentials. Copy the Consumer Key (`a1`), Consumer Secret (`a2`), Access Token (`a3`), and Access Token Secret (`a4`) and paste the keys into your Python script. Set the assignments, say,

[1] The webpages used for obtaining the credentials have changed in the past few years but the process has not. It's of course possible that the webpages will change again in the future.

B. Steele et al., *Algorithms for Data Science*,
DOI 10.1007/978-3-319-45797-0

a1 = 'xxx' in the Python script. Instruction 3 of the Sect. 12.5 tutorial provides details.

6. Return to the Twitter app development page and click on the *Test OAuth* button.
7. Check that the keys copied into the Python script match the OAuth settings. At the bottom of the OAuth Tool page is a button titled *Get OAuth Signature*. Click on that button.
8. Close the page. You're authorized to receive tweets through the Twitter API.
9. In your Python script for accessing the Twitter stream, there is a function call that requests access to the Twitter API. The function call appears so:

```
api = TwitterAPI(a1,a2,a3,a4)
```

The keys a1, a2, a3, a4 are identified in instruction 5.

References

1. D. Adair, The authorship of the disputed federalist papers. William Mary Q. **1**(2), 97–122 (1944)
2. C.C. Aggarwal, *Data Mining - The Textbook* (Springer, New York, 2015)
3. J. Albert, M. Rizzo, *R by Example* (Springer, New York, 2012)
4. American Diabetes Association, http://www.diabetes.org/diabetes-basics/statistics/. Accessed 15 June 2016
5. K. Bache, M. Lichman, *University of California Irvine Machine Learning Repository* (University of California, Irvine, 2013). http://archive.ics.uci.edu/ml
6. S. Bird, E. Klein, E. Loper, *Natural Language Processing with Python* (O'Reilly Media, Sebastopol, 2009)
7. C.A. Brewer, G.W. Hatchard, M.A. Harrower, Colorbrewer in print: a catalog of color schemes for maps. Cartogr. Geogr. Inf. Sci. **30**(1), 5–32 (2003)
8. British Broadcasting Service, Twitter revamps 140-character tweet length rules. http://www.bbc.com/news/technology-36367752. Accessed 14 June 2016
9. N.A. Campbell, R.J. Mahon, A multivariate study of variation in two species of rock crab of genus leptograpsus. Aust. J. Zool. **22**, 417–425 (1974)
10. Centers for Disease Control and Prevention, Behavioral Risk Factor Surveillance System Weighting BRFSS Data (2013). http://www.cdc.gov/brfss/annual_data/2013/pdf/Weighting_Data.pdf
11. Centers for Disease Control and Prevention, The BRFSS Data User Guide (2013). http://www.cdc.gov/brfss/data_documentation/pdf/userguidejune2013.pdf
12. Centers for Medicare & Medicaid Services, NHE Fact Sheet (2015). https://www.cms.gov/research-statistics-data-and-systems/statistics-trends-and-reports/nationalhealthexpenddata/nhe-fact-sheet.html
13. W.S. Cleveland, S.J. Devlin, Locally-weighted regression: an approach to regression analysis by local fitting. J. Am. Stat. Assoc. **83**(403), 596–610 (1988)
14. J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, in *Proceedings of the Sixth Symposium on Operating System Design and Implementation* (2004), pp. 107–113
15. J. Dean, S. Ghemawat, MapReduce: a flexible data processing tool. Commun. Assoc. Comput. Mach. **53**(1), 72–77 (2010)
16. R. Ecob, G.D. Smith, Income and health: what is the nature of the relationship? Soc. Sci. Med. **48**, 693–705 (1999)
17. S.L. Ettner, New evidence on the relationship between income and health. J. Health Econ. **15**(1), 67–85 (1996)

B. Steele et al., *Algorithms for Data Science*,
DOI 10.1007/978-3-319-45797-0

18. H. Fanaee-T, J. Gama, Event labeling combining ensemble detectors and background knowledge, in *Progress in Artificial Intelligence* (Springer, Berlin, 2013), pp. 1–15
19. J. Fox, S. Weisberg, *An R Companion to Applied Regression*, 2nd edn. (Sage, Thousand Oaks, 2011)
20. J. Geduldig, `https://github.com/geduldig/TwitterAPI`. Accessed 15 June 2016
21. D. Gill, C. Lipsmeyer, Soft money and hard choices: why political parties might legislate against soft money donations. Public Choice **123**(3–4), 411–438 (2005)
22. C. Goldberg, `https://github.com/cgoldberg`. Accessed 14 June 2016
23. J. Grus, *Data Science from Scratch* (O'Reilly Media, Sebastopol, 2015)
24. D.J. Hand, F. Daly, K. McConway, D. Lunn, E. Ostrowski, *A Handbook of Small Data Sets* (Chapman & Hall, London, 1993)
25. F. Harrell, *Regression Modeling Strategies* (Springer, New York/Secaucus, 2006)
26. Harvard T.H. Chan School for Public Health, Obesity prevention source (2015). http://www.hsph.harvard.edu/obesity-prevention-source/us-obesity-trends-map/
27. A.C. Harvey, *Forecasting, Structural Time Series and the Kalman Filter* (Cambridge University Press, Cambridge, 1989)
28. T. Hastie, R. Tibshirani, J. Friedman, *The Elements of Statistical Learning*, 2nd edn. (Springer, New York, 2009)
29. G. James, D. Witten, T. Hastie, R. Tibshirani, *An Introduction to Statistical Learning with Applications in R* (Springer, New York, 2013)
30. J. Janssens, *Data Science at the Command Line* (O'Reilly Media, Sebastopol, 2014)
31. T. Jurka, `https://github.com/timjurka`. Accessed 15 June 2016
32. Kaggle, `https://www.kaggle.com/competitions`. Accessed 12 June 2016
33. E.L. Korn, B.I. Graubard, Examples of differing weighted and unweighted estimates from a sample survey. Am. Stat. **49**(3), 291–295 (1995)
34. E. Kouloumpis, T. Wilson, J. Moore, Twitter sentiment analysis: the good the bad and the OMG! in *Proceedings of the Fifth International Association for the Advancement of Artificial Intelligence (AAAI) Conference on Weblogs and Social Media* (2011)
35. J.A. Levine, Poverty and obesity in the U.S. Diabetes **60**(11), 2667–2668 (2011)
36. G. Lotan, E. Graeff, M. Ananny, D. Gaffney, I. Pearce, D. Boyd, The Arab Spring: the revolutions were tweeted: Information flows during the 2011 Tunisian and Egyptian revolutions. Int. J. Commun. **5**, 31 (2011)
37. B. Lublinsky, K.T. Smith, A. Yakubovich, *Hadoop Solutions* (Wiley, Indianapolis, 2013)
38. J. Maindonald, J. Braun, *Data Analysis and Graphics Using R*, 3rd edn. (Cambridge University Press, Cambridge, 2010)
39. G. McLachlan, T. Krishnan, *The EM Algorithm and Extensions*, 2nd edn. (Wiley, Hoboken, 2008)
40. A.H. Mokdad, M.K. Serdula, W.H. Dietz, B.A. Bowman, J.S. Marks, J.P. Koplan, The spread of the obesity epidemic in the United States, 1991–1998. J. Am. Med. Assoc. **282**(16), 1519–1522 (1999)
41. F. Mosteller, D.L. Wallace, Inference in an authorship problem. J. Am. Stat. Assoc. **58**(302), 275–309 (1963)
42. E. O'Mahony, D.B. Shmoys, Data analysis and optimization for (citi) bike sharing, in *Proceedings of the Twenty-Ninth Association for the Advancement of Artificial Intelligence (AAAI) Conference on Artificial Intelligence* (2015)
43. A.M. Prentice, The emerging epidemic of obesity in developing countries. Int. J. Epidemiol. **35**(1), 93–99 (2006)
44. Project Gutenberg, `https://www.gutenberg.org/`. Accessed 7 June 2016
45. F. Provost, T. Fawcett, *Data Science for Business* (O'Reilly Media, Sebastopol, 2013)
46. R Core Team, *R: A Language and Environment for Statistical Computing* (R Foundation for Statistical Computing, Vienna, 2014)

47. L. Ramalho, *Fluent Python* (O'Reilly Media, Sebastopol, 2015)
48. F. Ramsey, D. Schafer, *The Statistical Sleuth*, 3rd edn. (Brooks/Cole, Boston, 2012)
49. J.J. Reilly, J. Wilson, J.V. Durnin, Determination of body composition from skinfold thickness: a validation study. Arch. Dis. Child. **73**(4), 305–310 (1995)
50. A.C. Rencher, B. Schaalje, *Linear Models in Statistics*, 2nd edn. (Wiley, New York, 2000)
51. R.E. Roberts, C.R. Roberts, I.G. Chen, Fatalism and risk of adolescent depression. Psychiatry: Interpersonal Biol. Process. **63**(3), 239–252 (2000)
52. M.A. Russell, *Mining the Social Web* (O'Reilly, Sebastopol, 2011)
53. D. Sarkar, *Lattice Multivariate Data Visualization with R* (Springer Science Business Media, New York, 2008)
54. S.J. Sheather, M.C. Jones, A reliable data-based bandwidth selection method for kernel density estimation. J. R. Stat. Soc. B **53**, 683–690 (1991)
55. A. Signorini, A.M. Segre, P.M. Polgreen, The use of Twitter to track levels of disease activity and public concern in the U.S. during the influenza A H1N1 pandemic. PLoS One **6**(5) (2011)
56. S.S. Skiena, *The Algorithm Design Manual*, 2nd edn. (Springer, New York, 2008)
57. B. Slatkin, *Effective Python* (Addison-Wesley Professional, Upper Saddle River, 2015)
58. B.M. Steele, Exact bagging of k-nearest neighbor learners. Mach. Learn. **74**, 235–255 (2009)
59. N. Super, The geography of medicare: explaining differences in payment and costs. Natl Health Policy Forum (792) (2003)
60. R.D. Telford, R.B. Cunningham, Sex, sport and body-size dependency of hematology in highly trained athletes. Med. Sci. Sports Exerc. **23**, 788–794 (1991)
61. Twitter Inc., API Overview. https://dev.twitter.com/overview/api/tweets. Accessed 14 June 2016
62. Twitter Inc., Firehose. https://dev.twitter.com/streaming/firehose. Accessed 14 June 2016
63. W.G. Van Panhuis, J. Grefenstette, S.. Jung, N.S. Chok, A. Cross, H. Eng, B.Y. Lee, V. Zadorozhny, S. Brown, D. Cummings, D.S Burke, Contagious diseases in the United States from 1888 to the present. N. Engl. J. Med. **369**(22), 2152–2158 (2013)
64. W.N. Venables, B.D. Ripley, *Modern Applied Statistics with S*, 4th edn. (Springer, New York, 2002)
65. H. Wickham, *ggplot2: Elegant Graphics for Data Analysis (Use R!)* (Springer, New York, 2009)
66. H. Wickham. *ggplot2* (Springer, New York, 2016)
67. J. Wiebe, R. Mihalcea, Word sense and subjectivity, in *Joint Conference of the International Committee on Computational Linguistics and the Association for Computational Linguistics* (2006)
68. Wikipedia, List of zip code prefixes - Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/List_of_ZIP_code_prefixes. Accessed 30 Apr 2016
69. L. Wilkinson, *The Grammar of Graphics*, 2nd edn. (Springer, New York, 2005)
70. I.H. Witten, F. Eibe, M.A. Hall, *Data Mining: Practical Machine Learning Tools and Techniques*, 3rd edn. (Morgan Kaufmann, Burlington, 2011)
71. X. Zhuo, P. Zhang, T.J. Hoerger, Lifetime direct medical costs of treating type 2 diabetes and diabetic complications. Am. J. Prev. Med. **45**(3), 253–256 (2013)

Index

B. Steele et al., *Algorithms for Data Science*,
DOI 10.1007/978-3-319-45797-0