

*Deborah Nolan, University of California, Berkeley
Duncan Temple Lang, University of California, Davis*

Data Technologies and Computational Reasoning

Preface

This data science text aims to integrate statistical and computational reasoning in the context of problem solving with data. We examine questions in science, social science, business, industry, and government and work with data from within these contexts as we attempt to answer the questions. We hope that by reading this material and trying the exercises and projects, the reader will:

- Gain the essential skills needed to engage in data-analysis-based problem solving and computational reasoning;
- Express ideas clearly, precisely, and concisely through code;
- Apply statistical thinking throughout the data analysis cycle, from data acquisition and cleaning to organization and analysis to communicating results; and
- Build the confidence needed to overcome future computational challenges.

These are broadly stated goals, and in order to achieve them we have created a sequence of materials covering a wide range of computational and statistical topics for readers spanning the spectrum of experience from those who have never programmed to those who regularly write code and are looking for guidance in advancing their abilities and filling in gaps in their skill set. Overall, we provide an introduction to elementary aspects of programming, program non-trivial tasks, explore how to make code more efficient, and cover a variety of data technologies for compare and contrast several different technologies for accessing, cleaning, merging, and ‘munging’ data from a variety of sources. Simultaneously, we cover many statistical topics, including exploratory data analysis, visualization, resampling techniques such as the bootstrap, jackknife and cross-validation, and the Monte Carlo method. We also introduce a broad array of modern statistical and machine learning methods. The intent is that data scientists can use this material to learn about most of the topics they are expected to use in their daily work.

The material is divided into 4 parts. We briefly describe these parts and include descriptions of the learning goals for each of the 14 chapters.

I. Scripting and Exploratory Data Analysis

This first part in the data science series introduces the reader to the basics of *R* and helps them become fluent in manipulating and exploring data and creating informative visualizations. We assume no prior experience with a programming language. Yet, someone who routinely uses *R* may want to dip into this material. For example, he or she may want to read the summary boxes or a particular section that highlights the core features of the language and why they are so useful in data analysis. In this part, we attempt to present the paradigms of the language and avoid code recipes for handling data. Moreover, we model the EDA mindset as we read and examine data, and we present guidelines for

making good graphics that may interest any level of reader. We think this part can be used in an introductory course that focuses on elementary *R* programming for reading data into *R*, transforming data into different structures, practicing exploratory data analysis, and critiquing and composing informative data visualizations.

Introduction to *R*

In this chapter, we introduce the statistical programming language *R*. We demonstrate how to express computations in the *R* language and how to think about the computational paradigms of the language. Rather than simply provide the nuts and bolts and code templates, we aim to explain the essential features of the language and how to think about *R*'s computational model. This includes a description of how these features help statisticians with their work. One goal for the reader is to understand the Read-Evaluate-Print-Loop (REPL), including being able to write simple expressions and evaluate them; read and correct the syntax of an expression; and use boolean algebra and logical expressions. Another goal is to become familiar with a handful of functions in *R* that perform simple statistical summaries or provide information about a data object. This includes being able to call a function, passing in the appropriate arguments, and to read documentation for a function to determine how to invoke it. We also introduce simple data objects, such as vectors and data frames, and the primitive data types, and how to create subsets of them.

Reading and Exploring Data in Tables

This chapter on data tables and exploratory data analysis aims to get the reader handling data that are organized in plain-text, rectangular formats. We provide examples of how to read, clean, and validate data sets and the exploratory phase of data analysis. More specifically, we distinguish between various formats of plain text data and map them into rectangular data structures. These formats include comma-separated values, tab-delimited values, fixed width format, and key-value pairs. We also demonstrate how to: perform simple summaries and visualizations to validate the data have been properly read and the values are as expected; initiate simple corrections to the process of reading the data and modify the data after it has been read to, e.g., identify missing values, rename variables, convert data values to a format more suitable for analysis, and create new variables from those supplied; determine how to rearrange values in a table so that the resulting structure is more conducive to analysis—this may require stacking columns of variables that contain the same information, transposing rows of values into columns, or creating arrays of data tables.

Exploratory Data Analysis (EDA) features prominently in this chapter. We focus on how to conduct initial explorations of data, interpret the findings, consider new directions to take, and continue exploration.

Visualization

Visualization plays an important role in exploration and presentation and this chapter touches on both goals for making statistical graphs. We address the process of how to: select an appropriate plot to make based on the type of data being analyzed; describe the shape of a distribution and relationships between two variables; apply the method of comparison to examine distributions and relationships for subgroups of data, augment plots with useful information, and consider transformations of variables; bring the data provenance into consideration of the implication of the findings to other, possibly more general, settings. We introduce basic guidelines for choosing colors, scaling axes, and transforming variables in order that plots make the data stand out, facilitate comparisons, and are information rich. Additionally, we model the iterative process of how beginning with a simple plot we can continue to refine and augment it while following these guidelines, and we demonstrate how at times we must create a unique sort of plot in order to best convey our discoveries.

With the exception of one section, we present this material in a context that does not

depend on the programming language. In that section we introduce the graphics model in both base *R* and *ggplot2* and provide code examples that demonstrate how to create a few of the figures in the chapter with these two approaches.

Reading and Exploring Complex Data

This chapter on complex data structures provides examples of handling data that are not simple table-like structures. As in Chapter 2, we focus on the thought process behind decisions about how to read and organize data. Here, the data are not necessarily easily mapped into a table or data frame. The goals are to: identify semi-structured data that cannot be conveyed in simple delimited text files that naturally map to a data frame or table, e.g., *JSON*-formatted data and ragged arrays where records have varying number of observations; carry out the process of organizing these data into a structure suitable for analysis, including how to handle data at different levels of granularity and how to organize the data for analysis; and for data in an *R* list structure, determine its organization and how to access an element, take a subset, and apply a function to each element.

II. Programming, Resampling, and Monte Carlo

In this part, we cover the basics of programming, including how to write, test, and debug functions. We assume no experience in programming and identify the steps in writing a function and demonstrate several core computational paradigms, such as control flow and call frames. In subsequent chapters, our reader gains experience in programming through several examples that address how to model and make inferences from data, develop prediction methods, and understand properties of stochastic processes and statistical estimators. Included with these motivating examples is an introduction to concepts in Monte Carlo simulation of stochastic systems and to resampling techniques for statistical inference and prediction. We expect this material to be accessible to a reader with little statistics background as we attempt to construct understanding of statistical ideas through simulation and resampling. Furthermore, as we work through many data-analysis problems, we aim to convey computational reasoning and develop expertise in writing well-tested, modular code.

Programming Concepts

This chapter provides a basic introduction to programming, including how to compose a function and write expressions that control the flow of evaluation of code. We demonstrate how to map the description of a simple task into a function; we first write code for a specific example and then generalize it by defining a function, encapsulating the code we have written into the function, and identifying the inputs to the function and their default values (if any). We describe the notion of a call frame and how variables and their inputs are set up when a function is invoked, including argument matching and lazy evaluation. We augment the sequential evaluation of expressions with how to conditionally evaluate expressions and how to use a loop mechanism to repeat the evaluation of code. Also in this chapter, we explore simple debugging strategies, e.g., how to distinguish between different types of errors, decode syntax errors, and set up test cases to confirm the code works as expected. Additionally, we introduce style guidelines for writing code so that our code is clear, consistent, and readable.

Resampling for Inference and Prediction

In this chapter, the reader gains experience with programming in the context of problem solving with resampling methods. The computational topics include: modularity, where we

identify subtasks, develop small functions to carry out these tasks, and combine them to solve a data analysis problem; testing, where we develop test cases for debugging code and demonstrate how to trace call frames and browse the variables as our code executes to uncover the source of errors; profiling code to find where bottlenecks occur; and redesign our code to improve computational efficiency; and developing more complex function signatures that use, e.g., ... and functions as parameters. The examples examine sampling distributions and approximate standard errors computed with the jackknife and bootstrap. A case study is worked where cross-validation is used in a nearest neighbor setting to select the number of neighbors.

Simulation and Stochastic Modeling

This chapter has the same computational goals as the chapter on resampling. The problems originate from trying to understand the behavior of a random quantity, such as the sampling distribution of a statistic calculated from data where the data follow an unusual distribution that arises from the manner in which the data are collected (e.g., censoring, truncation, or a mixture of 2 distributions). This chapter also includes a comprehensive case study that demonstrates the topics of modularity, testing, profiling, and efficiency. Specifically, the behavior of the Biham–Middleton–Levine traffic model is explored through simulation. This case study is revisited in the advanced programming part to demonstrate object-oriented programming and how to interface to *C* code.

III. Data Technologies

Each chapter in this part aims to familiarize readers with an essential data technology used in data science. As with the earlier chapters that introduce *R* and programming, our philosophy is to bring to the fore the core concepts in the technology and provide examples that demonstrate how and when the technology can be used. The technologies included here address the topics of accessing data via Web scraping and APIs, relational databases, text processing, and shell tools. We introduce specific technologies related to these topics, including *HTML*, *XML* and *XPath*; *SQL*; regular expressions; and shell commands. We describe their core concepts without reference to *R*. However, in order to work through the examples and address the problems posed, we expect the reader to have basic expertise in *R* at the level introduced in the first two parts of this series.

Text Manipulation and Regular Expressions

In this chapter, we demonstrate the fundamental ideas behind the regular expression language, introduce the basic elements of the language, and show how they can be combined to express patterns. We aim to provide the reader with skills necessary to work with text data, including being able to: modify strings with simple string manipulation functionality to split up and paste strings together; read and understand the syntax of a regular expression and construct regular expressions to search for patterns in strings. We introduce meta characters to modify a pattern, create alternate patterns, and identify sub-patterns. In the examples, we clean text to create variables for analysis and perform more intensive text mining. Text mining tasks including being able to remove words from a corpus, reduce related words to their stem, tally word counts for documents, and construct measures of word frequency to compare documents.

Relational Databases and SQL

In this chapter, we describe the concept of a relational database (RDBMS), introduce the

structured query language (SQL) to extract data, and consider how to work with data in a RDBMS from within *R*. More specifically, the goals of this chapter are for the reader to able to: understand the advantages of using a relational database to store data, including issues related to security, redundancy, and efficiency; read schema and describe the organization of tables in a relational database; understand the syntax of a SELECT statement; perform simple extractions from a single table in a database; and merge and combine information within and across tables. We use *R* functions to interface with a database, and we draw parallels between comparable tasks for data frames in *R* and tables in a RDBMS. In addition to column-oriented data bases, we briefly address the topic of other types of data bases, such as noSQL, and when they are useful. These are covered in more detail in Chapter 12.

Shell Tools

In this chapter, we introduce the shell language and perform simply command-line tasks. More specifically, the reader is introduced to syntax of a shell expression, including how to specify arguments, pipe commands, and identify syntax errors. We provide examples of how to use simple shell commands to manage files, e.g., copy, move, delete, and examine file contents, and how to manage processes, including remotely accessing files and running programs, e.g., with terminal multiplexers such as Screen and tmux. Finally, we explore files and their contents through simple shell commands and combinations of these commands. We also discuss reproducibility and versioning systems in this chapter.

XML

In this chapter, the reader learns about the structure of an *XML* document, including understanding the document's hierarchy of elements and what it means to be well-formed. Importantly, we cover the concept of how to locate content in the document with *XPath* expressions and demonstrate how to extract content to create data structures amenable for analysis.

Web Technologies

In this chapter the reader learns how to scrape data from Web pages and tables, including using *XPath* to locate content in *HTML* and *XML* documents. We also introduce the ideas underlying *HTTP* and provide examples of how to access data via forms and *RESTful* Web APIs. We briefly cover the basics of authentication in Web services and pages.

IV. Data Science Projects and Case Studies

Many fundamental statistical concepts and methods appear throughout the other parts in this book. This particular part supplements the earlier parts with projects and a fully worked case study. We provide descriptions of a dozen open-ended projects that integrate material from multiple computational topics in a modern data problem. These projects and the examples found throughout the text differ from traditional computer science assignments and data analysis projects in that they require both a demonstration of computing aptitude and insightful data analysis. Many introduce computationally intensive statistical methodologies, such as recursive partitioning, nearest neighbor methods, and hierarchical clustering, that are relatively easy to understand from an algorithmic perspective, that are not typically taught until late in an undergraduate statistics curriculum if at all, and that can be exciting to apply to a data analysis problem. Other projects focus on simulation studies of a complex system. Typically, there is not one correct answer for a project, and they can be solved using very different statistical methods and approaches from those in-

cluded with the project description. For example, the naive Bayes method is described in one project as an approach for classifying spam, but recursive partitioning, which appears in the description of the indoor positioning project, can be used instead for the spam case study. Finally, also in this part, we provide a case study that demonstrates how a data scientist might tackle large and/or complex data in a reproducible and systematic way.

Student Work

This book contains 3 different types of assignments. The Guided Practice problems found at the end of each chapter serve to reinforce the basic concepts introduced in the chapter. These problems are taken from our weekly laboratory assignments that we designed for students to gain experience with the material before embarking on more open-ended homework and projects. We provide solutions for these problems including detailed descriptions and alternative coding approaches.

The section called Exercises that follows the Guided Practice consists of problems that we typically assign for homework. These problems tend to be more open-ended than the guided practice problems but more structured than the projects. Many of our homework assignments have been split into several contiguous exercises in this section. Lastly, as mentioned already, projects can be found in their own part of this series. These projects typically require mastery of multiple computational topics, synthesis of statistical and computational thinking, and learning about and applying a new statistical methodology.

Selecting Material for a Course

This series goes beyond the basics to teach the practice and process of programming and computational reasoning for data science, and the material can be combined in different ways for different courses. We have experience teaching all of this material in a sequence of 2 10-week courses on the common computational tasks of data science. The first course in this series covers much of the material in Parts I and II, and the second focusses on the technologies found in Part III. The projects in Part IV are used as motivating examples and assignments in both courses. We provide a list of topics and corresponding chapters for each of the courses in Table 0.1. Note that there is some overlap from one course to the next.

We have also covered much of the material here in a 15-week course called Concepts in Computing with Data. This course has no programming or statistics pre-requisites and is aimed at the sophomore/junior level. In Table 0.2, we provide a sample syllabus that includes a week-by-week list of topics and readings.

Available Materials

The Web site <http://rdatasciencecases.org> provides data, code, and supplementary material for this book. It also provides ideas and details for additional case studies. We hope others will contribute their case studies so that we can make these available to the community at large.

TABLE 0.1: Topic List for a 2-course (20 week) Sequence in Data Science

Session	Topic	Reading
A	Introduction to <i>R</i>	Chap 1
A	Reading Data	2.1-4, 4.1-6
A	Programming	Chap 5
A	Simulation & Resampling	6.1-2, 6.4, 7.1-2, 7.6-7
A	String Manipulation & Regular Expressions	Chap 8
B	Review Programming	Chap 5
B	Text Manipulation & Regular Expressions	Chap 8
B	Relational Databases	Chap 9
B	Shell Tools	Chap 10
B	Web Technologies	Chap 11

TABLE 0.2: Sample Syllabi for a 15-week Data Science Course

Week	Topic	Reading
1	Introduction to <i>R</i>	1.1-7, 1.13-15
2	Vectorized operations & EDA	1.8-9, 2.7
3	Graphics	3.1-5
4	Data Structures	1.10-11, 2.4-6, 4.2
5	Introduction to Programming	5.1-6
6	Simulation & Project	5.7, 7.1-3, 15.11
7	Review & Midterm	
8	Debugging & Style	5.8-9
9	Shell	2.1-3, 10.1-5
10	Regular Expressions & Text mining	8.1-5, 15.9
11	XML & Simple Web Scraping	11.2-3, 4.4
12	SQL	9.1-5
13	Project & related topics, e.g., resampling	6.1-2, 6.4
14	Project-related topics, e.g., nearest neighbor	6.4, 15.1
15	Review & Project help	

Typographic Conventions

In this series, we use other languages in addition to *R*, such as *SQL*. While the context should clearly indicate that a code block is in a language other than *R*, we also specify this in the margin of the page. For example, a *UNIX grep* command appears as

Shell `grep position2d JRSPdata_2010_03_10.log | grep -v ' 004 '`

In the process of writing code, we introduce errors and mistakes with the idea that these are useful for learning how best to approach and solve computational problems. For this reason, some of the code in this book is purposefully incorrect or ill-advised (i.e., it works but is not a good approach). We identify such code with a no-entry symbol in the page margin, e.g.,

🚫 `createGrid(c(3, 5), .5)`

```
Error in grid[pos] = sample(rep(c("red", "blue"), numCars)) :  
  (converted from warning) number of items to replace is not  
  a multiple of replacement length
```

We note that the code we present for creating the figures differs slightly from the code we actually used to create the displayed figures. Furthermore, typically we add titles to our plots, either for interactive viewing or for including in presentations and reports. However, we have not done that in this book because the graphics are displayed in figures which include their own captions and titles. In order to avoid redundancy, we have eliminated the specification of a plot title in our code.

Acknowledgments

We first started teaching a course based on these ideas at our respective institutions in Spring 2005. Over the past dozen years, we have continued to develop and refine these materials, created new courses for different levels of students, and provided materials for our colleagues at our and other institutions to teach data science. We thank our colleagues for their feedback on these materials and this approach to teaching data analysis and data science in an integrated framework. In the process of developing our course materials and student assignments, we developed many case studies, which we spun off into a collection that appears as a separate publication, *Data Science in R: A Case Studies Approach to Computational Reasoning and Problem Solving*. This collection includes contributions from others and we thank them for their contributions and impact on our thinking. Most importantly, we owe a tremendous debt of gratitude to the graduate students who have assisted in the teaching of our courses and the undergraduate students who have taken our courses and provided valuable feedback and enthusiasm. We especially thank, Gabe Becker, Erica Christianson, Clark Fitzgerald, Inna Gerlovina, Christine Ho, Neal Fultz, Karl Kumbier, Gang Liang, Bitao Liu, Jarrod Millman, Sandy Nathan, Kellie Ottoboni, Bradly Stadie, Nick Ulle, and Charlotte Wickham.

We also would like to acknowledge the grants that in part supported this work. Specifically, this material is based in part upon work supported by the National Science Foundation under Grant Numbers DUE-0618865, DMS-0840001, and DUE-1043634.

Contents

I Scripting and Exploratory Data Analysis	1
1 Introduction to R	3
1.1 Introduction	4
1.2 Getting Started	4
1.3 Computations and Expressions	4
1.3.1 Arithmetic Expressions and Order of Operations	5
1.3.2 Call Expressions	6
1.4 Variables and Assignment	7
1.5 Syntax and Parsing	9
1.6 Data Types	12
1.6.1 Finding Information on Vectors	13
1.7 Vectorized Operations	15
1.7.1 Aggregator Functions	17
1.7.2 Relational Operations	17
1.7.3 Boolean Algebra	18
1.7.4 Coercion	19
1.8 Data Frames	21
1.9 Subsets	24
1.9.1 Subsetting with Logical Vectors	24
1.9.2 Subsetting by Position	26
1.9.3 Subsetting by Exclusion	28
1.9.4 Subsetting by Name	29
1.9.5 Subsetting All	30
1.9.6 Assigning Values to Subsets	30
1.9.7 Subsetting Data Frames	31
1.10 Applying Functions to Variables in a Data Frame	32
1.11 Creating Vectors	34
1.11.1 Concatenating Values into a Vector	34
1.11.2 Creating Sequences of Values	35
1.11.3 Functions for Operating on Vectors	36
1.11.4 Functions for Manipulating Character Vectors	37
1.11.5 Creating Vectors from Different Types	37
1.12 Recycling Rules	38
1.13 Managing Your Workspace	38
1.13.1 Storing and Removing Variables	39
1.13.2 Searching for Variables	40
1.14 Getting Help	42
1.15 Software for Statisticians	42
1.16 Summary	44
1.17 Summary of Basic Functions for Working with Vectors and Data Frames . .	45
1.18 Guided Practice	47

1.19 Exercises	50
2 Reading and Exploring Data in Tables	51
2.1 Introduction	51
2.2 Formats for Tabular Data	53
2.2.1 Delimited Data	54
2.2.1.1 Comma-Delimited Traffic Data	54
2.2.2 Fixed Width Format	56
2.2.2.1 Fixed Width Formatted Drug Abuse Warning Network Survey	56
2.2.3 Key-Value Pairs	58
2.3 Validating and Cleaning the Data	59
2.3.1 Updating Variable Names and Formatting Time for Traffic	60
2.3.2 Data Types for DAWN Survey	61
2.3.3 Validating Text – Hillary Clinton’s Email	64
2.4 Selecting a Structure: Data Frame, Matrix and Array	68
2.4.1 Collections of Matrices – Handwritten Digits Data	68
2.4.1.1 Applying Functions to Matrices and Arrays	70
2.5 Reshaping Data Tables	71
2.5.1 Stacking Traffic Flow	72
2.5.2 Rearranging World Bank Country Statistics	75
2.6 Merging Data Tables	81
2.6.1 Merging Names and Emails	82
2.7 Exploratory Data Analysis	85
2.7.1 Exploring Traffic on California Freeways	86
2.7.2 Exploring Country Statistics	91
2.7.3 Exploring Emergency Room Visits due to Drug Abuse	94
2.7.4 EDA Summary	99
2.8 Summary	102
2.9 Functions for Reading and Exploring Data	103
2.10 Guided Practice	104
2.11 Exercises	105
3 Visualization	107
3.1 Introduction	107
3.1.1 Composing a Graph of Voter Registration Trends	108
3.2 Data Types and Plot Choice	111
3.2.1 Terminology	111
3.2.2 The Kaiser Family Data	111
3.2.3 Univariate Plots	113
3.2.4 Bivariate Plots	117
3.2.5 Conveying Relationships between 3 or More Variables	122
3.2.6 Scalability – Real Estate Data with 500,000 records	125
3.2.7 Measurements in Time	128
3.2.8 Geographic Data	129
3.3 Guidelines	131
3.3.1 Scale	135
3.3.2 Position	135
3.3.3 Shape	136
3.3.4 Aggregates	136
3.3.5 Color	136

3.3.6 Context	137
3.3.7 Over Arching Considerations	137
3.4 Iterative process	138
3.5 Rs Graphics Models	139
3.5.1 Painter’s Model in Base R	139
3.5.2 Grammar of Graphics Model in <code>ggplot2</code>	142
3.6 Creating Unique Plots	146
3.7 Summary	146
3.8 Exercises	146
4 Reading and Exploring Complex Data	149
4.1 Introduction	149
4.2 Lists	155
4.2.1 Subsetting Lists	157
4.2.1.1 Accessing An Element of a List	158
4.2.2 Applying Functions to Elements of a List	160
4.2.3 Exploring Rainfall on the Colorado Front Range	161
4.3 Reading Data into a Character Vector	166
4.3.1 A Study of Web Page Updates	167
4.4 Reading Data from a Web Page	172
4.4.1 World Records in the Men’s 1500 meter	174
4.5 Reading JSON Formatted Data	175
4.6 Kiva	181
4.6.1 Loan Elements	183
4.6.2 The Payments	186
4.6.3 The Borrowers	186
4.6.4 Final Structure	187
4.7 Summary	187
4.8 Functions for Handling Complex Data Formats	187
4.9 Guided Practice	188
4.10 Exercises	189
II Programming, Monte Carlo, and Resampling	191
5 Programming Concepts	193
5.1 Introduction	193
5.2 Steps in Writing Functions	196
5.2.1 Calculating BMI from Height and Weight	196
5.2.2 Generalizing Code	198
5.2.3 Commenting Code	199
5.2.4 Reporting Run Times in Seconds	199
5.2.5 Taking the Logarithm of ‘0’	201
5.3 Defining a Function	202
5.3.1 Anonymous Functions	204
5.4 Calling a Function	205
5.4.1 Argument Matching by Name and Position	205
5.4.2 The ... parameter	207
5.4.3 Lazy Evaluation	208
5.4.4 The Search Path	209
5.4.5 Partial Matching	211
5.5 Exiting a Function	211

5.6	Conditionally Invoking Code	212
5.6.1	Conditionally Modifying Inputs to <code>log(0)</code>	214
5.6.2	Converting Liquid Measures into Tbs. Using <code>switch()</code>	220
5.7	Iterative Evaluation of Code	221
5.7.1	Reading Thousands of Files of Data With a <code>for</code> Loop	221
5.7.2	Generating Fibonacci Numbers	223
5.7.3	Playing Penney's Game	228
5.8	Style Guidelines	233
5.9	Beginning Notions of Debugging	237
5.10	Summary	240
5.11	Control Flow and Debugging Functions	240
5.12	Guided Practice	241
5.13	Exercises	244
6	Resampling for Inference and Prediction	245
6.1	Introduction	245
6.2	The Bootstrap and Inference	247
6.2.1	Percentile Bootstrap of Median House Price	247
6.2.2	Studentized Bootstrap for Repair Times	251
6.2.3	Jackknife Estimate of the Standard Error for Average Repair Time .	261
6.3	Permutations and Testing for a Gender Effect	267
6.4	Cross-Validation and Prediction	275
6.4.1	Nearest Neighbor Prediction	280
6.4.2	Hold Out Test Set	290
6.4.3	v-fold Cross Validation	296
6.4.4	Lazy Evaluation	299
6.5	Summary	299
6.6	Guided Practice	299
6.7	Exercises	299
7	Simulation and Stochastic Modeling	301
7.1	Introduction	301
7.2	Monte Carlo Studies	303
7.2.1	Monte Carlo Study of Magnetic Resonance Scans	304
7.3	Random Number Generation	312
7.3.1	The Cumulative Distribution Function	313
7.3.2	Congruential Methods	313
7.4	Inverse Probability Sampling	315
7.4.1	Simulating the Censored Log-Normal	317
7.4.2	Simulating the Truncated Normal Distribution	318
7.5	Acceptance-Rejection Sampling	320
7.5.1	Tail of a Distribution	323
7.6	Simulation Study of a Location Estimator for a Mixture Distribution . . .	326
7.7	Simulating the Biham-Middleton-Levine Model for Traffic	334
7.7.1	The Initial Traffic Configuration	335
7.7.1.1	Testing the Grid Creation Function	338
7.7.2	Displaying the Grid	341
7.7.3	Moving the Cars	345
7.7.4	Evaluating the Performance of the Code	350
7.8	Summary	358
7.9	Exercises	358

III Data Technologies	359
8 Text Manipulation and Regular Expressions	361
8.1 Introduction	361
8.2 Basic Concepts in Pattern Matching	364
8.2.1 Matching Literals	364
8.2.2 Modifiers and Meta Characters	365
8.2.3 Equivalent Characters	366
8.3 Using Regular Expressions in R	370
8.3.1 Writing Our Own Literal Matcher	374
8.4 Alternatives, Grouping, and Backreferences	376
8.4.1 Grouping Characters into Sub-patterns	376
8.4.2 Alternative Patterns	376
8.4.3 Back referencing a Sub-pattern	377
8.5 Greedy Matching	377
8.5.1 Lazy Matching	378
8.6 Examples	378
8.7 Summary	384
8.7.1 Summary	384
8.7.2 Resources	384
8.8 Summary of Pattern Matching and String Manipulation Functions	385
8.9 Text Mining & Natural Language Processing	386
8.10 Guided Practice	386
8.10.1 Answers	387
8.11 Exercises	392
9 Relational Databases and SQL	393
10 Shell Tools	395
10.1 Digital Information	396
10.1.1 Bits and Bytes	396
10.1.2 Representing Numbers in Binary	396
10.2 Files and File Systems	396
10.2.1 Plain Text Files	396
10.2.2 Binary Files	396
10.2.3 File Systems	396
10.2.4 Permissions	396
10.3 Basics of a Shell Command	396
10.3.1 Syntax	396
10.3.2 Getting Help	396
10.4 Managing and Navigating a File System	396
10.4.1 Wildcards	396
10.5 Redirection	396
10.6 Managing Processes with the Shell	396
10.6.1 Running in Batch	397
10.7 Remote Login	397
10.7.1 Screen/tmux	397
10.8 Cases	397
10.9 Advanced Topics	397
10.9.1 grep	397
10.9.2 cut	397

10.10	Summary	397
10.11	Guided Practice	397
10.12	Exercises	397
11	XML	399
11.1	Overview of <i>XML</i>	399
11.2	Hierarchical Structure	407
11.3	<i>XML</i> Namespaces and Additional <i>XML</i> Elements	410
11.4	The Document Object Model (<i>DOM</i>)	412
11.5	Accessing Nodes in the <i>DOM</i>	414
11.6	<i>XPath</i> and the <i>XML</i> Tree	421
11.7	<i>XPath</i> Syntax	425
11.7.1	The Axis	426
11.7.2	The Node Test	429
11.7.3	The Predicate	429
11.8	Summary of Functions to Read <i>HTML</i> , <i>XML</i> , and <i>JSON</i> into R Data Frames and Lists	431
12	Web Technologies	433
12.1	Introduction	433
12.2	Before You Scrape!	434
12.3	Scraping Data from <i>HTML</i> Tables	435
12.3.1	435
12.3.1.1	<i>HTML</i> Forms	437
12.3.2	Specifying the Column Types	441
12.4	Scraping Links from <i>HTML</i> Pages	441
12.5	Overview of <i>HTML</i> and <i>XML</i>	441
12.6	Scraping Arbitrary Content from <i>HTML</i> Pages with <i>XPath</i>	441
12.6.1	Finding Patterns in the <i>HTML</i> Content and using <i>XPath</i>	441
12.7	Scraping Data from <i>HTML</i> Forms	441
12.8	Dynamic <i>JavaScript</i> Content with Selenium	441
12.9	Basics of <i>HTTP</i> Queries	441
12.10	REST-based Web Services	441
12.10.1	<i>XML</i>	441
12.10.2	<i>JSON</i>	441
12.11	Summary	441
12.12	Guided Practice	441
12.13	Exercises	441
IV	Data Science Projects and Case Studies	443
13	Projects	445
13.1	Digit Recognition	445
13.1.1	Goal	445
13.1.2	Background	445
13.1.3	The Data	446
13.1.4	Tasks	447
13.1.5	k-Nearest Neighbors	448
13.1.6	Cross Validation and Model Selection	449
13.2	Adhoc Networks (Simulation Study)	451

13.3 Graphs: perhaps voting records of senators, citation networks, Enron Email (graphs)	451
13.4 Web Cache (Poisson arrival process)	451
13.5 Presidential Election (Maps)	451
13.6 NASA Satellite	451
13.7 Indoor Positioning System (Nearest neighbor and Cross-Validation)	451
13.8 Spam Filtering (classification trees)	451
13.9 State of the Union Speeches (Term Frequencies and Hierarchical Clustering and Multi-dimensional Scaling)	451
13.10 (Large Data)	451
14 Case: Reading Data with a Diverse Set of Approaches	453
Index	455

Part I

Scripting and Exploratory Data Analysis

1

Introduction to R

CONTENTS

1.1	Introduction	4
1.2	Getting Started	4
1.3	Computations and Expressions	4
1.3.1	Arithmetic Expressions and Order of Operations	5
1.3.2	Call Expressions	6
1.4	Variables and Assignment	7
1.5	Syntax and Parsing	8
1.6	Data Types	12
1.6.1	Finding Information on Vectors	13
1.7	Vectorized Operations	15
1.7.1	Aggregator Functions	16
1.7.2	Relational Operations	17
1.7.3	Boolean Algebra	18
1.7.4	Coercion	18
1.8	Data Frames	20
1.9	Subsets	24
1.9.1	Subsetting with Logical Vectors	24
1.9.2	Subsetting by Position	25
1.9.3	Subsetting by Exclusion	27
1.9.4	Subsetting by Name	28
1.9.5	Subsetting All	29
1.9.6	Assigning Values to Subsets	30
1.9.7	Subsetting Data Frames	31
1.10	Applying Functions to Variables in a Data Frame	32
1.11	Creating Vectors	33
1.11.1	Concatenating Values into a Vector	34
1.11.2	Creating Sequences of Values	34
1.11.3	Functions for Operating on Vectors	35
1.11.4	Functions for Manipulating Character Vectors	36
1.11.5	Creating Vectors from Different Types	37
1.12	Recycling Rules	37
1.13	Managing Your Workspace	38
1.13.1	Storing and Removing Variables	39
1.13.2	Searching for Variables	40
1.14	Getting Help	42
1.15	Software for Statisticians	42
1.16	Summary	44
1.17	Summary of Basic Functions for Working with Vectors and Data Frames	45
1.18	Guided Practice	47

1.19 Exercises	50
Bibliography	50

1.1 Introduction

In this chapter, we introduce the statistical programming language *R*. We learn how to express computations in the *R* language, and we take the time now, as we learn the basics, to understand the computational paradigm of the language. Rather than simply provide the nuts and bolts and code templates, we aim to explain the essential pieces of the language and how to think about *R*'s computational model. If we begin our introduction to *R* and go beyond a utilitarian approach to the language, then we can greatly simplify our programming tasks in the future.

While *R* is a specific language that you may or may not use extensively in the future, what you learn as we explore *R* is generally applicable to many different programming languages that you might use. For example, *R* is very similar to *Matlab*, and it also shares many of the same concepts as *Perl*, *Python*, *Java*, *C*, and *FORTRAN*. Although they are different languages, they share important commonalities that are essential for communicating about computations to others and to the computer.

1.2 Getting Started

We start the *R* programming environment by launching *RStudio* or the *R* graphical user interface (GUI) or by invoking the command *R* in the shell. If you have not installed *R* on your computer, please see the accompanying Web site noted in the preface. Once *R* is running, you should see the *prompt*, `>` in the console. This is where we type *R* expressions. Then, when we press the Return/Enter key, *R* evaluates this expression and prints a value in the console. Try it: enter `1 + 2` at the prompt, i.e.,

```
> 1 + 2
```

Press the Return key, and you should see:

```
[1] 3
```

Then, *R* presents the prompt again and is ready for us to type another expression in the console.

The expression `1 + 2` is an example of a simple arithmetic computation that *R* can perform. We describe these and other types of computations in the next section. Don't worry right now about the `[1]`, we explain what it means in Section 1.6.

1.3 Computations and Expressions

R provides an *interactive* environment where we can give an instruction, such as add 1 and 2, then press the Enter key to have the expression immediately evaluated. The answer is

printed at the console, and we can repeat this process. This sequence of actions is called a read-evaluate-print loop, or REPL for short. When we type an *expression* at the *prompt* in R's console and press the Enter key, we indicate that we want R to perform the *computation*. That is, we ask R to *evaluate* our expression. Here are examples of three kinds of expressions,

```
2 + 3
sample(7)
hist(precip)
```

When the first of these three expressions is evaluated, R *returns* the value 5, which is printed to the console. For the second expression, we ask R to provide a random ordering of the integers from 1 through 7, and R returns something like

```
[1] 3 7 1 4 6 2 5
```

The third expression doesn't print any value to the console. Instead, evaluating this expression yields a plot as a *side effect*. The latter two of these three expressions are function-style expressions. For all three expressions, after the computation has been performed, the loop is complete and the prompt is available for another round of computations.

1.3.1 Arithmetic Expressions and Order of Operations

With R, we can perform many simple arithmetic expressions, similar to what we can do with a scientific calculator. The following are examples of simple arithmetic expressions:

```
8 - 9
4 * 5
10 / 3
7 ^ 2
9 %/% 2
11 %% 7
```

In addition to the basic operations like addition, subtraction, multiplication and division (+, -, *, /), we use \wedge for exponentiation, $\%/\%$ for integer division, and $\%\%$ for modular arithmetic.

Of course, we can combine these arithmetic operations into more complex expressions. For example,

```
10 ^ 5 - 6 / 3
```

```
[1] 99998
```

Recall that 10^5 is 100 thousand, and when we subtract 6 divided by 3 (or 2), we get the result shown here.

Order of Operations

The following expression,

```
10 ^ (5 - 6 / 3)
```

returns a value of 1000, not 99998. Notice that this expression is very similar to $10^5 - 6 / 3$ except for the addition of parentheses. These parentheses change the order of operations, which is why the return value is so different. As expected, the order in which

operations are performed follows the precedence in algebra, i.e., exponentiation, then multiplication and division, followed by addition and subtraction. These operations are carried out from left to right. However parentheses can override this order. The first expression $10^5 - 6 / 3$ has no parentheses so the first computation is to raise 10 to the power 5. Next is the division of 6 by 3, and lastly 2 is subtracted from 10^5 . On the other hand, the second expression places parentheses about $5 - 6 / 3$ so these computations are performed first. That is, we divide 6 by 3 and subtract the result (2) from 5 to get 3. The final result is 10^3 or 1000.

1.3.2 Call Expressions

We called two functions in our discussion of expressions; these were `sample()` and `hist()`. Functions contain code (usually several expressions) that perform a specific task. We provide the functions with inputs to use in carrying out this task. For example, the `abs()` function takes the absolute value of the input provided,

```
abs(-0.8)
[1] 0.8
```

These inputs are called *arguments* and the output from the computation is the *return value*. When we provide a function with a particular set of values for its arguments and press the Return key, we say we are *calling* or *invoking* the function. For now, we work with R's many built-in functions. Later, in Chapter 5, we write our own functions.

We saw already that when we call `sample(7)`, R returns a random permutation of the numbers from 1 to 7, e.g.,

```
sample(7)
[1] 1 4 6 3 2 5 7
```

The input that we provide the `sample()` function is the largest integer in the sequence 1, 2, ... that we want permuted. However, `sample()` can take more than one input. It has 4 arguments; these are

```
args(sample)
function (x, size, replace = FALSE, prob = NULL)
```

The arguments to `sample()` have names, `x`, `size`, `replace`, and `prob`, and 2 of them, `replace` and `prob`, have default values of `FALSE` and `NULL`, respectively. This means that they are optional, i.e., we don't need to provide the inputs for these 2 arguments. If we don't provide them in our function call, then R simply uses the default values. The `size` argument does not have a default value, and if it is not supplied then R does the sensible thing – return a permutation of all of the values in `x`. For example, if we want to sample only 3 random values from the integers from 1 to 7, then we specify `x` (as always) and we also must specify the `size` argument. We do this with

```
sample(7, 3)
[1] 7 1 5
```

There are many ways to specify the arguments to a function call. We consider these in Section 5.4, after we learn more about the syntax of the R language.

Expression Syntax

An expression is an instruction to the computer. The computer evaluates the expressions that we write and returns a value. In order for the software to carry out our instructions, these directions must obey the grammar of the language.

Grammar

The *R* software uses blanks, commas, parentheses, algebraic and relational operators, and naming conventions to figure out the various parts of an expression and so determine the computation to perform. Importantly, if we understand how the software reads an expression, then we can more easily identify and correct syntax errors.

In the expression below:

```
round(abs(x - x^2), digits = 4)
```

R uses the parentheses, minus and exponentiation signs, comma, and equal sign to identify the variables and functions in this expression. Starting from the inner-most expression, the instructions are to square *x*, subtract this quantity from *x*, take the absolute value of the difference, and then round the resulting value to 4 significant digits.

Readability

We can eliminate the blanks in the expression above and code it as

```
round(abs(x-x^2),digits=4)
```

R can parse this expression and carry out the instructions, which are the same as in the previous expression. However, we need to be able to read the expressions that we write too, and it's much easier for us to read code that adopts some of the conventions of written English and places a space after a comma and before and after operators and numbers.

1.4 Variables and Assignment

In the previous section, we saw that when we provide *R* with an expression to evaluate, *R* prints the results to the console as output. We may want to save this output for future computations. We can do this by *assigning* the result of a computation to a name, e.g.,

```
x = 10 ^ 5 - 6 / 3
```

Now, when we type this expression and hit the return key, *R* does not print 99998 to the console. Instead, the return value is assigned to a *variable* named *x*. That is, the equals sign tells *R* to assign the result of the computation to the *variable* *x*. Now *x* is a name by which we refer to the value 99998. We can check the value of *x* by typing *x* at the prompt and hitting return,

```
x
```

```
[1] 99998
```

Here, when we type `x` at the prompt, the ‘computation’ that we have asked *R* to perform is simply to print the value of `x` to the console. We can also change the value associated with `x`, by assigning a new value to it. For example,

```
x = 1 + 3
x

[1] 4
```

This is one of two main ways to assign a value to a variable in *R*. In addition to `=`, we can also assign a value with `<-`, i.e.,

```
x <- 1 + 2
x

[1] 3
```

Either of these two forms of assignment can be used. We suggest that you choose one and stick with it. We consistently use `=` in this book.

Variables allow us to store values without needing to recompute them. Additionally, by storing a value, we reduce redundant calculations, which can help us avoid mistakes. Variables also allow us to write general expressions. For example, the length of the hypotenuse of a right triangle with sides `a` and `b` is

```
sqrt(a^2 + b^2)
```

Here, we can use this formula over and over for different triangles, by changing the values of `a` and `b` and re-evaluating this expression.

R uses “copying” semantics in assignments. That is, when we assign the value of `x` to `y`, then `y` gets the value of `x`, but the variable `y` is not “linked” to `x`. This means that when `x` is changed, `y` does not see that change. In the code below, `x` begins with the value of 3, then *R* copies this value in the assignment statement so that `y` has the value of 3. These two variables are unrelated after that so when we assign `x` the value of 10, `y` remains unchanged. Below is the code for this simple example,

```
x
[1] 3

y = x
x = 10
x

[1] 10

y
[1] 3
```

Again, `y` continues to have the value 3 after `x`’s value has changed.

1.5 Syntax and Parsing

How does *R* know what computations to perform? It breaks down an expression into parts, called tokens. From these parts, it can figure out what to do. This is very similar to how we read and understand text. When we read, we use punctuation, such as a period, comma, semicolon, quotation marks, etc., as well as blanks and capitalization, to make sense of what's written. These conventions help us figure out what the person who wrote the text is saying. For example, the blanks, capitalization and punctuation have been stripped from some text that begins,

```
hatheads...
```

Without blanks to identify words, and punctuation to identify sentences, phrases and contractions, we don't know if this text begins as

Ha! The ad's

as in "Ha! The ad's finished", or

Hat! Heads

as in: "Hat! Heads need to keep warm in winter." The basic conventions that *R* uses for parsing code are listed in Table 1.1.

Tokens

R breaks an expression up into meaningful pieces, called *tokens*. Similar to English, *R* uses blanks and quotation marks to identify tokens. Tokens include arithmetic operations and variable names. In *R*, the expression `2*3+1` and `2 * 3 + 1` are equivalent. The atomic tokens, `*` and `+`, let us know that the number 2 is multiplied by 3 and then 1 is added. The blanks are not needed here, but they make it easier for us to read the expression. On the other hand, the expression `sqrt(17)` and `s sqrt(17)` are definitely not the same. The blank in the second expression implies that we want to call the `qrt()` function, not `sqrt()`. We adopt the convention of placing blanks in expressions to help readability, which we discuss more in Section 5.8.

Naming Conventions

Naming conventions for variables also help *R* parse expressions. Variable names cannot start with a digit or underscore, can contain numbers, upper and lower case letters, and some punctuation. The `.` and `_` are allowed, but most others are not. Also, upper and lower case letters are not the same so `X` and `x` refer to different variables.

New Lines

R uses a 'new line' to parse expressions. We produce a "new line" when we hit the Return key. In all of the expressions we have written so far, we end the expression when we press the Enter key. Then, *R* carries out the calculation and returns the value. However, a new line does not always indicate the end of an expression. We can split an expression over multiple lines in the console. For example,

```
> 10 ^ 5 -
+ 6 / 3
```

```
[1] 99998
```

TABLE 1.1: Parsing R Expressions

Convention	Example
White space	<code>abs(-2)</code> and <code>abs(- 2)</code> are equivalent expressions, but the expression: <code>a bs(-2)</code> is different.
Atomic tokens	In addition to arithmetic tokens like <code>+</code> and <code>^</code> , there is also <code>#</code> which stands for comment, <code>(</code> and <code>)</code> , boolean algebra tokens, such as <code>&</code> and <code> </code> , and relational operators, such as <code>></code> and <code>!=</code> .
Quotation marks	These may be, <code>"Hi"</code> or <code>'Bye'</code> , but they must match, e.g., <code>"My'</code> is not valid.
Naming conventions	For example, <code>x22</code> is a valid name but <code>2x</code> is not because it does not begin with a letter or period.
New line	When working at the console or writing an <i>R</i> script, if we start a new line, then this indicates the end of the expression, unless <i>R</i> detects that the expression is incomplete.

Notice the `+` in the second line above. This symbol appears rather than the typical `>` prompt to indicate that the expression on the first line is not complete and *R* is waiting for us to write the rest of the expression on this continuation line. We can even spread this expression over four lines with

```
> 10 ^
+ 5 -
+ 6 /
+ 3
[1] 99998
```

Each line ends with an arithmetic operator so *R* continues the expression on the next line and looks for the second number. This is not a very clearly written expression! However, we may at times have long expressions and breaking them up across lines helps with the readability of the code. Of course, if we try to break up our expression after, say, the 6, then we do not get the expected results:

```
> 10 ^ 5 - 6
[1] 99994
```

Instead of providing us with a continuation line for us to type `/ 3`, *R* evaluates `10 ^ 5 - 6`. This happens because `10 ^ 5 - 6` is a valid *R* expression so *R* evaluates it and returns the value, 99994. Since we have provided *R* with a valid expression, *R* can't discern that we have not finished writing our expression.

Note that we typically do not include the prompts (`>` and `+`) in our code display. We do here only to make clear how *R* parses these expressions.

Compound Expressions

We have seen several simple call expressions, e.g., `sample(3, 7)` and `sqrt(16)`. A compound call expression is like a compound function in algebra, e.g., $f(g(x))$. Recall from algebra that the return value from evaluating $g(x)$ is passed to f as input to that function. For an example in *R*, say we want the integer part of the square root of `x`. Then, we can pass the return value of `sqrt(x)` to the `floor()` function; that is,

```
floor(sqrt(x))  
[1] 3
```

(Recall that `x` has the value 10). We can make compound expressions with functions that use more than one argument, e.g., `round(sqrt(10), digits = 2)` returns 3.16.

Ill-formed Expressions

An ill-formed expression is one that *R* cannot properly parse. For example,

```
floor(sqrt(x))  
Error: unexpected ']' in "floor(sqrt(x))"
```

Here, we have no return value because *R* can't parse our expression. Do you see the mistake? The error message indicates that the right square bracket is unexpected. What does *R* expect? A right parentheses. If we understand how *R* parses expressions, then that can help us figure out our mistakes and easily correct them. See, if you can spot the errors in the following expressions:

```
round(sqrt(10)), digits = 2  
Error: unexpected ',', in "round(sqrt(10)), "
```

```
round(sqrt(10, digits = 2))  
Error in sqrt(10, digits = 2) :  
  2 arguments passed to 'sqrt' which requires 1
```

Can you understand what the error message tells us about these ill-formed expressions?

Types of Expressions

In the examples below, `x` contains the value 10.

Arithmetic: `2 + x^2`

Arithmetic expressions, are composed with the typical operators, e.g., `+`, `-`, `*`, and `/`, for addition, subtraction, multiplication, and division. The order of evaluation follows the rules of precedence in algebra and parentheses can override. This particular expression evaluates to 102.

Relational: `x > 2`

Relational expressions use the operators such as `<`, `<=`, and `==` for less than, less than or equal to, and equal to, respectively. These expressions evaluate to TRUE or FALSE. This particular expression returns TRUE.

Boolean: `x > 20 | x == 0`

Boolean expressions operate on logical values with operators such as `&` and `!` for and and or. This particular expression evaluates to FALSE.

Assignment: `y = 2 + x^2`

The return value from a computation can be assigned to a variable. This variable can then be used in other expressions. Here `y` is 102.

Call: `sqrt(x)`

Expressions can invoke, or call, functions, e.g., this expression computes the square root of `x`. Functions can have multiple arguments; some arguments may be required, meaning that we must provide a value for them when we call the function. Other arguments may be optional; a default value is provided by the function, and we can

override this default in the call. For example, the `round()` function has an argument `x` with no default value and an argument `digits`, which has a default of 0. We can assign `sqrt(x)` to `z` and round the value in `z` to 1 significant digit with `round(z, digits = 1)` and the return value is 3.2. See Section 5.4 for more details on how to specify parameter values in a function call.

Compound: `round(sqrt(abs(x - y + 5)), 1)`

Functions can be nested, i.e., we can compose functions as in algebra. Here we have nested an arithmetic expression within 3 function calls. The first computation is `x - y + 5`, which returns -87. Then, this is passed into `abs()`, which returns 87, and 87 is passed to `sqrt()` which returns about 9.327, and lastly, this result is the input to the `round()` function. The second argument to `round()` is 1 so 9.3 is returned.

1.6 Data Types

In *R*, *vectors* are the primitive objects. A vector is simply an ordered collection of values grouped together into a single container. Some primitive types of vectors are numeric, logical, and character. A very important characteristic of these vectors is that they can only store values of the same type. In other words, a vector contains values that are homogeneous primitive elements. A *numeric* vector contains real numbers, a *logical* vector stores values that are either TRUE or FALSE, and *character* vectors store strings.

Example 1-1 Vectors of Measurements on a Family

We have created some artificial data on a 14-member family to help us explore many of the concepts in this chapter. These data are available in an RDA file for you to load into your *R* session and follow along by typing in the commands shown. We begin by loading the RDA file with

```
load("family.rda")
```

The names of the family members are in the vector called `fnames`, and we can see its contents with

```
fnames
```

```
[1] "Tom"      "Maya"     "Joe"      "Robert"   "Sue"      "Liz"      "Jon"
[8] "Sally"    "Tim"       "Tom"      "Ann"       "Dan"      "Art"      "Zoe"
```

The numbers [1] and [8] are there to help us keep track of the order of the elements in the vector. We see that "Tom" is 1st, "Maya" 2nd, ..., "Sally" 8th, ..., and Zoe 14th. We can confirm that `fnames` is indeed a character vector by calling the `class()` function and passing the vector `fnames` as input, i.e.,

```
class(fnames)
```

```
[1] "character"
```

There are several other vectors with information on the family, including: `fweight`, weight in pounds; `fbmi`, body mass index (BMI); `foverWt`, whether or not BMI is above 25; and `fsex`, which is `f` for female and `m` for male. These vectors provide examples of several data types. The variable, `fbmi` is a numeric vector,

```
fbmi
```

```
[1] 25.16239 21.50106 24.45884 24.48414 18.06089 28.94981 28.18797
[8] 20.67783 26.66430 30.04911 26.05364 22.64384 24.26126 22.91060
```

and `foverWt` is a logical vector,

```
[1] TRUE FALSE FALSE FALSE FALSE TRUE TRUE FALSE TRUE TRUE
[11] TRUE FALSE FALSE FALSE
```

■

Another type of number in *R* is the *integer*. The integer vector is similar to a numeric vector, except that the values must all be integers. The variable `fage`, which contains the person's age in years, is an example,

```
class(fage)
[1] "integer"
```

Factor

The *factor* is a somewhat special data type for use with qualitative measurements. The values are internally stored as integers, but each integer corresponds to a *level*, which is held as a character string. The values of `fsex`, which is a factor vector, are:

```
[1] m f m m f f m f m m f m m f
Levels: m f
```

Notice that the values are not printed with quotation marks, as with character values. We confirm the data type of `fsex()` with a call to `class()`,

```
class(fsex)
[1] "factor"
```

Also, the `levels()` function provides the levels or labels associated with the vector.

Special Values

R provides a few special values, including `NULL`, `NA`, `NaN`, and `Inf`. These stand for null or empty, not available, not a number, and infinity, respectively. `NULL` denotes an empty vector. The value `NA` can be an element of a vector of any type. It is different from the character string `"NA"`. We can check for the presence of `NA` values in a vector with the function `is.na()` and for an empty vector with `is.null()`.

The special values ‘not a number’ and infinity come about from computations. In particular, they can occur when we divide by 0. Here are three examples that return infinity, negative infinity, and not a number, respectively: `12 / 0`, `-100 / 0`, and `0 / 0`.

1.6.1 Finding Information on Vectors

R has many utility functions that provide information about vectors (and other objects that we will soon learn about). We mentioned already the two functions `is.na()` and `is.null()`, which provide us with information about the presence of NAs in a vector or whether or not a vector is empty, respectively. We may also be interested in the vector’s type. We can determine this with the `class()` function, or we can ask specifically if a vector is of a certain type with functions, such as `is.factor()`, `is.logical()`, etc. We can find the number of elements

in a vector with `length()`; the first or last few values with `head()` and `tail()`, respectively; and the names of elements with `names()`.

Example 1-2 Finding Details on the Family

Let's use some of these functions to find out more about the family. The length of `fnames`, which contains the names of the family members, is

```
length(fnames)
```

```
[1] 14
```

We can confirm that all of the vectors with family information have length 14 with further calls to `length()`, e.g., `length(fbmi)`.

We may also want summary information about, say, the weights of the family members. We may want to know the smallest and largest values, the average, or the median weight. We can find these with, respectively, `min()`, `max()`, `mean()`, and `median()`. For example the smallest weight in the family is

```
min(fweight)
```

```
[1] 98
```

Some of these functions return a single value, such as the minimum weight just calculated. Others return a value for each element of the input vector. One example is the `names()` function, e.g.,

```
names(fheight)
```

```
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n"
```

The `names()` function returns a character vector the same length as the input vector and each element of the return value is the name of the corresponding element of the input vector. For example, "c" is the name of the third element in `fheight`. Not all vectors have named elements, and when this is the case, the return value is an empty vector, e.g.,

```
names(fweight)
```

```
NULL
```

Often it can be helpful to examine the first few or last few values in a vector to confirm that the data are what we expect. The `head()` and `tail()` functions, respectively, return these initial or final values. By default we are given the first (last) 6 elements, i.e., the return value is a vector of length 6, e.g.,

```
head(fweight)
```

```
[1] 175 125 185 156 98 190
```

However, we can request more or fewer elements by specifying a value for the `n` argument, e.g., to view the last two elements in `fweight` we use `tail(fweight, n = 2)` and the return value is 150 125.

In addition to finding the type of a vector with the `class()` function, e.g.,

```
class(fsex)
```

```
[1] "factor"
```

we can ask whether or not a vector is a particular type. For example,

```
is.factor(fsex)
[1] TRUE
```

Similarly, `is.integer(fbmi)` returns FALSE, but `is.numeric(fbmi)` returns TRUE. The `fage` variable is an integer vector, what do you think the call `is.numeric(fage)` returns? Try it and see. Since an integer is a special case of a number, *R* returns TRUE in this case.

Data Types

Primitive Types

In *R*, *vectors* are the basic or primitive objects. A vector is an *ordered* container of *homogeneous* values. In other words, a vector contains values that are the same type and these values have an ordering. Statisticians analyze measurements of some quantity on a group, and the vector is a convenient structure for this purpose.

The primitive types include:

- `numeric`— real numbers,
- `logical`—TRUE and FALSE only,
- `character`— strings.

Factor Type

Qualitative measurements are an important kind of data, e.g., sex, marital status, and education level, and *R* has a `factor` data type for this purpose, i.e.,

- `factor`— values are stored as integers, and each integer corresponds to a *level*, which is a string. The levels in a factor can be ordered, e.g., education level.

Typically we want to perform different types of calculations with `factor` data, e.g., for a summary, we want tallies of the number of observations at each level, not a mean or median. Many *R* functions, e.g., `summary()`, perform different operations on a vector depending on whether it is `numeric` or `factor`.

1.7 Vectorized Operations

The philosophy in *R* is that operations work on an entire vector. This makes sense given that vectors are the basic data types. A simple example is with subtraction. We can subtract one vector from another element-wise using the `-` operator. For example, if we want to know the difference between the actual weight and desired weight for our family members, then we can simply subtract `fdesiredWt` from `fweight`. That is,

```
fweight - fdesiredWt
[1] 0 10 10 6 -12 40 25 6 10 20 16 4 10 0
```

Here, the 1st element of `fdesiredWt` is subtracted from the 1st element of `fweight` to get 0, the 2nd element of `fdesiredWt` is subtracted from the 2nd element of `fweight` to get 10, and so on.

The notion of vectorized operations is very powerful and convenient. It allows us to express computations at a high-level, indicating what we mean rather than hiding it in a loop.

Example 1-3 Computing BMI

Although we are provided with the BMI of each family member in the `fbmi` variable, we can compute this quantity ourselves from the height and weight of each individual. The formula for BMI is

$$\frac{\text{weight in kg}}{(\text{height in m})^2}.$$

If we use this formula, we need to convert our measurements from pounds into kilograms and from inches into meters. Since there are 2.2 pounds to a kilogram, we can change the units for the values in `fweight` with

```
fweight / 2.2
[1] 79.55 56.82 84.09 70.91 44.55 86.36 84.09 56.36 79.55
[10] 97.73 75.45 63.64 68.18 56.82
```

Similarly, we can convert inches to meters with

```
fheight * 0.0254
      a      b      c      d      e      f      g      h      i      j
1.778 1.626 1.854 1.702 1.549 1.727 1.727 1.651 1.727 1.803
      k      l      m      n
1.702 1.676 1.676 1.575
```

We see that this return vector appears somewhat differently than the return from dividing `fweight` by 2.2. This is because the elements in `fheight` are named so the return value has these names too.

We can combine these calculations into a single calculation of BMI with

```
(fweight / 2.2) / (fheight * 0.0254)^2
      a      b      c      d      e      f      g      h      i      j
25.16 21.50 24.46 24.48 18.56 28.95 28.19 20.68 26.66 30.05
      k      l      m      n
26.05 22.64 24.26 22.91
```

Again, the names of the elements in `fheight` have been carried over to the return value. Also, we can examine `fweight` and `fheight` to check that, e.g., the 3rd weight in `fweight` and the 3rd height in `fheight` are properly combined to create the 3rd BMI in the return value.

For greater clarity, we can create intermediate variables for the converted height and weight with

```
WtKg = fweight / 2.2
HtM = fheight * 0.0254
bmi = WtKg / HtM^2
```

We have assigned our computation of BMI to the variable `bmi`. ■

1.7.1 Aggregator Functions

Some vectorized functions work on all of the elements of a vector, but return only a single value for the result. We have seen examples of these already, others include `mean()`, `min()`, `max()`, `sum()`, `prod()`, and `median()`. For example, the average BMI for the family is `mean(bmi)`, which evaluates to about `24.61`.

1.7.2 Relational Operations

In addition to the arithmetic operators, such as `+` and `*`, R also has relational operators for comparing values. These operators are greater than (`>`), less than (`<`), greater than or equal to (`>=`), less than or equal to (`<=`), not equal to (`!=`), and equal to (`==`). The relational operators are vectorized, meaning that if you give them a vector of length n , they operate on all n elements.

Example 1-4 Compute Who is Over Weight

We can use relational operators to determine which of the family members are over weight. Our definition of over weight is a BMI that exceeds 25. Again, although the variable `foverWt` already contains this information, we can compute it ourselves by comparing `bmi` (`fbmi`) to 25. We do this with

```
overWt = bmi > 25
overWt
```

a	b	c	d	e	f	g	h	i	j
TRUE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	FALSE	TRUE	TRUE
k	l	m	n						
TRUE	FALSE	FALSE	FALSE						

We can compare our vector `overWt` to the one supplied for the family (i.e., `foverWt`) to see if they match. To do this, we can use another relational operator, namely the ‘equal to’ operator (`==`).

```
overWt == foverWt
```

a	b	c	d	e	f	g	h	i	j
TRUE									
k	l	m	n						
TRUE	TRUE	TRUE	TRUE						

Note that this computation checks that the 1st element of `overWt` equals the 1st element of `foverWt`, the 2nd element of `overWt` equals the 2nd element of `foverWt`, and so on. All of these values are TRUE so we have consistent results, i.e., the element of these two vectors have the same values. Rather than examine each of the 14 values to see if they are all TRUE, we can pass the return vector from our comparison to the `all()` function.

```
all(overWt == foverWt)
```

```
[1] TRUE
```

The `all()` function returns TRUE if all of the elements in the input vector are TRUE.

Another helpful function for determining if two objects are the same is the `identical()` function. We call it with

```
identical(overWt, foverWt)
[1] FALSE
```

This is somewhat surprising, given that we just compared the values of each of the elements and found them all to be the same. Can you figure out what is the problem? Notice that our new variable `overWt` is a vector with named elements, but `foverWt` is not. The `identical()` function checks more than the values of the elements. These two vectors are not identical because one has named elements and the other does not.

In addition to the relational operators, many functions in *R* are vectorized. The `nchar()` function is one example. If we give `nchar()` the character vector of family member names, then we get

```
nchar(fnames)
[1] 3 4 3 6 3 3 3 5 3 3 3 3 3 3
```

Here we have a vector of length 14 that contains mostly 3s because most family members' names have only 3 letters in them.

1.7.3 Boolean Algebra

Boolean operators take logical vectors as inputs and perform Boolean algebra. The three common operations are “not”, “or”, and “and”. The “not” operator, which is `!` in *R*, is a unary operator because it has only one input. It turns TRUE into FALSE and vice versa. For example, `!foverWt` returns

```
[1] FALSE  TRUE  TRUE  TRUE  TRUE FALSE FALSE  TRUE FALSE
[10] FALSE FALSE  TRUE  TRUE  TRUE
```

This vector has TRUE for individuals who are not over weight and FALSE for those who are.

The “and” operator, `&`, compares the elements of two vectors and returns a logical vector where TRUE indicates the corresponding elements in the input vectors are both TRUE, and FALSE indicates otherwise. The “or” operator, `|`, compares the elements of two vectors and returns a logical vector where TRUE denotes that either one or the other or both of the corresponding elements in the input vectors are TRUE. Of course, these operations can be combined into compound statements. We provide an example.

Example 1-5 Identifying Certain Family Members

Let's suppose we are interested in identifying female family members who are either over weight or under 45 years old. Then, we can create a logical vector that indicates these characteristics with the following expression:

```
fsex == "f" & (fage < 45 | foverWt)
[1] FALSE  TRUE FALSE FALSE  TRUE  TRUE FALSE FALSE FALSE
[10] FALSE  TRUE FALSE FALSE FALSE
```

We can check the values of `fsex`, `fage` and `foverWt` to confirm that our algebra is correct.

In addition to `all()`, another function that operate on logical vectors and can be quite useful is `any()`. As demonstrated in Q.1-4 (page 17), the `all()` function returns TRUE if all of the elements of the logical vector are TRUE. The `any()` function returns TRUE if any of the elements are TRUE. For example, since some of the elements in `foverWt` are TRUE and some are FALSE, we find `any(foverWt)` returns TRUE, and `all(foverWt)` is FALSE.

1.7.4 Coercion

At times, we may want to change the type of a vector, e.g., from logical to numeric so that the vector contains 1s and 0s rather than TRUEs and FALSEs. We can do this with the collection of “as.” functions, e.g., `as.numeric()` attempts to convert the input vector into a numeric vector. We try it with the logical vector `foverWt`

```
as.numeric(foverWt)
[1] 1 0 0 0 0 1 1 0 1 1 1 0 0 0
```

As expected, a value of TRUE is converted to 1 and FALSE to 0. When we try to convert `fnames`, which is a character vector of the family member names, we get:

```
as.numeric(fnames)
[1] NA NA
Warning message:
NAs introduced by coercion
```

In this case, the conversion results in a vector of NA values. Notice also that a warning message is issued to let us know that we have missing values in our result. *R* can convert some character values into numbers, e.g.,

```
as.numeric(" -17.01")
[1] -17.01
```

However, when a string is not made up of digits (and a possible period and negative sign), then the conversion results in NA.

The reverse coercion, i.e., converting a number into a character string, works as expected, with a few subtleties as shown in the next example.

Example 1-6 How Many Digits are in `fbmi`?

We can convert the numeric vector, `fbmi`, into a character vector with

```
as.character(fbmi)
[1] "25.1623879871136" "21.5010639538325" ...
```

These first two elements look a bit different from what we have seen before. When we print `fbmi` at the console, we see

```
head(fbmi)
[1] 25.16 21.50 24.46 24.48 18.51 28.95
```

It appears that the first two values of `fbmi` are 25.16 and 21.50, not 25.1623879871136 and 21.5010639538325.

This is simply a matter of controlling the number of digits printed to the console. If we want more digits printed for `fbmi`, then we can explicitly specify this with the `print()` function, e.g.,

```
print(fbmi, digits = 22)
[1] 25.16238798711363244820 21.50106395383245327935 ...
```

Now we have more digits than in the character strings! We discuss the topic of how numbers are represented in *R* in Chapter 10. For now, we simply recognize that what prints to the console may be different from the actual values in the vector.

We also mention that if we want to change the default number of digits printed for all subsequent computations in our *R* session, then we can do this through the `options()` function, e.g.,

```
options(digits = 12)
fbmi
[1] 25.1623879871 21.5010639538 ...
```

This example shows us that we should be careful when reading and comparing numeric values because what is printed at the console is not necessarily the actual value in a vector. ■

Converting Factors

Factors are a special data type and coercion of a factor has special behaviors. Recall that a factor consists of integer values where each integer represents a level, and a level is associated with a character string. When we coerce a factor into a number, we get the integer value of the level. However, when we coerce a factor into a character string, then the return value is the label for the level. We demonstrate with `fsex`:

```
as.numeric(fsex)
[1] 1 2 1 1 2 2 1 2 1 1 2 1 1 2

as.character(fsex)
[1] "m" "f" "m" "m" "f" "f" "m" "f" "m" "m" "f" "m" "m" "m" "f"
```

When we performed the relational operation, `fsex == "f"` in Q.1-5 (page 18), *R* implicitly converted `fsex` to a character vector before performing the comparison.

Implicit Coercion

Implicit coercion occurs when we operate on a vector in a way that is not intended for its type. For example, if we add 1 to a logical vector, then the logical values are converted to 0s and 1s implicitly, and 1 is added to each element.

```
1 + foverWt
[1] 2 1 1 1 1 2 2 1 2 2 2 1 1 1
```

As another example, when we use the logical operator `>` to compare the values in `fweight` to "150", *R* determines that it can convert "150" to a numeric value; that is, `fweight > '150'` produces

```
[1] TRUE FALSE TRUE TRUE FALSE ...
```

In general, with the exception of adding 0 or 1 to a logical vector to convert it to numeric, it's best to avoid implicit coercion. It can produce nonsensical and unexpected results, if we don't understand well enough how coercion works. For example, can you figure out why the following comparison yields all FALSEes?

```
fweight > "abc"
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE ...
```

To help you, examine the built-in vector `letters` and see if you can make sense of the comparison, `letters > "c"`.

1.8 Data Frames

We have used information about a 14-member family to demonstrate several features of the *R* language. This information appears in several vectors, each with 14 elements. It's natural for us to picture this information arranged in a table, where each column of the table corresponds to a variable, like name, age, height and weight, and each row corresponds to a particular individual, such as the 77-year old Tom in the first row and the 27 year-old Sue in the 5th row. *R* provides a data structure, called a *data frame*, for collecting vectors into one object, which we can imagine as a table. More specifically, a data frame is an ordered collection of vectors, where the vectors must all be the same length but can be different types.

Example 1-7 A Data Frame of Measurements on a Family

The workspace that we loaded into our *R* session in Q.1-1 (page 12) contains a data frame for our 14-member family. This data frame is called `family`. We can use `head()` to examine the first few rows of `family` with

```
head(family)

  firstName sex age height weight   bmi overWt
1      Tom   m  77     70    175 25.16   TRUE
2     Maya   f  33     64    125 21.50  FALSE
3      Joe   m  79     73    185 24.46  FALSE
4   Robert   m  47     67    156 24.48  FALSE
5      Sue   f  27     61     98 18.51  FALSE
6      Liz   f  33     68    190 28.95   TRUE
```

The names of the variables in the data frame are not exactly the same as the names of the individual vectors, but we can see that these data are the same as in the separate vectors, e.g., `fbmi` and the column labeled `bmi` have the same values in the same order. These vectors are heterogenous in type. That is, `firstName` is a character vector, `sex` a factor, `age` is integer, `bmi` is numeric, and `overWt` is logical. The ordering of the vectors in the data frame is `firstName`, `sex`, `age`, and so on.

The types of some of the vectors are not immediately apparent from examining the head of the data frame. For example `firstName` and `sex` look similar in type. We can try to confirm this with a call to the `class()` function:

```
class(family)
[1] "data.frame"
```

We did not get what we expected, i.e., the data types of all the variables in `family`. Calling `class()` with `family` returns the type of the `family` object, not the vectors that it contains. We figure out shortly how to get the data types of the vectors in `family`.

We can find out additional information about the data frame with some of the same functions that we used to find out information about vectors. For example, `length()` returns the number of vectors in the data frame, `names()` gives us the names of the vectors. Additionally, `dim()` provides the number of rows and columns in the data frame. ■

Dollar-sign Notation

To access the vectors within a data frame we can use the \$-notation. For example, we find the class of `firstName` and `sex` in `family` with

```
class(family$firstName)
[1] "character"

class(family$sex)
[1] "factor"
```

Now we have the answer that alluded us in Q.1-7 (page 21).

Example 1-8 Exploring the `family` Data

We can use the \$-notation to pass a vector in the `family` data frame as input to any of the functions we have introduced that accept vectors as inputs. Of course, our family data is artificial so we can not make too much of what we find in our exploration, but it gives an idea as to what is possible.

We can find the average height and weight of the family members with, respectively,

```
mean(family$height)
```

```
[1] 67.86
```

```
mean(family$weight)
```

```
[1] 157.8
```

Alternatively, the `summary()` function accepts a data frame as input and provides summary statistics for each variable. We call `summary()` with

```
summary(family)
```

	firstName	sex	age	height
Length:	14	m:8	Min. :24.0	Min. :61.0
Class :	character	f:6	1st Qu.:33.0	1st Qu.:65.2
Mode :	character		Median :47.5	Median :67.0
			Mean :48.1	Mean :66.9
			3rd Qu.:58.0	3rd Qu.:68.0
			Max. :79.0	Max. :73.0
	weight	bmi	overWt	
Min. :	98	Min. :18.5	Mode :logical	
1st Qu.:	129	1st Qu.:22.7	FALSE:8	
Median :	161	Median :24.5	TRUE :6	
Mean :	158	Mean :24.6	NA's :0	
3rd Qu.:	182	3rd Qu.:26.5		
Max. :	215	Max. :30.0		

Notice that `summary()` does not provide the same summary statistics for all the vectors in `family`. The factor, `sex`, is summarized with counts of the number of elements in each level, and the same type of summary is provided for the logical `overWt`. The character vector `firstName` is summarized only by its length. The summary of integer and numeric variables include mean and median, minimum and maximum, and upper and lower quartiles.

Lastly, we can also make a scatter plot of height and weight with

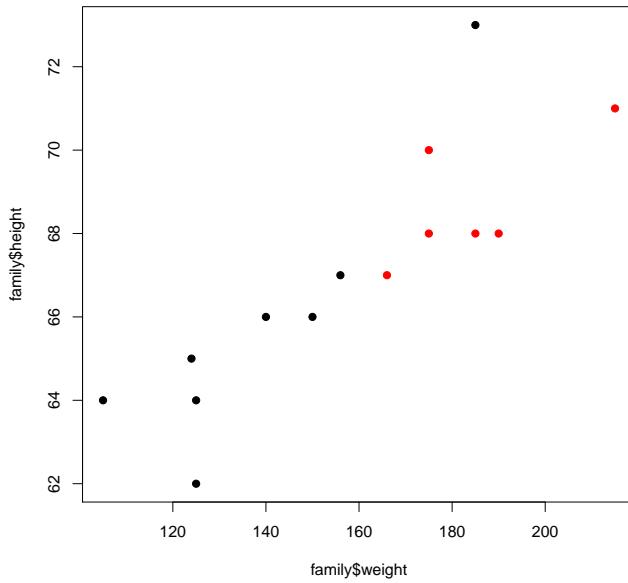


Figure 1.1: Family Heights and Weights. *This scatter plot shows the heights and weights of the individuals in the artificial family that we created for code demonstration. The red points correspond to the over weight family members.*

```
plot(y = family$height, x = family$weight,
      pch = 19, col = 1 + family$overWt)
```

Notice that we coerced `overWt` into a numeric vector with values 1 and 2 in order to use these values to specify colors for the points (the 2s are the red points). The resulting plot appears in Figure 1.1. ■

Data Frames

Data from a study or an experiment often consist of different types of measurements on a set of subjects of experimental units. For example, the World Bank provides summary statistics on countries around the world. Here the unit is the country and the measurements include gross domestic product, life expectancy, and literacy rate. These data are naturally organized into a table format. In *R*, the data frame is a data structure designed for the purpose of working with these kinds of data.

Rows

Each row in a data frame corresponds to a subject in the study, unit in an experiment, etc. The various measurements on a subject are in one row of the data frame.

Columns

The columns in a data frame correspond to variables. These can be different types. For example, a health study may contain height (numeric), sex (factor), education level (ordered factor), and a unique identifier (character) for all subjects.

Working with Data Frames

The data frame facilitates many kinds of statistical analyses. For example, we can easily examine the relationship between income and sex using the formula `income ~ sex` in a call to the `plot()` function. And this same formula can be used to compare the average income levels for the two sexes via the `lm()` function. In both cases, the function performs a computation on the columns in a data frame. Depending on the data types, a different plot is created or a different model is fitted.

1.9 Subsets

A lot of what we do in statistics and exploratory data analysis is to examine subgroups of a sample or population. We determine characteristics about that subset and compare them to other groups or to the overall group. We might look at the age of over weight individuals and compare these values to the ages of all members of the group. Or, we may want to compare the BMI for men and women. These are all examples of how we look at different parts of our data using categorical or continuous variables to “zoom in” on a subgroup.

Being able to compute subgroups easily within our data is a very powerful and flexible feature of *R*, but takes some getting used to. There are essentially 5 different ways to compute a subgroup in *R*. They all use the `[` operator, and the only differences are what you specify as the value to use to identify the particular subset of interest.

1.9.1 Subsetting with Logical Vectors

One of the most intuitive approaches to subsetting uses a logical vector that is the same length as the vector from which we want to compute a subgroup. The TRUE values in this logical vector appear in the positions that correspond to the elements in the original vector that we want in our subset, and the FALSE values indicate those elements to be dropped. Figure 1.2 provides a diagram of the concept behind subsetting with a logical vector.

Example 1-9 Finding the Ages of the Over Weight Family Members

To find the ages of the over weight family members, we use the `foverWt` vector to identify the elements in `fage` that we want.

```
fage[foverWt]
[1] 77 33 67 59 27 55
```

If we are only after the average age of these individuals then we can pass the return vector of the 6 ages to the mean function with

```
mean(fage[foverWt])
[1] 53
```

On the other hand, if we want to perform additional calculations on this subgroup then we can assign the subset to a variable, e.g.,

```
ageOfOverWt = fage[foverWt]
```

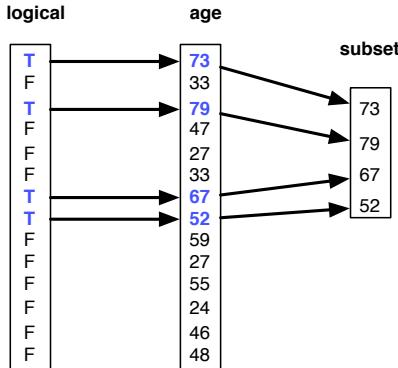


Figure 1.2: Subsetting with Logicals. This diagram illustrates the concept of subsetting by *logicals*. The logical vector on the left has the same length as the vector `age` from which we want to compute a subgroup. The TRUE values in the logical vector identify the elements in `age` that we want to keep. The logical vector identifies the 1st, 3rd, 7th, and 8th elements in `age` for the subset. The resulting subset is on the righthand side of the diagram. Its length matches the number of TRUE values in the logical. (Note that for brevity we use shorthand of T for TRUE and F for FALSE.)

Then we can continue to work with the subsetted ages that are now in `ageOfOverWt`. Reciprocally, to obtain a subset of those who are not over weight, we use the boolean not operator to turn the TRUE values into FALSE and vice versa. We do this with `fage[!foverWt]`.

Example 1-10 Finding the Heights of the Females in the Family

As another example, to examine the heights of the females in the family, we can create a logical vector that indicates which family members are female and use it to subset `fheight`. We use the relational operator `==` to create a vector of TRUEs for females and FALSE for males, with

```
fsex == "f"
[1] FALSE  TRUE FALSE FALSE  TRUE  TRUE FALSE  TRUE FALSE
[10] FALSE TRUE FALSE FALSE  TRUE
```

Then we use this logical vector to index into `fheight` with

```
fheight[fsex == "f"]
[1] 64 61 68 65 67 62
```

The `subset()` function performs subsetting by logicals without the use of the `[` operator. It takes two inputs: the object to be subsetted and the logical expression that indicates the elements to keep. In other words, these two examples of computing a subset of the ages of over weight people in Q.1-9 (page 24) and the heights of the females in Q.1-10 (page 25) can be re-expressed using `subset()` as follows:

```
subset(fage, foverWt)
subset(fheight, fsex == "f")
```

This function can be very handy, but it does not support the other forms of subsetting, which we describe next, nor subsetting other data structures, such as lists.

1.9.2 Subsetting by Position

If we know that we want the BMI of the 10th person in the family, then we can use this position to specify the subset, i.e.,

```
f bmi[10]
```

```
[1] 30.05
```

More generally, we can provide a vector of positions to use in subsetting. These positions identify the elements in the original vector to appear in the subset. The order of the positions determines the order of the values in the return vector. If a position appears multiple times then the corresponding element appears multiple times in the subset. Figure 1.3 provides two conceptual diagrams of subsetting by position.

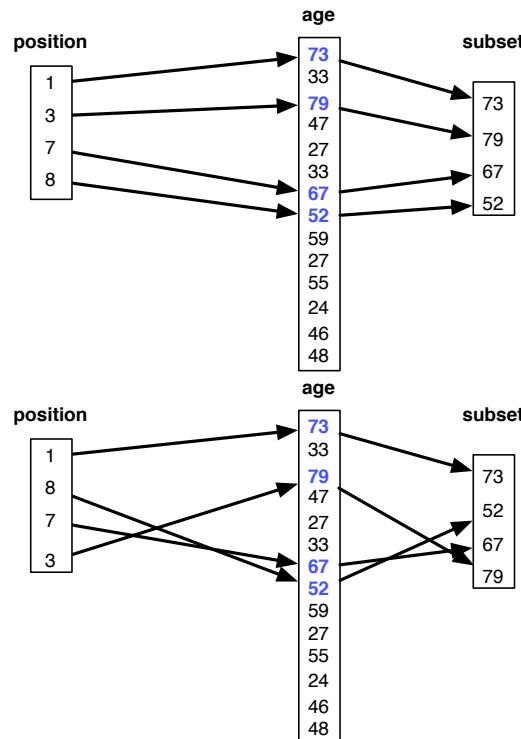


Figure 1.3: Subsetting by Position. These two diagrams show how subsetting by position works. The vector of integers on the lefthand side of each diagram identifies the positions of the values in the `age` vector that we want in the subset. As in Figure 1.2, we have identified the 1st, 3rd, 7th, and 8th elements for our subset. Notice that the position vector in the bottom diagram has these same values except they appear in a different order and one value appears more than once. The order determines the ordering of the subset, e.g., the value 8 is the 2nd and 5th elements in the positions vector so the 8th element of `age` is placed in the 2nd and 5th elements of the subset.

Example 1-11 Finding a Subset of BMI by Position

Suppose we want the 1st and 14th BMI values in our subset. We can create a vector with two elements, 1 and 14, with the expression `c(1, 14)`, where `c()` stands for concatenate (more on this function in Section 1.11.1). Then we compute our subset with

```
fbmi[ c(1, 14) ]
```

```
[1] 25.16 22.91
```

If we swap the order of 1 and 14 in our vector of positions, then the 14th BMI value in the original vector is the first value in the subset and the 1st BMI value in `fbmi` is the 2nd element of the subset. Furthermore, if we include 1 twice in the vector of positions, then the BMI for the first person appears twice in the subset. That is,

```
fbmi[ c(14, 1, 1) ]
```

```
[1] 22.91 25.16 25.16
```

This may seem a bit unusual, but it can be very useful, particularly for coloring points in plots (see Section 3.5). ■

What if we give a position that makes no sense, e.g., that is larger than the length of the starting vector? For example, we know there are only 14 members in our family so let's ask for the 20th element of one of the vectors. When we do this, we find, e.g.,

```
fage[20]
```

```
[1] NA
```

The result is a missing value, `NA`. This makes sense in many contexts. It is something we should be aware of so that we can understand how NAs might be introduced into our computations.

There are two other values that might be considered meaningless. What if we ask for the 0-th element of a vector? For example,

```
fage[0]
```

```
integer(0)
```

```
fage[c(0, 2)]
```

```
[1] 33
```

Essentially, *R* ignores a request for the 0-th element and doesn't include a value in the result for that element. This means that the result may not have as many elements as we asked for. That is, in our second expression above, we requested 2 elements of `fage` and were given only 1. Can you figure out what is the return value from `fage[c(20, 0, 2)]`? Try to reason it out from the previous examples. It makes sense that the return value has 2 elements, not 3, and these are `NA 33`, in that order.

Now, what if we ask for a negative index? That is subsetting by exclusion.

1.9.3 Subsetting by Exclusion

If the positions that we supply are negative numbers, then *R* drops those particular elements of the vector to create the subset (see Figure 1.4). For example, we drop the 1st through 3rd elements of `fage` to compute a subset from the 4th through 14th elements with

```
fage[ c(-1, -2, -3) ]
```

```
[1] 47 27 33 67 52 59 27 55 24 46 48
```

We cannot mix positive and negative indices in a single subsetting call. In other words, we cannot include some elements and omit others in one action. The following request is not valid:

```
fage[c(-1, -3, 5, 6, 7)]
```

It might seem reasonable to drop the 1st and 3rd elements and include the 5th, 6th and 7th. However, if we give such a command, we get an error,

```
fage[c(-1, -3, 5, 6, 7)]
```

```
Error in fage[c(-1, -3, 5, 6, 7)] :
  only 0's may be mixed with negative subscripts
```

This mix of positive and negative positions does not make sense because we are saying that we want to only include elements in positions 5, 6, and 7, but also we want to exclude those in positions 1 and 3. What about the 2nd, 4th, 8th, ..., 14th elements? Are they part of the inclusion or the exclusion? For this reason, *R* accepts either all nonnegative or nonpositive positions.

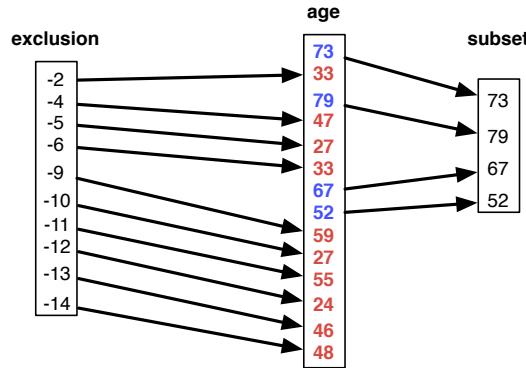


Figure 1.4: Subsetting by Exclusion. This diagram demonstrates the idea of subsetting by exclusion. The vector of negative integers on the left identifies the positions of the values in `age` that we want excluded from the subset. According to these negative indices, all but the 1st, 3rd, 7th, and 8th elements are excluded from `age` to create the subset. This yields the same subset as in Figure 1.2 and in the top diagram in Figure 1.3.

1.9.4 Subsetting by Name

We have seen that vector elements can have names. If we subset a vector where the elements are named, then we can refer to the elements we want in the subset using these names (see Figure 1.5). Recall that `fheight` has named elements, where the names are the letters "a" through "n". We can compute a subset of the elements named "c" and "a", in that order, with

```
fheight[ c("c", "a") ]
```

```
c   a  
73  70
```

As with subsetting by position, if we ask for a non-existent element, then we get an NA in the result, e.g.,

```
fheight[ c("c", "z", "a") ]
```

```
c <NA>      a  
73     NA    70
```

Notice that in the resulting subset both the 2nd element and its name are NA.

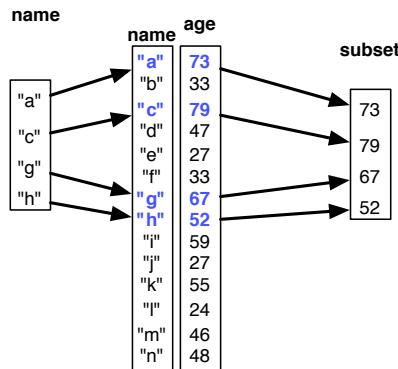


Figure 1.5: Subsetting by Name. This diagram illustrates the idea of subsetting by name. The `age` vector in this diagram has names associated with the elements. The character vector on the lefthand side of the diagram contains the strings, "a", "c", "g", and "h". These identify the elements in `age` that we want in the subset. Note that the resulting vector is also a named vector, but we do not include these names for simplicity.

We cannot use this style of subsetting to exclude elements. Think about what `fheight[-c("c", "z", "a")]` means when R interprets the command. While we can understand that we mean to drop the elements named "c", "z" and "a", R first evaluates `-c("c", "z", "a")`. This is meaningless as the negative of a string doesn't make sense and can't be coerced into values that do make sense. The error R gives us comes from this part of the computation.

```
fheight[ -c("c", "z", "a") ]
```

```
Error in -c("c", "z", "a") : invalid argument to unary operator
```

What's the unary operator? It is the `-` operator.

1.9.5 Subsetting All

The last kind of subsetting occurs when we pass no value for the indexing vector, e.g., `fage[]`. The result is the `fage` vector itself, i.e, all of the elements in their original order. This is not the same as passing in a vector with length 0, e.g., `fbmi[integer(0)]`. That returns a subset with the same length as the indexing vector, which is an empty vector,

```
fbmi[ integer(0) ]  
numeric(0)
```

Why is this kind of subsetting useful? There are several reasons. Something that we have not mentioned about subsetting yet is that not only can we access sub-vectors using these 5 techniques, but we can also modify the contents of these sub-vectors in the original vector. We show how to do this next in Section 1.9.6. Another time when “subsetting all” is useful is when we work with data frames and multi-dimensional vectors, such as matrices. In these cases, we must provide both the row and column indices so subsetting all rows (or columns) can be quite useful. We provide examples of this in Section 1.9.7.

1.9.6 Assigning Values to Subsets

We can use the 5 subsetting techniques to assign new values to a subgroup of the elements in the original vector. In this case, we place the subsetting specification on the lefthand side of the equal sign and the new values on the righthand side. For example,

```
fweight[ 10 ] = NA
```

This assignment sets the weight of the 10th element in `fweight` to NA.

Example 1-12 Re-assigning Age Values

Suppose we made a mistake and erroneously switched the ages of the 1st and 10th individuals, i.e., the two Toms in the family have each other’s age. Since we know that the 1st Tom is listed as 77 years old and the 2nd Tom as 27 we can swap these ages with the following assignment:

```
fage[ c(1, 10) ] = c(27, 77)
```

Alternatively, we don’t need to type the ages. We can simply get the ages via subsetting and then assign them, i.e.,

```
fage[ c(1, 10) ] = fage[ c(10, 1) ]  
fage
```

```
[1] 27 33 79 47 27 33 67 52 59 77 55 24 46 48
```

An even more general approach that uses the names of the individuals and does not require us to know their positions in `fage` is to use subsetting with logicals on the lefthand side. We do this with

```
fage[fnames == "Tom"] = rev(fage[fnames == "Tom"])
```

Let’s examine this assignment statement more closely. Breaking it down, we see the expression `fnames == "Tom"`, which returns a logical vector where the 1st and 10th elements are TRUE and the rest are FALSE. Then `fage[fnames == "tom"]` computes a subset of `fage` from its 1st and 10th values. Now the equal sign indicates that we are replacing

these values in `fage` with the values provided on the righthand side of the equal sign. The expression within the `rev()` function call on the right returns the two element vector `77 27`. When this vector is passed to the `rev()` function, the order of the elements is reversed so `27 77` is assigned into the 1st and 10th elements of `fage`.

We provide a fourth approach that demonstrates the `which()` function. This function returns the positions of the TRUE elements in a logical vector. This means that `which(~fnames == "Tom")` returns the vector with elements `1 10`. Then we can use these values to modify `fage` with

```
pos = which(fnames == "Tom")
fage[ pos ] = fage[ rev(pos) ]
```

Hopefully this example helps reveal how powerful subsetting can be in R! ■

We can use this form of assignment with the “subsetting all” technique. Suppose we want to change everyone’s first name to “X”, we can do this by subsetting all elements of `fnames` and assigning them to “X” as follows:

```
fnames[ ] = "X"
fnames
[1] "X" "X"
```

This assignment is very different from: `fnames = "X"`. If we try it, then we find that `fnames` is now a vector with only one element (“X”). In the first assignment, we assigned “X” to all elements of the vector `fnames`. In the second assignment, we assigned the value “X” to the name “fnames”. In other words, we re-assigned `fnames` to a new value, which is a character vector of length 1.

1.9.7 Subsetting Data Frames

The 5 types of subsetting we introduced for vectors also work on data frames. The main difference is that we must specify how to subset both dimensions, i.e., rows and columns. For example, we can subset by position the first 4 rows and the first two columns of `family` to create a 4 by 2 data frame with

```
family[1:4 , 1:2]
```

	firstName	sex
1	Tom	m
2	Maya	f
3	Joe	m
4	Robert	m

When we compute a subset of a data frame, the row and column position vectors are separated by a comma. Note that the expression `1:4` returns the vector `1 2 3 4` (we provide more information on the `:` operator in Section 1.11.2).

Here, we used subsetting by position on both the rows and columns, but that is not necessary. For example, we can modify the column specification and compute the subset by exclusion. To do this, we can use the expression `-(3:14)`, which returns the following negative integers:

```
[1] -3 -4 -5 -6 -7 -8 -9 -10 -11 -12 -13 -14
```

Then, `family[1:4, -(3:14)]` yields the same subset as `family[1:4, 1:2]`. It's typically a better idea to specify the columns that we want by name so that we are not dependent on the particular order that the variables appear in the data frame. Since the first two columns are `firstName` and `sex`, we can get the same subset with subsetting by name on the column dimension of the data frame, e.g.,

```
family[1:4, c("firstName", "sex")]
```

	firstName	sex
1	Tom	m
2	Maya	f
3	Joe	m
4	Robert	m

For data frames, we often want some of the rows and all of the columns or all of the rows and some of the columns. That is, we use “subsetting all” on one of the dimensions.

Example 1-13 Creating a Subset of the Over-Weight Family Members

To obtain a data frame containing all of the variables (columns) for the over-weight individuals in the family, we use `family[family$overWt,]`. The return value is

	firstName	sex	age	height	weight	bmi	overWt
1	Tom	m	77	70	175	25.16	TRUE
6	Liz	f	33	68	190	28.95	TRUE
7	Jon	m	67	68	185	28.19	TRUE
9	Tim	m	59	68	175	26.66	TRUE
10	Tom	m	27	71	215	30.05	TRUE
11	Ann	f	55	67	166	26.05	TRUE

By not providing a vector after the comma in the square brackets, we compute the subset of all columns. Also, we have computed a subgroup of the rows using a logical vector (with the dollar-sign notation).

As another example, to subset all rows and some columns, we can, e.g., use the following expression, `family[, c("sex", "firstName")]`. This returns a data frame of `sex` and `firstName`, in that order, of all 14 family members. Furthermore, since a data frame is considered an ordered collection of vectors, we can eliminate the specification of the row subsetting, i.e., `family[c("sex", "firstName")]` produces the same subset. ■

The `subset()` function also accepts a data frame as its first input. The logical vector that we provide as the second argument to the function is used to subset the rows of the data frame. Implicitly, the `subset()` function computes subsets of all the columns for a data frame. In our previous example, we can use the expression `subset(family, overWt)` to obtain the same subset as in Q.1-13 (page 32). Notice that in this case, we do not need to use the \$-notation to specify the column in `family`.

1.10 Applying Functions to Variables in a Data Frame

Often, we want to apply a function to each variable in a data frame. For example, we may want to know each vector's type. We have seen already that we can find one vector's type with, e.g.,

```
class(family$sex)
[1] "factor"
```

However, it is cumbersome to call `class()` 7 times for each of the variables (and we also need to know all of the variable names). Instead, we can ask *R* to apply the `class()` function to each of the vectors in `family` with

```
sapply(family, class)
firstName          sex         age      height      weight
"character"     "factor"    "integer"   "numeric"   "integer"
bmi            overWt
"numeric"      "logical"
```

The `sapply()` function can be very useful. This function also accepts additional arguments which it passes on to the function being called for each variable.

Example 1-14 Quantiles of the Family Measurements

We can find the median for each of the variables in `family` with,

```
sapply(family, median)
```

```
Error in median.default(X[[2L]], ...) : need numeric data
In addition: Warning message:
In mean.default(sort(x, partial = half + 0L:1L)[half + 0L:1L]) :
  argument is not numeric or logical: returning NA
```

Unfortunately, since `firstName` and `sex` are character and factor vectors, respectively, the `median()` function returns an error indicating that it takes only numeric inputs. If we want to avoid taking the median of these two variables, we can apply the `median()` function to all but the first two variables by excluding them with

```
sapply(family[-(1:2)], median)
age height weight      bmi overWt
47.50  67.00 161.00  24.47   0.00
```

We have used subsetting by exclusion to drop `firstName` and `sex`, and then we apply the `median()` function to the remaining variables in the data frame. Also notice that the logical vector, `overWt` has been converted to numeric by the `median()` function.

Suppose that we want the lower quartile of these vectors, rather than the median. We can find the lower quartile for `height` with `quantile(family$height, probs = 0.25)`. In addition to supplying the input `family$height` to the `quantile()` function, we also provide the particular quantile via the `probs` argument. In order to find the lower quartile of all the numeric vectors in `family`, we can use `sapply()`, but we need to also provide the `probs` argument to `quantile()`. We can do this with

```
sapply(family[-(1:2)], quantile, probs = 0.25)
age.25% height.25% weight.25%      bmi.25% overWt.25%
33.00       65.25      128.75      22.71       0.00
```

R offers several apply functions for working with data frames and other data structures. We only describe `sapply()` here. Others, such as `lapply()`, `tapply()`, and `apply()` are discussed in Chapter 4.

1.11 Creating Vectors

We have already used a few of *R*'s functions for creating vectors when we demonstrated how to subset by position and exclusion. In those examples, we saw that we can concatenate values together into a vector with the `c()` function and we can create simple sequences of integers with the `:` operator. In this section, we describe these functions in greater detail and introduce other, richer functions for creating vectors.

1.11.1 Concatenating Values into a Vector

The notion of concatenation was first introduced in Q.1-11 (page 26). There we saw how to use the `c()` function to take one or more values and put them into a new vector. These values do not need to be integers, or even numbers. For example,

```
y = c(1.2, 4.5, 3.2)
y

[1] 1.2 4.5 3.2

c(TRUE, FALSE, FALSE, TRUE)

[1] TRUE FALSE FALSE TRUE

c("Tim", "Jessica", "Jo", "Isabella")

[1] "Tim"      "Jessica"   "Jo"        "Isabella"
```

Additionally, if we want to name the elements in the vector, we can do this with, e.g.,

```
c(bob = TRUE, x = FALSE, mm = TRUE)

bob      x      mm
TRUE FALSE  TRUE
```

When we concatenate integers into a vector do we obtain an integer vector in return? Well, in *R*, all numbers that we type at the console are made into real numbers. This means that when we type `c(1, 0, 200)`, we get a numeric vector because the individual values are treated as numeric. However, if we place an `L` after each integer, then an integer vector is created. For example,

```
x = c(1L, 0L, 200L)
class(x)

[1] "integer"
```

The `c()` function can be used to concatenate vectors, e.g., `z = c(y, x)` yields the numeric vector `1.2 4.5 3.2 1.0 0.0 200.0`. In this example, the integer vector `x` is converted to numeric in the concatenation. More on the topic of conversion/coercion in Section 1.11.5.

1.11.2 Creating Sequences of Values

The `:` operator is a built-in syntax for creating an integer sequence, e.g.,

```
3:5
[1] 3 4 5
```

```
class(3:5)
[1] "integer"
```

The sequence can include negative integers and it can decrease rather than increase. For example,

```
-6:2
[1] -6 -5 -4 -3 -2 -1 0 1 2
```

The `:` operator can also be used to create a sequence of 1-apart numerics, e.g., the expression `1.7:6.5` returns the numeric vector `1.7 2.7 3.7 4.7 5.7`. Notice that 5.7 is the largest value in the form $1.7 + n$ that is less than or equal to 6.5.

The `:` operator is a very specific and simple version of the more general `seq()` function. With `seq()`, we can create sequences with strides other than 1. The arguments to the `seq()` function include `from`, `to`, `by` and `length.out`. They are not all required; all we need to provide is enough information to uniquely specify the sequence. This can be: `from`, `to`, and `length.out`; `from`, `length.out`, and `by`; `from`, `to`, and `by`; or `to`, `length.out`, and `by`. Below are examples of all of these possibilities:

```
seq(from = 1, to = 6, by= 2)
[1] 1 3 5

seq(from = 1, to = 6, length.out = 3)
[1] 1.0 3.5 6.0

seq(from = 1, by = 2, length.out = 3)
[1] 1 3 5

seq(to = 6, by = 2, length.out = 3)
[1] 2 4 6
```

We do not get the same sequence for each function call, even though we have used the same values for the arguments. That is, we have called `seq()` with all 4 possible subsets of 3 arguments from `from = 1, to = 6, length.out = 3`, and `by = 2`. The reason for this is that the `to` argument is taken to be the end value of the sequence when accompanied with the `length.out` argument. That is, with `from = 1, to = 6`, and `length.out = 3`, in order for the three elements to be equi-spaced and the beginning and end of the sequence to be 1 and 6, the middle element must be 3.5. Likewise, with `to = 6, length.out = 3`, and `by = 2`, the last element is 6 and working back by 2s, we get 2, 4, and 6 as our sequence. When we supply all 4 arguments, we get an error. Try it and see what the error message is. Does it make sense?

1.11.3 Functions for Operating on Vectors

One function that can be very useful in creating vectors is `rep()`. Short for repeat, this function repeats the elements of a vector to create a new vector. The two arguments `times` and `each` perform this repetition differently – one repeats the individual elements and the other repeats the entire sequence. An example makes this clear. Recall that we earlier created a vector `x` with values `1 0 200` so

```
rep(x, times = 2)

[1] 1 0 200 1 0 200

rep(x, each = 2)

[1] 1 1 0 0 200 200
```

We can also supply a vector to the `times` argument that is the same length as `x`, where the elements in this vector indicate the number of times to repeat each element in `x`. For example, `rep(x, times = c(2, 0, 3))` returns a vector of length 5 with values `1 1 200 200 200`.

Other functions that offer general facilities for operating on vectors include `sort()`, `rev()`, and `order()`. These functions perform the computations that their names suggest. The `sort()` function sorts the elements of a vector into ascending or descending order, and `rev()` reverses the order of the elements. For example, recall `z` contains `1.2 4.5 3.2 1.0 0.0 200.0`, in that order, then

```
sort(z)

[1] 0.0 1.0 1.2 3.2 4.5 200.0

rev(z)

[1] 200.0 0.0 1.0 3.2 4.5 1.2
```

The `order()` function returns a vector of positions. These positions can be used to re-order elements to obtain, e.g., the 2 smallest values in `x`.

```
order(z)

[1] 5 4 1 3 2 6

z[ order(z) [1:2] ]

[1] 0 1
```

Let's examine this last expression carefully. We know that the return value from `order(z)` is the vector `5 4 1 3 2 6`. This indicates that the smallest value in `z` is the 5th element, the next smallest is the 4th element, and so on. When we subset this return value, i.e., this vector of positions, with `1:2`, we obtain the first two elements, `5 4`. We use these to subset `z` by position to get the two smallest values in `z`. Of course, we could also call `sort(z) [1:2]` to obtain the same results.

1.11.4 Functions for Manipulating Character Vectors

For character vectors, the `paste` function is convenient for combining strings together, e.g., we can create one long string of all the names in `fnames` with

```
paste(fnames, collapse = "*")
[1] "Tom*Maya*Joe*Robert*Sue*Liz*Jon*Sally*Tim*Tom*Ann*Dan*Art*Zoe"
```

Note that the `collapse` parameter is used to specify what, if any, character(s) to place between the strings being pasted together.

The function `strsplit`) splits strings by user-specified delimiters. For example, we can split the names in `fnames` on the letter "o", which for the first two names of "Tom" and "Maya" returns

```
strsplit(fnames[1:2], "o")
[[1]]
[1] "T" "m"

[[2]]
[1] "Maya"
```

We see that "Tom" is split into two pieces, the "T" before the o and the "m" after it. Whereas, "Maya" is not split at all because the string does not contain an o. If we split the name "Yamamoto" on "o" then we would get three strings, "Yam" "m" "t".

The `substring`) function can extract a piece of a string, e.g.,

```
substr("Yamamoto", start = 3, stop = 7)
[1] "momot"
```

In addition, we can match and substitute text in strings using regular expressions. These capabilities are available in the functions `grep()`, `gsub()` and others. See Chapter 8 for more details on string manipulation and regular expressions.

1.11.5 Creating Vectors from Different Types

We have emphasized that vectors must contain the same type of elements. If we try to combine different types of elements, *R* coerces them to an appropriate common type. That is, we cannot use a vector to store values of different types, such as number and strings. However, we can concatenate numbers and strings together into a vector, and when we do, the numbers are converted to strings so that we have a character vector. For example,

```
c("Hi", -2, "Bye", 10.3, 0)
[1] "Hi"   "-2"   "Bye"  "10.3" "0"
```

Below are a few other examples of how *R* coerces values of different types into a single type:

```
c(1, 2, 3, TRUE)
[1] 1 2 3 1

c(1L, 2L, 3L, 10.7)
[1] 1.0 2.0 3.0 10.7
```

We see that when combining numbers and logicals, the logicals are coerced to 0s and 1s, and when concatenating integers and numerics, the integers are converted to numeric. You can try combining different elements and see what you get. For example, what do you think happens when you combine strings and logicals? And, how are the values coerced when we concatenate strings, logicals, and numeric values? Try it and see.

1.12 Recycling Rules

What happens when we add two vectors with different lengths? We have seen already the results for computations such as `2 + c(1, 2, 7)`. However, you probably didn't realize that we were adding two vectors of different lengths. That is, the 2 in this expression is a vector of length 1. Why? Recall that the primitive object in *R* is a vector so the expression `2 + c(1, 2, 7)` is adding a vector of length 1 to a vector of length 3. We would like *R* to be smart enough to add 2 to each element, and that is what happens:

```
c(1, 2, 7) + 2
```

```
[1] 3 4 9
```

Consider another example: `c(11, 22) + c(100, 200, 300, 400)`. Here the vectors are of length 2 and 4, respectively. When we add them together we get:

```
[1] 111 222 311 422
```

What did *R* do? It appears to have created the vector `c(11 + 100, 22 + 200, 11 + 300, 22 + 400)` and indeed that is what it did. This is a general concept in *R*; it *recycles* the values in the smaller vector to have the same length as the larger one. In this case, we recycle `11 22` to have length 4. *R* does this as the function `rep()` would, basically by concatenating several copies of the original vector to get the right length. In this case, we get `11 22 11 22`, which has the same length as the larger vector. Then *R* can do the basic arithmetic as before.

We can now understand how `2 + c(1, 2, 7)` works. The vector with 1 element is recycled so that it has length 3. What about the following expression `1:2 + 10:12`, i.e., adding vectors of length 2 and length 3. We can try it to find out:

```
1:2 + 10:12
```

```
[1] 11 13 13
```

Warning message:

In `1:2 + 10:12` :

longer object length is not a multiple of shorter object length

The first thing to note is that *R* generates a warning telling us that we may want to check whether the result is as we expected. The problem is that recycling the shorter vector did not naturally yield a vector of the same length as the longer one so *R* gives a warning. However, *R* went ahead and performed the addition using `1 2 1` and `10 11 12`. That is, *R* recycled the shorter vector once and threw away any extra elements to make a vector of the correct length.

1.13 Managing Your Workspace

We have seen how we can store the results of computations or simple values in variables. We can think of these as being stored in our workspace. The workspace is like our desk

with pieces of paper holding different information. We put these pieces of paper in different places on our desk so that we can easily find them again when we need them. The place where we put a paper allows us to quickly find it and is analogous to a variable name that allows us to easily refer to a particular set of values.

When we carry out some computations and create new variables, then we may want to save some or all of these variables for the next time we do some work in *R*. We can save our entire workspace, i.e., all the variables we have created, by calling the `save.image()` function. This function puts all the objects in our session workspace into a file named `.RData`. If we start *R* again (in that directory), the contents of that `.RData` file are loaded into the new session and are immediately available to us again. If we start *R* in a different directory, we can still load the values into the *R* session, but we must do this ourselves using the `load()` function (or pull-down menu in RStudio) and giving it the fully qualified name of the file to load (i.e., the full directory path to the file).

Quitting *R*

We can end our *R* session using the `q()` function. When we do this, we are normally asked whether we want to save the session or not. Responding “yes” calls `save.image()` implicitly.

1.13.1 Storing and Removing Variables

If we don’t want to store all the variables in our session, we can explicitly save one or more objects to a file with the `save()` function. For example, we can save `x` and `y` that we created earlier with

```
save(x, y, file = "myData.rda")
```

We pass the `save()` function all of the objects that we want to save, separated by commas, and lastly, we supply, via the `file` parameter, the name of the file to save these variables to. Saving variables is convenient when we create a big dataset and then want to ensure that it gets saved before we do anything else so that later we can pick up where we left off with our data scrubbing. Or if we want to make an object available to another *R* session, e.g., to somebody with whom we are working, then we can simply write the object to disk and send it that person in an entirely portable format. This is what we did to make the family data available to you as a `.rda` file.

In the same way that we might overload our desk with pieces of paper as we move from task to task, or just have too much information, we need to manage the variables we have in our work area or desktop. *R* provides functions which we can use to dynamically manage the variables and the contents of our workspace. The `objects()` function (or the `ls()` function) gives us the names of the variables we have in our workspace. Currently, we have the following variables,

```
objects()
[1] "fage"        "family"       "fbmi"         "fdesiredWt"
[5] "fheight"     "fnames"       "foverWt"      "fsex"
[9] "fweight"     "x"            "y"            "z"
```

We can remove variables using the `remove()` function (or the `rm()` function). We simply pass to the `remove()` function the variables that we want deleted from our workspace or the names of these variables as quoted strings. For example, the following two calls both remove variables from our workspace:

```
remove(x, y, z)
remove("fbmi", "fage", "fdesiredWt")
```

Now, the remaining variables in our session's workspace are

```
ls()
```

```
[1] "family"   "fheight"  "fnames"   "foverWt"  "fsex"      "fweight"
```

We can remove all of the variables in our workspace with

```
remove(list = objects())
```

Notice that in this call to `remove()` we provided a value for the parameter named `list` rather than naming the variables individually. We set `list` to the return value from a call to `objects()`. This return value is a character vector of the names of all objects in our workspace. Using `list` is a quick way to remove all the variables without having to type them one by one. We check that our workspace is now empty with another call to `objects()`:

```
objects()
```

```
character(0)
```

The return value is an empty character vector so indeed all our variables are gone. Remember that since we saved the family information, we can reload it into our workspace again with `load("family.rda")`. Now we have all of the family data back, but we no longer have the variables `x`, `y` and `z` that we created in this *R* session.

1.13.2 Searching for Variables

Like a calculator, *R* has several built-in named values. One such value is `pi`,

```
1 + pi
```

```
[1] 4.141593
```

In the expression `1 + pi`, the term `pi` refers to a variable. We can associate new values with this name by *assigning* a value to it. For example, we can give `pi` the value 1 and then use that in our computations, e.g.,

```
pi = 1
1 + pi
```

```
[1] 2
```

We see that `pi` is not a mathematical constant in this programming world. It is merely a variable to which we can *bind* (assign) new values. Of course, this is not a good idea. If we use this new value for π , then we may get strange results!

What happened to the original version of `pi`? We assigned a 1 as the value of `pi` and used that in our computations. We can remove this incorrect `pi` with

```
rm("pi")
```

Now that we removed it, is `pi` defined at all? Is the old value put back? The answer is that the old value is now in effect again, but it wasn't "put back." *R* did not remember the old value and restore it when we removed our version of `pi`. The explanation is a little more complicated. It relates to where we find the variable `pi`.

When we issue the command `pi = 1` we are telling *R* to associate the value 1 with the variable name `pi`. This puts it in our workspace. However, before we did this, we managed to find `pi`, and then it had the usual value of 3.14159. Where did this `pi` come from? It wasn't in our workspace, yet it was still available to us.

The answer involves understanding how *R* finds variables when we refer to them. *R* actually keeps a collection of places in which to search for variables. This is called the search path. The search path is an ordered collection of workspaces containing variables and their associated values. At any point during an *R* session, we can ask *R* what this collection of workspaces is. We do this with the `search()` function. In this session, we get

```
search()
```

```
[1] ".GlobalEnv"      "tools:rstudio"    "package:stats"
[4] "package:graphics" "package:grDevices" "package:utils"
[7] "package:datasets" "package:methods"   "Autoloads"
[10] "package:base"
```

The first entry, i.e., `".GlobalEnv"` is our own personal workspace. It's also called the global environment. When we quit *R*, this workspace disappears. The other entries are packages or libraries of functions and data that are also available to us.

When we implicitly cause *R* to look for a variable, it walks along this collection and examines each space to determine whether it has the relevant variable. Suppose as before that we have defined our own version of `pi` with the expression `pi = 1`. Then when we use `pi` in a computation, such as `1 + pi`, *R* starts its search for `pi`. It begins in the first element of the search path, i.e., in `".GlobalEnv"`, and finds `pi` there. That is, *R* finds `pi` in our workspace, where we put it.

Then after we remove `pi` (with `rm(pi)`), the search for `pi` is rather different. *R* no longer finds `pi` in `".GlobalEnv"` so the search continues. *R* looks through each element of the search path and finds `pi` only in the last entry, i.e., in `"package:base"`. This place contains the built-in variables provided by the *R* system itself (rather than add-ons).

How could we know where *R* finds a particular variable? We can use the `find()` function. In the following, we again define `pi` and then ask *R* where we can find it.

```
pi = 1
find("pi")

[1] ".GlobalEnv"    "package:base"
```

Here `pi` is found in both our workspace and base *R*, and their order in the return value tells us that the value in our workspace will be used.

Now, we remove our version of `pi` and then ask *R* again where we can find `pi`:

```
remove("pi")
find("pi")

[1] "package:base"
```

This time there is only the one `pi`, the one in "package:base".

Additionally, all the functions we have seen so far (and functions in general) are simply values assigned to variables. R finds functions in the same way when we refer to them in a computation. It looks through the search path until it finds the variable. However, it is slightly smarter for functions. If it knows we are calling the value of the variable as a function, then it will only look for a function and skip over other types of variables.

What if we look for a variable that doesn't exist? For example, suppose we use the variable named `fred` in a computation

```
fred^2
```

What happens? R looks through each element of the search path and eventually gives up, giving the error message:

```
Error: object 'fred' not found
```

We can determine whether a variable is defined using `find()`, or using the more convenient function `exists()`. For example,

```
exists('fred')
```

```
[1] FALSE
```

1.14 Getting Help

R has lots of functions (over 1500 immediately available to you and thousands more in add on packages). It is impossible to remember all of these and their details, e.g., what arguments they take, what they do in all situations, and what they return. To make effective use of R, we get into the habit of using the help system. One way to obtain help is to type `help("rnorm")` (or `help(rnorm)`) to get help on the `rnorm()` function. Then, we can scroll through the documentation provided for the functions. This typically includes: the function signature, which provides the arguments and their default values, if any; descriptions of the values expected for each of the arguments; a "see also" section, which refers the reader to related functions; and examples of how to use the function.

Additionally, online resources can be very helpful. A simple Web search that includes the letter R as one of the search terms, often provides several excellent resources. For example, the search for "R random number generation" yields links to Revolution Analytics (e.g., blog.revolutionanalytics.com), Stack Overflow (stackoverflow.com), a Wikibook on R programming, R documentation available through `help()`, and many others.

1.15 Software for Statisticians

R is an interactive and interpreted language designed by statisticians for statisticians. Interactivity is a very useful feature for statisticians. When we work with data, we often want to visualize data, look at numerical summaries, and the output from fitting a model and then

decide what to do next. This process has been given the name, Exploratory Data Analysis, or EDA for short (see Section 2.7). It is a highly iterative process where we attempt to let the data direct us as to what to do next. We try different things as we go along different branches or paths. Sometimes these lead to useful insights that we want to report. At other times, they verify that certain assumptions are justified, or they suggest trying different methods to better understand the data. The ability to dynamically specify what we want to do next is important. *R* also allows us to combine commands into a script or “program” that we can re-run on new or different data to recreate our analyses. Running the script is often termed batch programming since we are doing several commands in a single run. This combination of interactive commands and running scripts in a batch gives us the best of both worlds: we use interactive facilities during exploration, and programming facilities when the exploration is more “complete”.

When we say *R* is an interpreted language, we mean that we can give an instruction and immediately have it evaluated. Then, we can give another command. In non-interpreted languages, we must write an entire program made up of a sequence of commands before we run the code. We have to order the instructions and take account of different possibilities. Once the program is running, we cannot change the commands. All we can do is either wait for it to complete or terminate it and re-run it with the commands altered or different inputs.

There are several features of the *R* language that are specially designed for how statisticians work with and think about data. Statisticians typically want to work on groups of observations or experimental units, such as our family of 14 individuals that we have used as an example throughout this chapter. Vectors and data frames are natural extensions of this notion. The vector `height` represents the heights of the family members and we typically want to operate on the entire vector of heights to, e.g., convert the values from inches to centimeters, find the average height, or plot the distribution of height. The philosophy that operations work on an entire vector means we (the users) don’t have to write loops for many operations. Vectorized operations are very convenient.

Why *R*?

R and its predecessor *S* were developed by statisticians for data analysis and have many features that help in this process, including

Interactivity Many statisticians prefer investigating data in an interactive and flexible fashion, where a plot or results from an analysis suggests new paths to take.

Vectorized The vector is a convenient structure for representing a variable. The vector contains measurements on a set of subjects. Similarly, the data frame is a convenient data structure for holding variables from a study. A data frame is an ordered collection of vectors (variables) where the 1st value in each vector contains the measurements for the 1st subject, and so on.

Data Types Special data types have been created for nominal (factor) and ordinal (ordered factor) data, and many functions analyze data differently, depending on whether the input is numeric or a factor.

Subsetting The rich set of methods for taking subsets of data structures enables statisticians to create, examine, and compare subgroups of observations.

Creating Vectors The methods for creating vectors from sequences and repeated

values offers a flexible approach for setting up, e.g., design variables for model fitting.

Programming The ability to write code that uses control flow and to create functions helps statisticians develop statistical methodology and taylor their data analysis.

Sound Algorithms Core statistical methods utilize computationally sound algorithms to minimize computational limitations and errors.

Advanced Methods Statistical researchers contribute packages that implement their latest statistical methodologies for others to use.

The data frame is another example of a design feature for statisticians. Although it might appear to be the same as a matrix, it is not. The columns of the data frame are vectors that can be different types. A matrix is rectangular in shape, but all of the elements must be the same primitive type. The data frame is very convenient for working with, e.g., various measurements on a collection of individuals. Our `family` data frame represents a variety of measurements/variables (name, sex, height, weight, BMI, etc.) on 14 individuals. We have seen already that these variables are not all numeric, e.g., the names are strings and whether or not someone is over weight is a logical.

One of these data types is the factor. As noted earlier in this chapter, this type is somewhat unusual because the values are stored as integers with character labels and we cannot perform algebraic operations on them. This data type is designed for representing qualitative information such as nominal and ordinal variables. Examples of these are sex, marital status, and income class. Statisticians typically analyze qualitative data differently than quantitative variables. For example, it doesn't make sense to find the average of marital status. Instead, we want the number or the proportion of observations of each type of status. Many functions in *R* operate differently on a factor variable than a numeric variable. One example, is the `summary()` function, which produces counts for each level of a factor variable but a minimum, mean, median, and maximum for numeric data.

A lot of what we do in statistics and exploratory data analysis is to look at subgroups of a sample or population. We determine characteristics about that subset and compare them to other groups or the same characteristic of the overall group. Since being able to easily compute subgroups from our data is so important to statisticians, *R* offers several different ways to specify subsets of data. We saw in this chapter 5 different ways to compute a subset (logical, position, exclusion, name, and all). Additionally, there are specialized functions for taking a subset, such as `subset()` and `filter()`, `slice()` and `subset()` in the `tidyverse` package.

Working with experimental units and designs we often need to create variables that have repeated patterns of numbers, and the `seq()` and `rep()` functions (as well as others such as `expand.grid()`) can be very useful in this regard.

There are many other features in the language which have been designed with data analysis in mind, e.g. high quality graphics (Chapter 3) and more complex data structures (Chapter 4).

1.16 Summary

This chapter introduced some basic coding concepts. We saw how to write and evaluate expressions in *R*. Additionally, we introduced two basic data structures in *R*—the vector and the data frame. This introduction included taking subsets, creating vectors, and investigat-

ing data via simple summaries. Our aim is to carefully introduce these basic concepts in the language because they are the building blocks for using *R* effectively. Understanding how to parse an *R* expression, work with vectors and data frames, and call functions is the starting point to writing code.

The *R* programming language has its origins in *S*, which was developed at Bell Labs by John Chambers, Rick Becker, and Allan Wilks [1]. *R* was designed by Robert Gentleman and Ross Ihaka and is supported by the *R* Foundation for Statistical Computing [2].

1.17 Summary of Basic Functions for Working with Vectors and Data Frames

This chapter introduced many basic functions available in *R* to create and examine vectors and data frames. These are summarized and organized into three groups according to functions for examining the structure of existing vectors and data frames, creating vectors and data frames, and managing the workspace.

Examining Objects

For each of the following functions, the first argument is the required object for which information is desired.

`head()` Return the first six elements of a vector, the first six rows of a matrix or data frame, or the first six lines of a function. Use the `n` argument to specify a different number of elements to print.

`tail()` Similar to `head()`, except the *last* six elements, rows, or lines of the vector, data frame, or function, respectively, are returned. To specify the number of elements, use the `n` argument.

`length()` Return the number of elements in the supplied vector.

`dim()` Return the dimensions of the matrix, array, or data frame.

`levels()` Return the unique levels of a factor vector.

`names()` Return the names of each element of a named vector.

`nchar()` Return the number of characters in each element of the character vector.

`print()` Print relevant information about the *R* object supplied. The information displayed depends on the class of the object. For vectors, the full vector is printed. The format can be specified by the caller, e.g., the minimal number of significant digits for numeric vectors can be specified with `digits`; also, whether or not to show quotes for character vectors is specified with `quote`.

`all()` Return TRUE if all elements in the supplied logical vector are TRUE, otherwise return FALSE.

`any()` Return TRUE if any element in the supplied logical vector is TRUE.

`identical()` Return TRUE if two *R* objects are the same, including attributes like the names of the elements of the vectors.

`class()` Return the class of the *R* object supplied.

is.logical() Return TRUE if the object is of class `logical`.

is.datatype() Return TRUE if the object is of class `datatype`, where “datatype” can be an atomic vector type (e.g., `logical`, `integer`, `numeric`, or `character`), `factor`, and `dataframe`, and more.

is.na() Return TRUE if the object has any missing values.

is.null() Return TRUE if the object is NULL.

sapply() Apply to each vector in the data frame the supplied function. If possible, return the result as a vector. This function and related apply functions are described in more detail in Chapter 4

Creating vectors

The following functions can be used to create vectors, including, e.g., the sorted version of a vector.

c() Concatenate the inputs to this function together into a single vector (or list). The inputs are coerced to the same data type, if they are different.

as.datatype() Coerce the object supplied to the `datatype` class, where “datatype” can be an atomic vector type (e.g., `logical`, `integer`, `numeric`, or `character`), `factor`, `dataframe`, and more.

expand.grid() Create a data frame containing all combinations of the elements in the supplied vectors.

subset() Take a subset of rows from the data frame supplied, where the rows satisfy the supplied condition.

rep() Create a vector of repeated values from a vector. To repeat individual elements, use the `each` argument. To repeat the entire sequence, use the `times` argument; if `times` is a vector with the same length as the supplied vector of values, then each value is repeated the number of times indicated by the corresponding multiple in `times`.

seq() Create a sequence that begins with `from` and ends with `to`. The default step size of 1 is changed with the `by` argument. To return a vector with a specified length (and equal step size between the starting and ending values), use the `length.out` argument.

rev() Reverse the order of the elements of the supplied vector.

sort() Return the supplied vector with its elements in increasing order. To sort the elements in decreasing order, set `decreasing` to TRUE.

order() Return a vector of positions that sorts the supplied vector, e.g., `x[order(x)]` sorts the elements of `x`. When multiple vectors are supplied, the next one in the sequence is used to break ties. To sort the elements in decreasing order, set `decreasing` to TRUE.

Managing the Workspace

The following functions can be useful in finding, saving, removing, and loading objects in your workspace. For many of these functions the input is a character string that contains the name of the object to be, e.g., found. In this case we mention that the input is the named object, rather than the object itself.

exists() Return TRUE if the named *R* object is defined.

find() Find the named object and return the workspaces that contain an object of that name. If the object appears in multiple workspaces, these are provided in the order of the search path.

help() Display the help documentation for supplied function (or name of the function). The package of the function can be specified with the *package* argument. To get a list of functions in a package, specify the *package* argument without supplying a function.

save() Save objects to the specified *file*. To specify a character vector of the objects to be saved, use the *list* argument.

save.image() Save current workspace to specified *file*.

load() Load objects from the specified *file* into the current workspace.

rm() Remove the objects supplied as inputs (can be specified as the names of objects) from the workspace. To provide a character vector of the names of the objects to be removed, use the *list* argument.

remove() Has the same functionality as **rm()**.

search() Return the search path (no inputs are provided).

objects() Return the names of the objects in the current workspace, if no *name* is provided for the workspace in which to search.

ls() Has the same functionality as **objects()**.

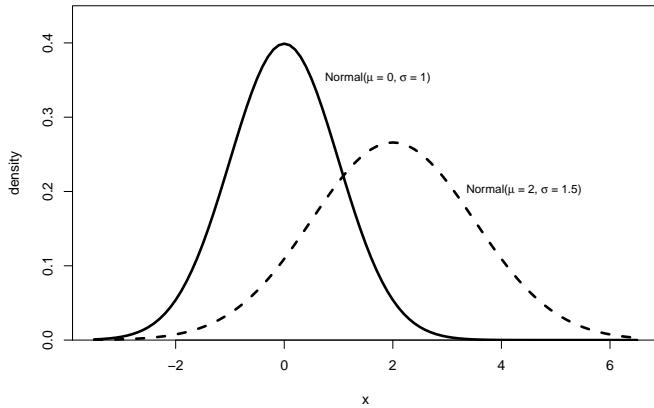
options() Set/change the behavior of the *R* session. For example, the *digits* argument controls the minimal number of significant digits that are displayed when printing numeric values and the *max.print* argument controls the maximum number of elements that are printed to screen. When called with no inputs, the current settings are returned.

1.18 Guided Practice

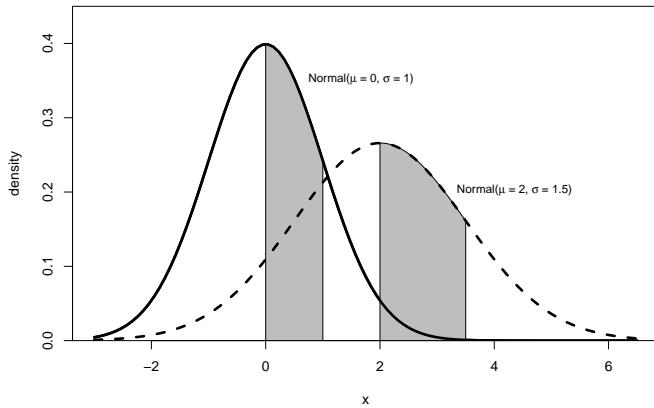
Calling Functions

These exercises provide practice with invoking functions and reading documentation on functions. Begin by reading the help page for the *R* functions for working with the normal distribution (type `?rnorm` or `help(rnorm)` at the prompt). The Usage section of the documentation shows the function's arguments, including any default values. The Arguments section explains what kind of input is expected for each of these arguments. The help page also describes other related functions, such as `pnorm()`, `dnorm()`, and `qnorm()`; a few of these functions are helpful in the following exercises.

If you are unfamiliar with the normal distribution, we provide a brief background here. The standard normal curve is centered at 0 and has a standard deviation of 1. This curve is drawn in the figure below along with a normal curve that has a center of 2 and a standard deviation of 1.5 (dashed curve).



The family of normal curves have the same shape and each is uniquely specified by its center (mean or μ) and standard deviation (sd, for short, or σ). When we generate a large number of random values from the normal distribution, the average and sd of these values are close to the mean and sd of the normal distribution. Additionally, the histogram of these values will look approximately like the normal curve. The area under a normal curve is 1, and the proportion of the random values between a and b is approximately the area between a and b under the normal curve. Moreover, the area between a and b under the normal with mean μ and sd σ equals the area under the standard normal between $(a - \mu)/\sigma$ and $(b - \mu)/\sigma$. See for example the shaded area between 2 and 3.5 under the normal $\mu = 2$, $\sigma = 1.5$ in the figure below. It is the same as the area between 0 and 1 beneath the standard normal curve.



1. Use `rnorm()` to generate 5 random values from a normal distribution with mean 17 and standard deviation 3.
2. Create a vector called `normSamps` containing 1000 random values from a normal distribution with mean 1 and standard deviation 2.
3. The 10% trimmed mean is obtained by taking the mean of the middle 80% of the values (removing the lowest 10% and the highest 10% of the values). Find the 10% trimmed mean of `normSamps`. *Hint:* Look at the help file for the `mean()` function.
4. Calculate the fraction of values in `normSamps` that are less than 3. Write your solution so that it coerces a logical vector to numeric without using the function `as.numeric()`.
5. Find the chance of drawing a value less than 3 for a normal distribution with mean 1

and standard deviation 2. That is, calculate the area under the $\text{Normal}(\mu, \sigma)$ curve to the left of 3. Use one of the functions mentioned in the help file of `rnorm()`. Compare your answer to the answer for previous question where we computed on `normSamps`, a random sample from the normal distribution.

6. A percentile is the value at which a given percentage of samples fall at or below it and is sometimes synonomously called a quantile, e.g., the 90th percentile is the value at which 90% of samples fall at or below it. Calculate the 90th percentile of `normSamps`. *Hint:* The `quantile()` function is helpful for this question.
7. Calculate the 90th percentile of a normal distribution with mean 1 and standard deviation 2. The value calculated is what we expect from a random sample from the $\text{Normal}(\mu, \sigma)$. Compare your answer to the observed sample quantile in the previous question.
8. Calculate the greatest absolute distance from 1 for the values in `normSamps`. Assign this value to a variable called `maxDev`, short for "maximum deviation" from the mean. *Hint:* The functions `abs()` and `max()` are helpful for this question.
9. Find the chance of a random value having a distance larger than `maxDev` from the center of a normal distribution with mean 1 and standard deviation 2. In other words, find the area under the $\text{Normal}(\mu = 1, \sigma = 2)$ curve that is above `1 + maxDev` or below `1 - maxDev`.

Subsetting Data Frames

This section provides practice on subsetting data frames. There are many ways to take subsets of the rows and columns of a data frame (see Section 1.9.7), but the three most popular ways are:

1. by position/index (using an integer vector),
2. by name (using a character vector), and
3. by a logical vector (indicating if an element is to be included in the subset).

The rows and columns of the subset are referenced as `dataframe[rows, columns]`. Also, the columns (vectors) of a data frame can be accessed with the \$-notation.

For the following exercises, you are to work with the data frame `family` introduced in Section 1.6. Load the data frame with `load("family.rda")`.

1. Examine the `family` data frame. What are its dimensions? What are the names and classes of the variables in the data frame?
2. Create a new data frame containing only the males in `family` who are overweight. Call this data frame `maleOWt`. Be sure to keep all variables that are present in the original `family`. Use a logical vector to create `maleOWt`. Check the dimensions of your new data frame — there should be 4 rows and 7 columns.
3. For the new data frame `maleOWt`, change the weight to a missing value (by assigning `NA`) for any individual who is named Tom. To do this, do not subset by position. Instead, use a logical vector.
4. Since `maleOWt` contains only males, the variable `gender` is no longer needed. Drop `sex` from `maleOWt`. Use subsetting with a logical vector to do this. By the end of this operation, `maleOWt` should have 4 rows and 6 columns — check that this is true.

5. All family members have the last name "Smith". Add a new variable called `lastName` to the original data frame `family`. *Hint:* Columns can be added to a data frame by subsetting or with \$-notation.
 6. Create a factor vector called `bmiLevel` that groups family members into two BMI categories (low and high), where a BMI strictly less than 23 is considered low.
-

1.19 Exercises

Bibliography

- [1] Rick A Becker, John M Chambers, and Allan R Wilks. *The New S Language: A Programming Environment for Data Analysis and Graphics*. Wadsworth, Pacific Grove, CA, 1988.
- [2] R Development Core Team. *R: A Language and Environment for Statistical Computing*. Vienna, Austria, 2016. <http://www.r-project.org>.

2

Reading and Exploring Data in Tables

CONTENTS

2.1	Introduction	51
2.2	Formats for Tabular Data	53
2.2.1	Delimited Data	54
2.2.1.1	Comma-Delimited Traffic Data	54
2.2.2	Fixed Width Format	55
2.2.2.1	Fixed Width Formatted Drug Abuse Warning Network Survey	56
2.2.3	Key-Value Pairs	58
2.3	Validating and Cleaning the Data	59
2.3.1	Updating Variable Names and Formatting Time for Traffic	60
2.3.2	Data Types for DAWN Survey	61
2.3.3	Validating Text – Hillary Clinton’s Email	64
2.4	Selecting a Structure: Data Frame, Matrix and Array	68
2.4.1	Collections of Matrices – Handwritten Digits Data	68
2.4.1.1	Applying Functions to Matrices and Arrays	70
2.5	Reshaping Data Tables	71
2.5.1	Stacking Traffic Flow	72
2.5.2	Rearranging World Bank Country Statistics	75
2.6	Merging Data Tables	80
2.6.1	Merging Names and Emails	82
2.7	Exploratory Data Analysis	85
2.7.1	Exploring Traffic on California Freeways	86
2.7.2	Exploring Country Statistics	91
2.7.3	Exploring Emergency Room Visits due to Drug Abuse ..	94
2.7.4	EDA Summary	99
2.8	Summary	102
2.9	Functions for Reading and Exploring Data	102
2.10	Guided Practice	103
2.11	Exercises	105
	Bibliography	105

2.1 Introduction

Data often are provided in a simple table-like format in a plain text file where rows correspond to observations and columns to variables. In this chapter, we examine several file formats for tables; address considerations in choosing the best structure for working with the data in *R*, which includes choosing data types, reshaping the table, and merging tables;

describe techniques for cleaning and validating data; and discuss approaches for exploring data. We use several data sets as examples throughout the chapter. These are described below. Each description includes a summary of what constitutes rows and columns in the data table.

Not all data sets have a simple table-like structure, and in Chapter 4 we tackle the challenge of working with data in more complex formats. Furthermore, in Chapter 14 we consider situations that require extensive programming to acquire and clean the data, in Chapter 9 we work with data in relational databases, and in Chapter 12 we examine data available through Web scraping, forms and APIs.

Example 2-1 Traffic on California Freeways

The freeway Performance Measurement System (PeMS) has a Web site [3] that allows registered users to get information about traffic on California's freeways. The Web site provides access to data from loop detectors, which are recording devices embedded in the road at various locations in California's freeway system. We have an extract from PeMS that contains recordings, for each of 3 lanes, of the flow (number of cars) and occupancy (the percentage of time cars were over the loop) in successive 5 minute summaries over several days at one location.

row Five minute interval in a day.

columns Time of day, and flow and occupancy at one location for each of 3 lanes (in the same direction).

Example 2-2 Drug Abuse Warning Network Survey

The US federal government conducts several large complex surveys to inform Congress and government agencies on important issues facing the US public and businesses. One such survey is the annual Drug Abuse Warning Network Survey (DAWN), which studies substance-related emergency room visits [8]. Each record in DAWN corresponds to one emergency-room visit for substance-related reasons. For each visit, information is recorded about the patient, including the type of visit (e.g., suicide attempt, adverse reaction, accidental ingestion) and the substances found in the patient.

row Emergency-room visit for a substance-related reason.

columns Patient information (e.g., age, sex, race) and substances present in patient.

Example 2-3 World Bank Reports

The World Bank provides financial and technical assistance to developing countries. In 2010, the World Bank launched an Open Data Website [9] that provides access to data from their reports on topics such as GDP, education, health, and the environment.

row A country reported in the World Bank.

columns Information for a particular year about the country, such as GDP, birth rate, educational attainment for females in various age groups, and employment to population ratios.

Example 2-4 Hillary Clinton's Emails

The US State Department has released thousands of Hillary Clinton's emails from the time

when she was Secretary of State. These emails are controversial because they include work-related emails that were sent or received by Clinton on her private email account. The emails were released as PDFs and have been made available by the Wall Street Journal at [5]. Ben Hammer has processed these PDFs into plain text files and they have been made available on Kaggle [4] for public analysis.

row An email sent to or received by Hillary Clinton's private email address.

columns Information about the email, including who sent the email, who received the email, the date the email was sent, and the redacted text of the email (extracted from a PDF file).

Example 2-5 Handwritten Digits

MNIST (Modified National Institute of Standards and Technology) is a collection of images of handwritten digits for developing algorithms to classify digits [?]. The image for each digit is a 28×28 grid of pixels. The value for a pixel is an integer between 0 and 255, inclusive, which indicates the lightness or darkness of the pixels with 0 being white or empty and 255 being black. These images have been 'hand'-classified so that we know the true number that was written.

row An image of a handwritten digit.

columns The values for the 784 pixels that compose the image.

2.2 Formats for Tabular Data

Our first step in accessing tabular data for exploration and analysis is to determine how the information is organized in the source file, such as whether or not the values are separated by commas. In this chapter, we consider 3 standard formats for the data: delimited, fixed width, and key-value. We describe each of these formats and provide examples from the data introduced in Section 2.1. We also examine binary data in Section 2.4.

Plain Text or Binary?

Statisticians often work with plain text source files. We can typically determine whether or not the file is plain text from its filename extension. Extensions such as *csv*, *txt*, *fwf*, and *dcf* correspond to plain text files. These are shorthand for comma-separated values, text, fixed-width format, and debian control format, respectively. However, filename extensions are only a convention, and they do not guarantee that the contents of the file is plain text nor need they determine the format of the data. We can confirm a file is plain text by viewing the contents in a plain text editor, such as Emacs, NotePad++, Sublime, TextWrangler, and Vim. Additionally, the file may be accompanied with documentation that indicates whether or not the contents are plain text.

Filename extensions, such as *xls*, *rda*, and *sas*, correspond to Excel, *R*, and SAS binary files. If we open one of these files with a plain text editor, the contents appear as gibberish. With these files, we typically use the associated software to read and analyze the data. Data in these formats can sometimes be exchanged between these software programs. For example, the *R* packages *haven* [14] and *R.matlab* [2] provide functions to read SPSS and

MATLAB data files into *R*, respectively. Data can also be exchanged via binary formats that do not directly map to plain text and that are not specially formatted for particular software tools. In order to read these files, we need precise knowledge of the file format. An example is provided in Section 2.4.1.

2.2.1 Delimited Data

In a typical arrangement of tabular data, one line in a source file corresponds to one row in a table, which becomes one record in an *R* data frame. For each line in the file, we need to be able to distinguish the values for the various variables. Delimited data use a delimiter, such as a comma, to separate these values. In addition to the comma, other typical delimiters include the tab, semicolon, and white space. With these input files, each row is simply divided into pieces at the locations of the delimiters. The file displayed below is an example of comma-delimited data. These data are for the simple `family` data frame introduced in Q.1-7 (page 21). Each row corresponds to a family member, and the values for the person's first name, sex, age, height, etc. are separated by commas, i.e.,

```
firstName,sex,age,height,weight,bmi,overWt
Tom,m,77,70,175,25.16239,TRUE
Maya,f,33,64,124,21.50106,FALSE
Joe,m,79,73,185,24.45884,FALSE
Robert,m,47,67,156,24.48414,FALSE
Sue,f,27,61,98,18.51492,FALSE
Liz,f,33,68,190,28.94981,TRUE
...
...
```

Notice how the lines in the file have different lengths, depending on the number of characters in the person's name, the number of digits in the weight, and whether the value for the over weight indicator is TRUE or FALSE. We can use the commas in a line to split the text into values for the variables. That is, the value before the 1st comma is assigned to the 1st variable (`firstName`), the value between the 1st and 2nd commas is the value for the 2nd variable (`sex`), and so on. We have color-coded the values for sex and weight to show that they do not line up from one record to the next, but the value for sex appears after the 1st comma and weight appears after the 4th comma in each row. If, for example, height is missing for an individual, then this can be conveyed by the absence of a value between the 3rd and 4th commas or by the presence of a special value such as -9 for height. Of course, the values in a comma-delimited file cannot contain a comma, unless they appear within a quoted value.

2.2.1.1 Comma-Delimited Traffic Data

The file `flow-occ.txt` contains recordings from the loop detector system described in Q.2-1 (page 52). That is, each row in the file corresponds to a recording taken at 3 loop detectors. These measurements are the flow (number of cars) and the occupancy (the percentage of time cars were over the loop) in a 5-minute interval for each lane. The data contain readings for successive 5-minute intervals over several days. Below is a snippet of the source file:

```
'Timestamp','Lane 1 Occ','Lane 1 Flow','Lane 2 Occ',\
'Lane 2 Flow','Lane 3 Occ','Lane 3 Flow'
3/14/2003 00:00:00,.01,14,.0186,27,.0137,17
3/14/2003 00:05:00,.0133,18,.025,39,.0187,25
3/14/2003 00:10:00,.0088,12,.018,30,.0095,11
3/14/2003 00:15:00,.0115,16,.0203,33,.0217,19
```

```
3/14/2003 00:20:00,.0069,8,.0178,25,.0123,13
3/14/2003 00:25:00,.0077,11,.0151,24,.0092,13
```

The filename extension of *txt* indicates this is a plain text file. When we examine the contents of the file, we can see that these data are comma-delimited. Notice that the first row of the file is a ‘header’ row that contains variable names. The labels ‘Lane 1 Occ’ and ‘Lane 1 Flow’ refer to the occupancy and flow measurements for the leftmost lane on the freeway. Similarly, lane 2 is the center lane, and lane 3 is the farthest right lane. The header line is displayed here across 2 rows in order for it to fit within the page margins (The \ indicates that the line in the file is continued on the following line in the display.)

We have the information required to determine how to read the data into a data frame in *R*. Specifically, the values are comma separated, and the first line is a header. We use `read_delim()` in the `readr` package [13] to read the file with

```
library(readr)
traffic = read_delim("flow-occ.txt", delim = ",")
```

Notice that we provided 2 arguments to `read_delim()` – the file name and the delimiter, both as strings. The return value from `read_delim()` is a data frame (see Section 1.8). Let’s examine the first few rows of this data frame to confirm that the data are read into *R* as expected

```
head(traffic)
```

Source: local data frame [6 x 7]

	'Timestamp'	'Lane 1 Occ'	'Lane 1 Flow'	'Lane 2 Occ'
	(chr)	(dbl)	(int)	(dbl)
1	3/14/2003 00:00:00	0.0100	14	0.0186
2	3/14/2003 00:05:00	0.0133	18	0.0250
3	3/14/2003 00:10:00	0.0088	12	0.0180
4	3/14/2003 00:15:00	0.0115	16	0.0203
5	3/14/2003 00:20:00	0.0069	8	0.0178
6	3/14/2003 00:25:00	0.0077	11	0.0151

Variables not shown: 'Lane 2 Flow' (int), 'Lane 3 Occ' (dbl),
 'Lane 3 Flow' (int)

The classes of the variables are automatically determined by `read_delim()`. We see that the occupancy variables are `numeric` (also known as double-precision, or `dbl` for short), and since flow is measured in counts of cars, the flow variables are integer vectors. The `Timestamp` variable is treated as a character vector. In Section 2.5.1, we consider whether or not these are appropriate data types for analysis and issues about the organization of the data, e.g., whether or not to keep 3 separate vectors for occupancy.

There are many functions available in *R* for reading data in `csv` files; these include `read.csv()`, `read.delim()` and `read.table()` that belong to the utilities provided in *R* (i.e., there is no need to load a separate package), and in the `readr` package we have `read_csv()` and `read_delim()`. We do not go into the details of the parameters for each of these functions, but note only that they provide very similar functionality. In this chapter, we primarily use the functions in `readr` because they have a consistent set of parameters.

2.2.2 Fixed Width Format

Another common format for tabular data is fixed width, where the value for a variable appears in the same position in each row in the source file. For example, below is a fixed-width version of the simple family from Chapter 1:

Tom	m7770175 25.16239TRUE
Maya	f3364124 21.50106FALSE
Joe	m7973185 24.45884FALSE
Robert	m4767156 24.48414FALSE
Sue	f2761 98 18.51492FALSE
Liz	f3368190 28.94981TRUE
...	

Here, the first 8 characters in each line contain the person's name. When, the name is, say, 3 letters long, then the trailing 5 characters are blank. Similarly, the values for whether or not the person is over weight (TRUE or FALSE) appear in the 25th to 29th characters in each row. Notice how the values for, say, Tom's, age, height, weight, and bmi, appear as a string of digits with no separators, e.g., 777017525.16239. In order to pull apart this string of digits into 77, 70, 175, and 25.16239 for age, height, weight, and bmi, respectively, we need to know that age appears in positions 10-11, height in 12-13, weight in 14-16, and BMI in 17-24. We have color-coded the values for sex and weight in each record, and we can see clearly that for each record these values are located in column 9 and columns 14-16, respectively. Note that even when a weight is only 2 digits, e.g., 98, the weight value has a leading blank so that it takes up 3 positions.

2.2.2.1 Fixed Width Formatted Drug Abuse Warning Network Survey

The DAWN survey data are in the file *34565-0001-Data.txt*. We assume from the file's extension that it is plain text, but we need to examine the file contents or read the accompanying codebook to determine how the values are organized in the file. Below is a single record from this file, which corresponds to one emergency room visit related to substance abuse (see Q.2-2 (page 52)).

```

1 2251082 .9426354082 3 4 1 2201141 2 865 105 1102 \
005 1 2 1 2.00-7.00-7.0000-7.0000-7.00001255 105 1142032 4 1 \
1 2.50 5.00 5.0100-7.0000-7.0000 -7 -7 -7 -7-7-7-7-7.00\ \
-7.00-7.0000-7.0000-7.0000 -7 -7 -7 -7-7-7-7-7.00-7.00-7\ \
.0000-7.0000-7.0000 -7 -7 -7 -7-7-7-7-7.00-7.00-7.0000-7\ \
.0000-7.0000 -7 -7 -7 -7-7-7-7-7.00-7.00-7.0000-7.0000-7\ \
.0000 -7 -7 -7 -7-7-7-7-7.00-7.00-7.0000-7.0000-7.0000 \
-7 -7 -7 -7-7-7-7-7.00-7.00-7.0000-7.0000-7.0000 -7 -7 \
-7 -7-7-7-7-7.00-7.00-7.0000-7.0000-7.0000 -7 -7 -7 -7-7-7-7\ \
-7-7-7-7-7.00-7.00-7.0000-7.0000-7.0000 -7 -7 -7 -7-7-7-7-7\ \
7.00-7.00-7.0000-7.0000-7.0000 -7 -7 -7 -7-7-7-7-7.00-7.\ \
00-7.0000-7.0000-7.0000 -7 -7 -7 -7-7-7-7-7.00-7.00-7.00\ \
00-7.0000-7.0000 -7 -7 -7 -7-7-7-7-7.00-7.00-7.0000-7.00\ \
00-7.0000 -7 -7 -7 -7-7-7-7-7.00-7.00-7.0000-7.0000-7.00\ \
00 -7 -7 -7 -7-7-7-7-7.00-7.00-7.0000-7.0000-7.0000 -7 \
-7 -7 -7-7-7-7-7.00-7.00-7.0000-7.0000-7.0000 -7 -7 -7-7-7\ \
-7-7-7-7-7.00-7.00-7.0000-7.0000-7.0000 -7 -7 -7 -7-7-7-7-7\ \
7-7-7-7.00-7.0000-7.0000-7.0000 -7 -7 -7 -7-7-7-7-7.00-7.00\ \
0-7.00-7.0000-7.0000-7.0000 -7 -7 -7 -7-7-7-7-7.00-7.00-7\ \

```

```
7.0000-7.0000-7.0000 -7 -7 -7 -7-7-7-7-7.00-7.00-7.0000-\n
7.0000-7.00008 611001
```

We can see that this record has a fixed-width format. Since there are no delimiters between values, we must specify the positions for the variable values when reading the data into *R*. These records are very long, with this one record taking 21 lines of text to be displayed here. The data file comes with a 2,356 page codebook that describes the format, which is essential for determining the locations of the variables.

Figure 2.1 shows screenshots for the codebook entries for the SEX and CASETYPE variables. We see from the codebook that the sex of the patient is provided in columns 36-37. We also see that 1 stands for male, 2 for female, and -8 for ‘not documented’. Similarly, the type of visit appears at location 1214 and has 8 possible values.

SEX	GENDER																																													
Location:	36-37 (width: 2; decimal: 0)																																													
Variable Type:	numeric																																													
Range of Missing Values (M):	-8																																													
<table border="1"> <thead> <tr> <th>Value</th><th>Label</th><th>Unweighted Frequency</th><th>%</th><th>Valid %</th></tr> </thead> <tbody> <tr> <td>1</td><td>MALE:(1)</td><td>119111</td><td>52.0 %</td><td>52.0%</td></tr> <tr> <td>2</td><td>FEMALE:(2)</td><td>110030</td><td>48.0 %</td><td>48.0%</td></tr> <tr> <td>-8 (M)</td><td>NOT DOCUMENTED:(-8)</td><td>70</td><td>0.0 %</td><td>-</td></tr> </tbody> </table>		Value	Label	Unweighted Frequency	%	Valid %	1	MALE:(1)	119111	52.0 %	52.0%	2	FEMALE:(2)	110030	48.0 %	48.0%	-8 (M)	NOT DOCUMENTED:(-8)	70	0.0 %	-																									
Value	Label	Unweighted Frequency	%	Valid %																																										
1	MALE:(1)	119111	52.0 %	52.0%																																										
2	FEMALE:(2)	110030	48.0 %	48.0%																																										
-8 (M)	NOT DOCUMENTED:(-8)	70	0.0 %	-																																										
Based upon 229141 valid cases out of 229211 total cases.																																														
CASETYPE	TYPE OF VISIT																																													
Location:	1214-1214 (width: 1; decimal: 0)																																													
Variable Type:	numeric																																													
<table border="1"> <thead> <tr> <th>Value</th><th>Label</th><th>Unweighted Frequency</th><th>%</th><th>Valid %</th></tr> </thead> <tbody> <tr> <td>1</td><td>SUICIDE ATTEMPT:(1)</td><td>9033</td><td>3.9 %</td><td>3.9%</td></tr> <tr> <td>2</td><td>SEEKING DETOX:(2)</td><td>14841</td><td>6.5 %</td><td>6.5%</td></tr> <tr> <td>3</td><td>ALCOHOL ONLY (AGE < 21):(3)</td><td>7421</td><td>3.2 %</td><td>3.2%</td></tr> <tr> <td>4</td><td>ADVERSE REACTION:(4)</td><td>88096</td><td>38.4 %</td><td>38.4%</td></tr> <tr> <td>5</td><td>OVERMEDICATION:(5)</td><td>18146</td><td>7.9 %</td><td>7.9%</td></tr> <tr> <td>6</td><td>MALICIOUS POISONING:(6)</td><td>793</td><td>0.3 %</td><td>0.3%</td></tr> <tr> <td>7</td><td>ACCIDENTAL INGESTION:(7)</td><td>3253</td><td>1.4 %</td><td>1.4%</td></tr> <tr> <td>8</td><td>OTHER:(8)</td><td>87628</td><td>38.2 %</td><td>38.2%</td></tr> </tbody> </table>		Value	Label	Unweighted Frequency	%	Valid %	1	SUICIDE ATTEMPT:(1)	9033	3.9 %	3.9%	2	SEEKING DETOX:(2)	14841	6.5 %	6.5%	3	ALCOHOL ONLY (AGE < 21):(3)	7421	3.2 %	3.2%	4	ADVERSE REACTION:(4)	88096	38.4 %	38.4%	5	OVERMEDICATION:(5)	18146	7.9 %	7.9%	6	MALICIOUS POISONING:(6)	793	0.3 %	0.3%	7	ACCIDENTAL INGESTION:(7)	3253	1.4 %	1.4%	8	OTHER:(8)	87628	38.2 %	38.2%
Value	Label	Unweighted Frequency	%	Valid %																																										
1	SUICIDE ATTEMPT:(1)	9033	3.9 %	3.9%																																										
2	SEEKING DETOX:(2)	14841	6.5 %	6.5%																																										
3	ALCOHOL ONLY (AGE < 21):(3)	7421	3.2 %	3.2%																																										
4	ADVERSE REACTION:(4)	88096	38.4 %	38.4%																																										
5	OVERMEDICATION:(5)	18146	7.9 %	7.9%																																										
6	MALICIOUS POISONING:(6)	793	0.3 %	0.3%																																										
7	ACCIDENTAL INGESTION:(7)	3253	1.4 %	1.4%																																										
8	OTHER:(8)	87628	38.2 %	38.2%																																										
Based upon 229211 valid cases out of 229211 total cases.																																														

Figure 2.1: Screenshot of Codebook Entries for the DAWN Survey . *The codebook entry for a variable provides the variable name, definition, location in the row, possible values and their labels, etc. For the sex variable, these are SEX; GENDER; 36-37; 1, 2, and -8; and male, female, and ‘not documented’, respectively. The counts for each value are provided to help us check whether or not we have correctly read the data.*

We have skimmed the codebook and selected several variables to extract for further investigation. To do this, we place the starting position of each variable in a vector, which we call `start`; the ending positions appear in `end`, and variable names of our choosing are in `varNames`, e.g.,

```
start = c(1, 9, 11, 14, 15, 30, 34, 36, 38, 45, 46, 66, 68, 70,
       75, 80, 87, 94, 1214, 1215, 1217, 1218, 1219, 1220, 1221)
end = c(6, 10, 13, 14, 29, 33, 35, 37, 39, 45, 47, 67, 69, 74, 79,
       86, 93, 100, 1214, 1216, 1217, 1218, 1219, 1220, 1221)
varNames = c("id", "strata", "psu", "replicate", "wt",
            "psuframe", "age", "sex", "race", "daypart",
            "numsubs", "toxtest", "s1", "s2", "s3", "s4",
```

```
"s5", "s6", "type", "disp", "alc", "nonalc",
"pharma", "nonmedpharma", "allabuse")
```

We use the `fwf_positions()` function in `readr` to collect this information into one object that we pass to the `read_fwf()` function, i.e.,

```
library(readr)
pos = fwf_positions(start, end, col_names = varNames)

dawn = read_fwf("ICPSR_34565/DS0001/34565-0001-Data.txt",
                 col_positions = pos)
```

We explore and check values for these variables later in Section 2.3.2.

2.2.3 Key-Value Pairs

We mention one other plain text format: the key-value pair. The idea is that the value for a variable in a record is provided along with its variable name, which is called the ‘key’. A typical key-value pair appears as `age:77`, i.e., the delimiter between the key, `age` and its value `77` is often a colon. Another common delimiter is the equal sign. A typical record has each variable appearing on its own line in the source file with records separated by blank lines. For example, below are the key-value pairs for the first two family members.

```
firstName:Tom
sex:m
age:77
height:70
weight:175
bmi:25.16
overWt:TRUE

firstName:Maya
sex:f
age:33
height:64
weight:125
bmi:21.50
overWt:FALSE
```

This particular arrangement of key-value pairs is known as `dcf`, short for Debian control format. It is commonly used to hold properties lists on UNIX-type operating systems. Another arrangement of key-value pairs has all of the key-value pairs for one record appear on one line with delimiters separating these pairs.

We can read the family data in this `dcf` format into `R` with the `read.dcf()` function as follows:

```
family.dcf = read.dcf("family.dcf")
```

When we examine the first few values of `family.dcf` and check the class, we find that all of the values are character strings and the structure is a matrix, not a data frame, i.e.,

```
head(family.dcf)
```

```

firstName sex age height weight bmi      overWt
[1,] "Tom"   "m"  "77" "70"   "175"  "25.16" "TRUE"
[2,] "Maya"  "f"  "33" "64"   "125"  "21.50" "FALSE"
[3,] "Joe"   "m"  "79" "73"   "185"  "24.46" "FALSE"
[4,] "Robert" "m"  "47" "67"   "156"  "24.48" "FALSE"
[5,] "Sue"   "f"  "27" "61"   "98"   "18.51" "FALSE"
[6,] "Liz"   "f"  "33" "68"   "190"  "28.95" "TRUE"

```

```

class(family.dcf)
[1] "matrix"

```

The `matrix` class is discussed in more detail in Section 2.4. We leave to the exercises the task of converting this matrix into a data frame for analysis.

Reading Tabular Data

Data are often naturally arranged in a table-like format in plain text files. To read the raw data into *R*, we need to know how the records and their values are stored in the source file. We can address the following questions to ascertain this information.

- Is the file plain text? We can often determine this by viewing the file contents in a plain text editor. More simply, we can examine the filename extension (csv, txt, fwf, and dcf are examples of plain text filename extensions) or check any available documentation.
- Can the data be arranged in a table format, i.e., with rows for observations/records and columns for variables?
- How are the values for the variables distinguished –
 - Are there delimiters between values (e.g., comma-separated values)?
 - Does a value for a variable appear in the same position in each line in the source file (fixed width format)?
 - Are the values supplied in key-value pairs (with the key as the variable name)?
- Is there any encoding for special characters?

2.3 Validating and Cleaning the Data

After we read data into *R* we want to check the results to determine whether: a) we have correctly read the data; b) the data require further processing for analysis; c) there are problems with the data; and d) the data have unexpected characteristics that influence our approach to their analysis. Some typical mistakes that we can make when reading the data into *R* include the incorrect specification of the file format, e.g., misspecification of the delimiter, the presence or absence of a header line, and incorrect data types. The most common data type misspecification occurs when strings are treated as a `factor` rather than a `character` class, or vice versa. How do we decide between `factor` and `character`? If there are only a fixed set of pre-determined values the variable can take and these correspond

to categories in a variable, then we typically use the `factor` data type. On the other hand, when the strings can take on potentially any possible value, such as with a street address or person's name, then we typically use the `character` class. We provide an example of data-type conversion for the traffic data in Section 2.3.1.

What kind of further processing might we need to do? One example is the conversion of strings into a factor. Another example is the specification of missing values. We may want to re-read the data and specify how missing values are represented in the source file via a parameter in the function we use to read the data. For example, the `na` parameter in `read_delim()` allows us to provide a character vector of missing values which are converted into NA in the resulting data frame. Alternatively, we can replace specific values in a variable in the data frame, such as -8 for gender, with NA. We provide an example of this with the DAWN survey data in Section 2.3.2. We may also need to process the data to create new variables. In Section 1.7 we calculated desired BMI from `height` and `desiredWt` for the `family`. This new variable can be added to the data frame to keep all measurements on the family together in one structure.

What kinds of problems might we uncover when we validate our data? In addition to the discoveries that we have described already that result from not properly reading the data, we may find problems with the quality of the data. When this happens, we need to determine if it's feasible to clean the data, and if so, how to fix the problems. We provide an example with the Clinton email data in Section 2.3.3.

We leave the topic of exploring the data for insights about the distribution of values to Section 2.7.

2.3.1 Updating Variable Names and Formatting Time for Traffic

In Section 2.2.1.1, we successfully read the traffic data into *R*. Although this is a simple data set, there are a couple of issues we can address to make the data easier to analyze and work with. First, the names of the variables in the data frame are long and contain blanks and quotation marks, i.e.,

```
names(traffic)
[1] "'Timestamp'"   "'Lane 1 Occ'"   "'Lane 1 Flow'"  "'Lane 2 Occ'"
[5] "'Lane 2 Flow'" "'Lane 3 Occ'"   "'Lane 3 Flow'"
```

These variable names are cumbersome to use in expressions, e.g., to compute summary statistics for the flow in lane 1 we call `summary()` with

```
summary(traffic$"Lane 1 Flow")
Min. 1st Qu. Median Mean 3rd Qu. Max.
0.0    17.0   67.0  65.8  101.0 203.0
```

We can rename these variables with shorter simpler names, e.g.,

```
names(traffic) = c("time", "occ1", "flow1", "occ2", "flow2",
                   "occ3", "flow3")
```

Notice that when we place `names(traffic)` on the lefthand side of an assignment statement, then the variable names are reassigned.

Alternatively, we can specify these shorter names when we read in the data. In this case, we want to skip the first line of the file (it contains the variable names) and specify the names ourselves. We can do this with

```
traffic = read_delim("flow-occ.txt", delim = ",", skip = 1,
                      col_names = c("time", "occ1", "flow1", "occ2",
                                    "flow2", "occ3", "flow3"))
```

Now it's much easier to write expressions with these variables, e.g.,

```
summary(traffic$flow1)
Min. 1st Qu. Median Mean 3rd Qu. Max.
0.0    17.0   67.0  65.8 101.0 203.0
```

Another issue with these data is that the times are stored in a character vector. Strings are not very useful in data analysis. For example, we can't easily use them to examine the times of day when traffic flow is at its peak. However, we can convert these strings to a time data type with

```
traffic$time = as.POSIXct(traffic$time,
                           format = "%m/%d/%Y %H:%M:%S")
```

Here, the format parameter specifies how the times are reported, e.g., "3/14/2003 00:00:00" corresponds to a month/day/year hour:minutes:second format and the shorthand for this format is "%m/%d/%Y %H:%M:%S". The `POSIXct` data type for dates is a standard format developed by the IEEE (Institute of Electrical and Electronics Engineers) Computer Society and is recognized and properly handled by many *R* functions. For example, we can plot flow against time with the `plot()` function, and `plot()` automatically labels the x-axis tick marks so they display the days of the week (see Figure 2.2).

```
plot(flow1 ~ time, data = traffic, type = "l",
      ylab = "Number of cars per 5 minute interval", xlab = "Time")
```

The plot in Figure 2.2 aids us with data validation. We can see that the peaks of traffic correspond to morning and evening rush hours and that traffic on the weekend has a lower volume and is less sharply peaked, which indicates that we have properly read and converted the time values. We continue our exploration of these data in Section 2.5.1 and Section 2.7.1.

2.3.2 Data Types for DAWN Survey

The codebook for the DAWN survey provides extensive documentation of the data, and we can use the information there to validate our work. We begin by examining the structure of our data frame to find that it has the correct dimensions, 229211 by 25, and that the variables are all `numeric/integer`. We do this with

```
str(dawn)
Classes 'tbl_df', 'tbl' and 'data.frame':
  229211 obs. of  25 variables:
 $ id          : int  1 2 3 4 5 6 7 8 9 10 ...
 $ strata      : int  25 29 7 8 22 10 3 51 3 46 ...
 $ psu         : int  108 129 25 29 94 40 6 232 5 210 ...
 $ replicate   : int  2 2 1 2 2 1 1 2 2 2 ...
 $ wt          : num  0.943 5.992 4.723 4.08 5.178 ...
 $ psuframe    : int  3 9 6 6 10 112 7 25 7 2 ...
 $ age         : int  4 11 11 2 6 11 5 8 1 7 ...
 $ sex         : int  1 1 2 1 1 1 2 1 1 1 ...
 $ race        : int  2 3 2 3 3 -8 3 1 3 1 ...
 ...
 ...
```

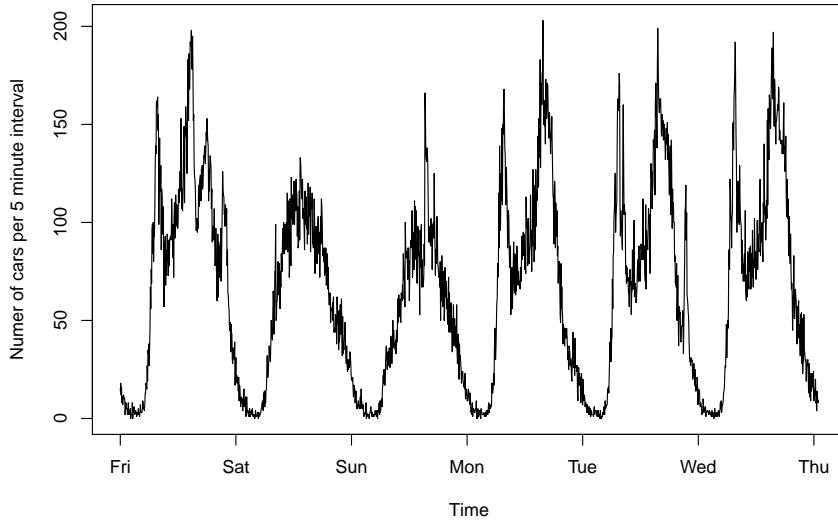


Figure 2.2: Traffic Flow in Leftmost Lane. *This line plot shows how the traffic flow varies throughout the day and across days. The sharp peaks correspond to heavy traffic, for Monday through Friday we see roughly two of these each day for morning and evening rush hours. The traffic on the weekend is generally lighter and less spiked.*

The `str()` function provides basic information about the variables, but to learn about the distribution of the values for a variable, we need another approach.

We can compare tallies of each variable's values to the codebook to see whether or not we have correctly identified positions of the variables in the records. With fixed-width formats, it is easy to make a mistake in specifying the start or end position of a variable. If a codebook is not available, then we want to examine these tallies to check that the values are reasonable. We check `sex`, `toxtest`, and `type` with

```
table(dawn$sex)
-8      1      2
70 119111 110030

table(dawn$toxtest)
-9      1      2
7946 23329 197936

table(dawn$type)
1      2      3      4      5      6      7      8
9033 14841 7421 88096 18146 793 3253 87628
```

These counts match those in the codebook (see Figure 2.1).

We also see that `sex` and `toxtest` have some negative values. Survey data often use different (and atypical) values to denote various reasons for missing information. The DAWN codebook tells us that -7 means not applicable, -8 is for undocumented, and -9 codes for

missing. We must decide whether we want to treat all 3 values as NA or keep the values as is and decide how the missing values are handled for each computation. In this instance, there are only a few missing values and there does not appear to be more than 1 type of missing value for any variable so we assign all negative values to NA. To do this, we re-read the raw data with `read_fwf()` and use the functions `na` argument to specify the values to convert to NA. We do this with

```
missing = c("-7", "-8", "-9")
dawn = read_fwf("ICPSR_34565/DS0001/34565-0001-Data.txt",
                 col_positions = pos, na = missing)
```

We check `sex` and `toxtest` to confirm that -7 and -9 have been converted into NA. We do this with simple summary statistics, e.g.,

```
summary(dawn$sex)
Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
1.00 1.00 1.00 1.48 2.00 2.00 70

summary(dawn$toxtest)
Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
1 2 2 2 2 2 7946
```

Notice that these statistics are not particularly meaningful. What does it mean that the average value of `sex` is 1.48? The problem is that `sex` is a categorical variable so averages are not useful summaries (the same holds for `toxtest` and `type`). A more meaningful statistic is the proportion of males and females. If the variable is a `factor`, then the `summary()` function provides that sort of tally.

Survey data often contain many categorical variables, which is the case with the DAWN data. This typically occurs because the questions on a survey ask the respondent to choose an answer from a list of possibilities, e.g. race, education level. Also, for privacy reasons, exact values are often not ascertained, e.g., income level. Unlike the functions to read data in base R, the functions in the `readr` package do not allow the caller to specify `factor` as a category. This means that we must convert these variables to `factors` after we have read them into a data frame. To do this, we use the `factor()` function with

```
dawn$sex = factor(dawn$sex, labels = c("male", "female"))
dawn$toxtest = factor(dawn$toxtest, labels = c("yes", "no"))

typeLabels = c("suicide", "detox", "alcohol", "adv react",
              "overmed", "poisoning", "acc ingest", "other")
dawn$type = factor(dawn$type, labels = typeLabels)

ageLabels = c("5 & under", "6 - 11", "12 - 17", "18 - 20",
             "21 - 24", "25 - 29", "30 - 34", "35 - 44",
             "45 - 54", "55 - 64", "65 & over")
dawn$age = factor(dawn$age, labels = ageLabels, ordered = TRUE)
```

Note that we have converted `age` to an ordered `factor` because the categories correspond to age intervals that have an ordering, e.g., a person in the '5 & under' category is younger than someone in the '6 - 11' category, and so on. The variables `sex`, `toxtest`, and `type` are not ordered.

Again, we confirm that the tallies for the various categories for these variables match the codebook in Figure 2.1.

```
summary(dawn$sex)
```

male	female	NA's
119111	110030	70

```
summary(dawn$type)
```

suicide	detox	alcohol	adv	react	overmed
9033	14841	7421		88096	18146
poisoning	acc	ingest		other	
	793	3253		87628	

These counts match those in the codebook, except that the labels for the levels are used rather than numeric value and the counts for negative values are shown as counts for NAs.

2.3.3 Validating Text – Hillary Clinton’s Email

The file made available on Kaggle (see Q.2-4 (page 52)) is called *Emails.csv*. To confirm that the values are indeed comma-separated and whether or not there is a header, we open the file in TextEdit (see Figure 2.3). There we see that the file contains a header line and the values in the first row are comma-separated. Given this information, we read the raw data into *R* with

```
emails = read_delim("clinton/Emails.csv", delim = ",",
                     col_names = TRUE)
```

We shorten the name of the variable *SenderPersonId* to *FrId* (short for ‘From Id’) and examine the first few records of the data frame to compare the values of *ExtractedFrom* and *MetadataFrom* as we would expect them to be the same or similar. There is limited documentation about these variables on Kaggle and the associated github site so we want to explore them in greater depth to understand their distinctions.

```
names(emails)[6] = "FrId"
head(emails[, c("ExtractedFrom", "MetadataFrom", "FrId")], 10)
```

Source: local data frame [10 x 3]

	ExtractedFrom (chr)	MetadataFrom (chr)	FrId (int)
1	Sullivan, Jacob J <Sullivan11@state.gov>	Sullivan, Jacob J	87
2		NA	NA
3	Mills, Cheryl D <MillsCD@state.gov>	Mills, Cheryl D	32
4	Mills, Cheryl D <MillsCD@state.gov>	Mills, Cheryl D	32
5		NA	80
6		NA	80
7	Mills, Cheryl D <MillsCD@state.gov>	Mills, Cheryl D	32
8		NA	80
9	Sullivan, Jacob J <Sullivanli@stategov>	Sullivan, Jacob J	87
10		NA	NA

```

Id,DocNumber,MetadataSubject,MetadataTo,MetadataFrom,SenderPersonId,Meta
dataDateSent,MetadataDateReleased,MetadataPdfLink,MetadataCaseNumber,Met
adataDocumentClass,ExtractedSubject,ExtractedTo,ExtractedFrom,ExtractedC
c,ExtractedDateSent,ExtractedCaseNumber,ExtractedDocNumber,ExtractedDate
Released,ExtractedReleaseInPartOrFull,ExtractedBodyText,RawText
1,C05739545,WOW,H,"Sullivan, Jacob J",
87,2012-09-12T04:00:00+00:00,2015-05-22T04:00:00+00:00,DOCUMENTS/
HRC_Email_1_296/HRCH2/DOC_0C05739545/
C05739545.pdf,F-2015-04841,HRC_Email_296,FW: Wow,, "Sullivan, Jacob J
<Sullivan11@state.gov>,, "Wednesday, September 12, 2012 10:16
AM",F-2015-04841,C05739545,05/13/2015,RELEASE IN FULL,, "UNCLASSIFIED
U.S. Department of State
Case No. F-2015-04841
Doc No. C05739545
Date: 05/13/2015
STATE DEPT. - PRODUCED TO HOUSE SELECT BENGHAZI COMM.
SUBJECT TO AGREEMENT ON SENSITIVE INFORMATION & REDACTIONS. NO FOIA
WAIVER.
RELEASE IN FULL
From: Sullivan, Jacob J <Sullivan11@state.gov>
Sent: Wednesday, September 12, 2012 10:16 AM
To:
Subject: FW: Wow
From: Brose, Christian (Armed Services) (mailto:Christian_Brose@armed-
service.essenate.gov)
Sent: Wednesday, September 12, 2012 10:09 AM

```

Figure 2.3: Screenshot of a TextEdit Display of *Emails.csv*. From this *TextEdit* display we see that the first line (which is wrapped across 5 lines in the display) contains a header and the variable names are comma-separated. The remainder of the display shows the contents of the second record in the file, the bulk of which is the value for the *RawText* field.

We make several observations: it appears that H in *MetadataFrom* stands for Hillary Clinton; the *FrId* appears to be a unique numeric identifier for the sender; the extracted version of the ‘from’ field contains both names and email addresses and the metadata version contains only names; Jacob J Sullivan has 2 different email addresses and the second one appears to be incorrect as the domain is *stategov* rather than *state.gov*. This suggests that we may want to investigate how clean are the data.

We have found out a lot about the data from looking at the first 10 records. However, this is a very small fraction of the data and we want to examine more to learn about the variables. One way to do this is to examine the distribution of the frequency of senders, i.e., the counts of emails received from different addresses. We use *MetadataFrom* to compute these counts, as the meta data are provided by the Freedom of Information Act (FOIA) release and may be more accurate than the values extracted from the PDFs.

```
fromCounts = table(emails$MetadataFrom)
```

We make a histogram of these counts with

```
hist(fromCounts, xlab = "Emails Sent", main = "")
```

The histogram appears in Figure 2.4. We can see that the number of emails from different addresses is highly skewed with a handful of addresses sending hundreds (even thousands) of emails.

Let’s examine a few of the frequent senders; we can do this by sorting the counts in the *fromCounts* table and examining, say, the top 5:

```
sort(fromCounts, decreasing = TRUE)[1:5]
```

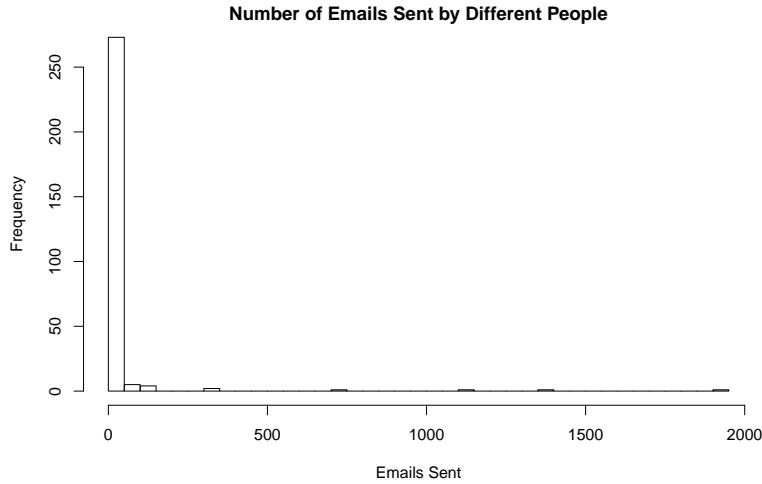


Figure 2.4: Distribution of the Number of Emails Sent to Clinton. *This histogram shows that most people who send emails to Clinton send very few, i.e., well under 25, and a few people send hundreds, even thousands of emails. To get a better sense of the distribution, we may want to plot the histogram on log scale, or to zoom in on those sending fewer than 100 emails.*

H	Abedin, Huma	Mills, Cheryl D
1906	1380	1146
Sullivan, Jacob J	sbwhoeop	
750	316	

In hindsight, these are not surprising because the emails include those sent and received by Clinton, and because Mills, Abedin, and Sullivan are 3 of Clinton's long-time staff and advisors. We earlier uncovered a problem with multiple values for Sullivan in `ExtractedFrom` so it makes sense to check whether there are any issues of multiplicity of identifiers in `MetadataFrom`. To do this, we use `order()` to find the cell in the table that corresponds to this count for Sullivan.

```
order(fromCounts, decreasing = TRUE) [4]
```

```
[1] 257
```

This implies Sullivan is the 257th record in `fromCounts`. Since the table `fromCounts` is in alphabetical order according to `MetadataFrom`, let's examine the neighboring cells to Sullivan, Jacob J, e.g.,

```
fromCount[253:260]
```

Strider, Burns	Sullivan JJ@state.gov	Sullivan, Jack
2	2	2
Sullivan, Jacob	Sullivan, Jacob J	Sullivan, Jake
2	750	37
Sullivan, Jake J	SullivanJJ@state.gov	
14	24	

It appears that 7 of these entries correspond to the same person, but their names differ slightly. That is, the values in `MetadataFrom` include: a nickname such as Jake, rather than Jacob; the presence of a middle initial J or not; and an email address rather than a person's name. The most common form of Sullivan's `MetatdataFrom` value occurs 750 times and the other forms have counts that vary between 2 and 37. This may seem small, but when added together they amount to 81 emails, which is about 10% of the total sent by Sullivan.

We also want to corroborate the sender ID with these values, i.e., do each of these variants of Sullivan's name have the same `FrId`? We saw from the first few rows of the data frame that Sullivan is associated with the id 87. Let's count how many records have an `FrId` of 87. We do this with

```
sum(emails$FrId == 87, na.rm = TRUE)  
[1] 871
```

This is 40 more than the counts we have been examining. On closer inspection of the `fromCounts` table, we find that there are 3 more cells for Sullivan that begin with 'jake' and these account for 40 emails. This indicates that if we want to track who is who, then we better use `FrId`. Furthermore, if we want to associate a name with the `FrId` number, then we should use a second file provided by Kaggle called *Persons.csv*. This file links the identifiers with names. We take this task up in Section 2.6.1.

These explorations highlight the problems related to working with data derived from text. Text data are often very messy and it can be a long process to clean the data to get more accurate values. In this cleaning process, it's advisable to use code to change values in variables, rather than editing the source file 'by hand'. This way there is a record of how the data were modified, and if the data are later updated and reissued, then the code can be re-run to clean the new version. We do not pursue this topic further here, but note that to edit text data, we typically employ string manipulation techniques and regular expressions. These are the topic of Chapter 8.

Data Validation Checklist

In our initial examination of the data, we want to check the following characteristics of the variables to confirm that they are reasonable and as expected.

- Range of values
- Rough distribution of values
- Prevalence of missing values
- Unusual or anomalous values
- Consistency across subgroups
- Consistency between variables

To validate the data, we typically compute the following numerical summaries and visualizations:

- Examine the first and last few records
- Compute numerical summaries (min, max, quartiles, etc.) for quantitative data and tables of counts of values for qualitative data.

- Visualize the data with
 - univariate plots of individual variables, e.g., histograms, density curves, bar plots, dot charts.
 - comparative plots of individual variables across subgroups, e.g., side-by-side box plots, bar plots, and dot charts, and super-posed density plots.
 - bivariate plots of relationships, e.g., scatter plots, mosaic plots.

These plots are reviewed in Chapter 3.

2.4 Selecting a Structure: Data Frame, Matrix and Array

The data frame is a natural structure for tabular data, where our variables can have different data types and the rows all have the same number of variables in each of them. However, it isn't the sole way for us to represent data in *R*. We introduce some alternatives in Chapter 4, including examples of list structure for data that are provided in *JSON* format. In addition in Chapter 9, we examine data arranged in multiple tables in relational databases, and in Chapter 12, we introduce the *XML* format which can represent complex data structures. In this section, we introduce one more example of tabular data that are naturally stored in a collection of matrices, rather than a data frame. With a matrix, the columns must all have the same data type. For this reason, the matrix does not have the same flexibility as the data frame, but nonetheless, it can be a very useful structure. We present an example, where the columns all contain the same type of quantity: pixels in an image that take on values ranging between 0 and 255. We also introduce functionality for working with matrices; in particular, we demonstrate how to use the `apply()` function, which is especially suited for working with matrices and arrays.

2.4.1 Collections of Matrices – Handwritten Digits Data

In Q.2-5 (page 53), we described data containing thousands of images of handwritten digits. Each image consists of a 28×28 pixel grid, where each pixel has an integer value between 0 and 255. All together, there are 28×28 or 784 pixel values for each image. As described in that example, we can think of a tabular format for these data where a row represents an image and each row contains 784 values, corresponding to a sequence of the 784 pixels in the image. That is, the 784 values are arranged so that the same pixel position in each image falls in the same column in the table. However, there is an alternative arrangement that is a more natural organization of the values as a matrix for each image. In this case, the cells in the matrix map directly to the layout of the pixels in the image, i.e., each image is represented as a 28×28 matrix of values. For example, the value in the 1st row and 28th column of the matrix corresponds to the top right pixel in the image. Furthermore, we can collect these matrices into a set of matrices, i.e., a 3-dimensional array. In this array, the 1st dimension corresponds to a row of pixels in an image, the 2nd to a column of pixels in that image, and the 3rd dimension of the array corresponds to the different images. Figure 2.5 provides a conceptual diagram of a similar, but smaller structure.

The files that contains these data are not plain text. The values are in a binary format, where each pixel is represented by 1 byte, i.e., a collection of 8 bits (or 0-1 values). For example, the 8-bit binary representation of the number 121 is 01111001 because these 8

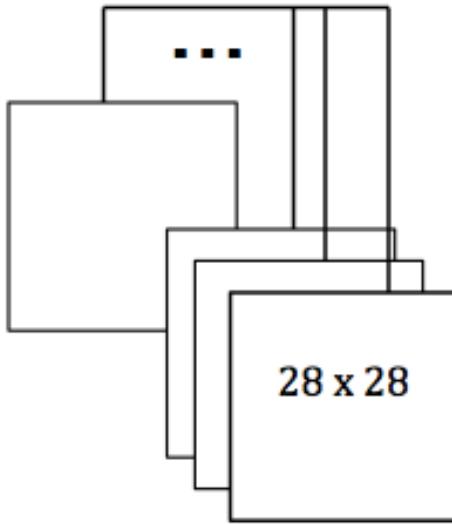


Figure 2.5: Conceptual Diagram of a 3-Dimensional Array. *This diagram shows a conceptual representation of a 3-dimensional array for handwritten digits. It is a collection of 1000 matrices that have 28 rows and 28 columns. Each matrix represents the pixel values for one handwritten image.*

bits provide the coefficients for the powers of 2 that correspond to that number, i.e.,

$$0 * 2^7 + 1 * 2^6 + 1 * 2^5 + 1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0$$

Notice the exponent corresponds to the position in the binary number and the coefficient is the value at that position. We typically work with numbers in base 10, but the same idea holds. That is, 121 is

$$1 * 10^2 + 2 * 10^1 + 1 * 10^0$$

In base 10, the coefficients can range from 0 to 9.

There is a lot of flexibility in how binary data can be stored, which means that we have to provide more information in order to read the values in the file. We must specify that the data are 1-byte unsigned integers, meaning that each number consists of 8-bits like the example of 121 as 01111001. In addition, we must specify how many bytes to read from the file. The online documentation indicates that each file contains 1000 images so we want to read $1000 * 28 * 28$, or 784000 bytes from a file. We also must manage the opening and closing of the file for reading. That is, we need to establish a connection to a file so that we can read the data, and once we have completed reading the data, we close the connection.

We begin by reading the data for the digit 4. The images for each digit are stored in separate files, e.g., *data4* contains all 1000 images for 4. We chose this digit because it is asymmetric so that we can easily tell from a plot if we have correctly read the data or mistakenly reversed the order of the pixels. We open the connection to the file, *data4* with

```
digitdata = file("data4", "rb")
```

Then, we read the 1000 sets of 784 pixels with

```

pixels = readBin(digitdata, integer(), size = 1, signed = FALSE,
                  n = 1000 * 28 * 28)
class(pixels)
[1] "integer"

length(pixels)
[1] 784000

```

We see that all $1000 \times 28 \times 28$ pixel values have been read into *R* as an `integer` vector. We can check that the values are all between 0 and 255 with

```

summary(pixels)

Min. 1st Qu. Median      Mean 3rd Qu.      Max.
0        0        0       31        0       255

```

It appears that we have properly read the data so we close the connection with

```
close(digitdata)
```

We can arrange the vector into a 3-dimensional array with

```

imageArray = array(pixels, dim = c(28, 28, 1000))
dim(imageArray)
[1] 28   28  1000

```

How can we confirm that we have properly arranged the data? For example, does each matrix correspond to an image of the number 4? Given that our data are somewhat unusual in format, we can't answer these questions with simple summary statistics. Instead, to validate our data, we can make a plot that corresponds to the image, i.e., a 28×28 grid of shaded squares where the shading scales with the pixel value.

Let's start by checking the first image, i.e., `imageArray[, , 1]`. Note that we have indexed into the array using the position of the 3rd dimension and all rows and all columns (see Section 1.9 for more details about how to make subsets). We can create this plot with

```

grays = rev(gray.colors(10))
image(z = imageArray[ , , 1], col = grays,
      xaxt = "n", yaxt = "n")

```

We see in Figure 2.6 that it appears that the image is of the digit 4, except that the *y* coordinates are arranged incorrectly. We can fix this by plotting `imageArray[, 28:1, 1]`, i.e., we reorder the *y* values in each column.

2.4.1.1 Applying Functions to Matrices and Arrays

We have checked that the first matrix in our array of matrices looks like the digit 4, but what about the rest of them? As mentioned earlier, we cannot simply study a statistical summary of these pixel values in an attempt to validate our data. Instead, we can create a unique summary, such as a plot of the average of each pixel across the 1000 images. To calculate this average, we use the `apply()` function. The `apply()` function is designed for matrices and arrays. With it, we can apply the provided function across different dimension(s). In this case, we want to take the mean for each row and column entry across the 1000 ‘pages’ of images so we specify the `MARGIN` argument as `1:2`, i.e.,

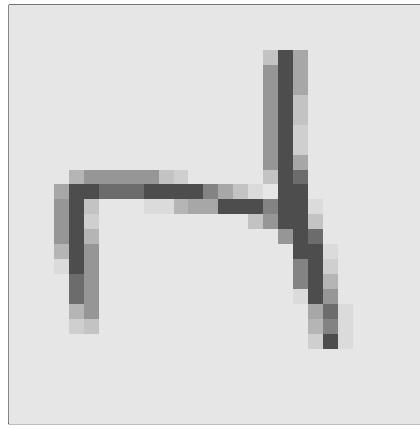


Figure 2.6: The 28×28 Pixel Image of a Handwritten Digit. *This plot is created from the pixel values for an image of a hand written numeral 4. The arrangement of the pixel values in the 28×28 matrix is incorrect. We can see that the y-values run in the wrong direction.*

```
avg4 = apply(imageArray, MARGIN = 1:2, FUN = mean)
```

We confirm that the result is a single 28×28 matrix, i.e.,

```
class(avg4)
[1] "matrix"
```

```
dim(avg4)
```

```
[1] 28 28
```

A plot of `avg4` appears on the left side of Figure 2.7. Although faint, we see that the ‘average’ image looks like the digit 4. Given the large number of 0 values in the pixels (we saw that over 3/4 of the pixel values are 0), we may want to summarize the pixel values with a statistic other than the mean. As an example, we calculate the upper quartile with

```
uq4 = apply(imageArray, 1:2, quantile, probs = 0.75)
```

The image of the upper quartile of the values for each pixel appears on the righthand side of Figure 2.7. We can see the digit 4 quite clearly.

2.5 Reshaping Data Tables

We may find that the organization of the data is not conducive to a particular analysis, and we want to reshape the data. For example, we may want to stack columns of data together to produce a longer data frame with more rows and fewer columns. Reciprocally, we may determine that analysis is better suited to a wider organization of the data, and we break

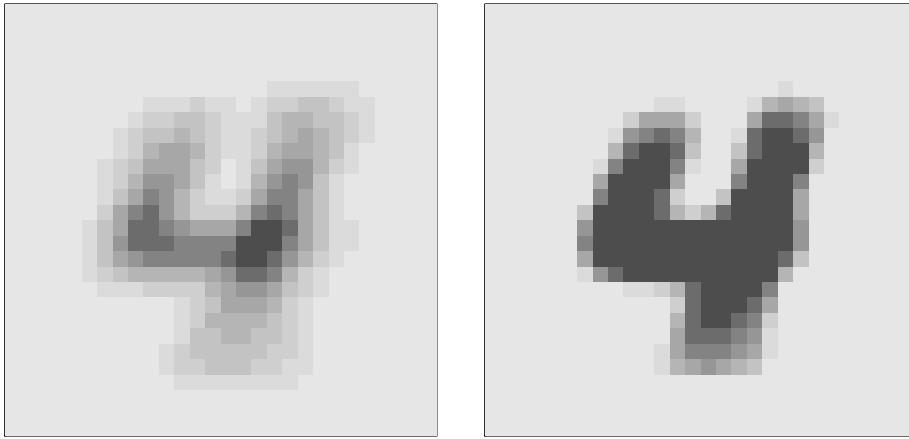


Figure 2.7: Image Plot of 1000 Digits. *The lefthand plot shows the average pixel value for each pixel across 1000 images. The plot on the right shows the upper quartile of the 1000 values for each pixel in the 28-by-28 pixel image.*

columns up into multiple columns with fewer rows. We demonstrate this reshaping with two examples. In Section 2.5.1, we create a longer, narrower data frame from the PeMS traffic data (see Q.2-1 (page 52)), and in Section 2.5.2, we create a wider, shorter data frame from the World Bank country data (see Q.2-3 (page 52)).

2.5.1 Stacking Traffic Flow

The PeMS data provide flow and occupancy over a loop detector in 6 distinct variables for 3 lanes of traffic. These data consist of 3 types of measurements: flow, occupancy, and the lane in which the measurements were made. However, with the current organization, the information about the freeway lane is found in the variable name, not in its own column in the data frame. This organization can be useful, if, for example, we want to compare the flow of traffic in the various lanes. We can make a scatter plot where a point corresponds to the flow for a particular 5-minute interval for the right and left lanes. We make the scatter plot and add a reference line with

```
plot(flow1 ~ flow3, data = traffic, pch = 19, cex = 0.3,
     ylim = c(0, 200), ylab = "", xlab = "",
     main = "Flow in Left Lane vs Right Lane")
abline(a = 0, b = 1, lty = 2, lwd = 3, col = "pink")
```

This plot and a similar plot for the flows in the center and right lanes appear in Figure 2.8. When we compare the locations of the points to the line, we see that the center lane consistently has more vehicles than the right. On the other hand, the left lane has fewer cars than the right when traffic is light, but when traffic is heavy (more than about 50 cars per 5 minutes), then the left lane serves many more vehicles.

For other analyses and plots, this arrangement of flow and occupancy in 6 variables can be inconvenient. For example, if we want to compute summary statistics for flow regardless of lane, then we need to concatenate the 3 vectors in the data frame together, e.g., `summary(~c(traffic$flow1, traffic$flow2, traffic$flow3))`. Instead, if the 3 columns

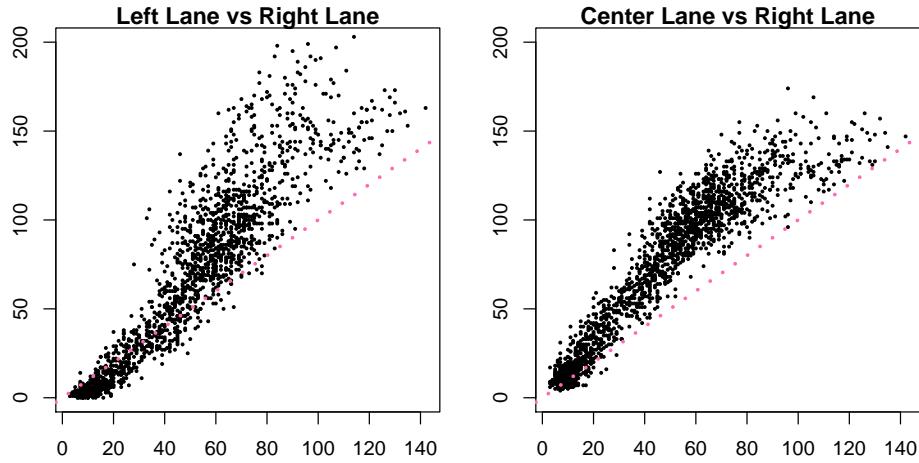


Figure 2.8: Comparison of Traffic Flow in Different Freeway Lanes. *The scatter plot on the left compares flow in the left and right lanes. We see that when throughput is high, the left and center lanes serve more vehicles than the right. Also, it appears that when flow in the right lane exceeds 60, the flow in the other two lanes is more variable.*

are combined into 1, i.e., stacked one on top of the other into `flow`, then we simply call `summary(flow)`. Additionally, if we have a variable that tracks the lane in which the measurements were taken, then we can make lane comparisons with formulas and grouping arguments to the plotting functions. We reformat the data and place the 3 vectors that contain flow measurements one above the other with

```
flow = stack(traffic[ , c("flow1", "flow2", "flow3")])$values
```

The return value from the call to `stack()` is a two-column data frame with vectors `values` and `ind`; the `values` vector contains the concatenated flow measurements and `ind` contains the name of the original variable (`flow1`, `flow2`, and `flow3`) to indicate the original variable the value comes from. We keep only `values` because we plan to create `lane`, an equivalent but more informative version of `ind`. Similarly, we create `occ` with

```
occ = stack(traffic[ , c("occ1", "occ2", "occ3")])$values
```

Then, we create the lane ‘key’ as a `factor`; we do this with

```
lane = factor(rep(c("left", "middle", "right"),
                  each = nrow(traffic)))
```

We can combine these variables into a single data frame, and include `time` as well with

```
trafficLong = data.frame(lane, flow, occ,
                        time = rep(traffic$time, 3))
```

We confirm that the data are arranged correctly by examining the first and last few rows in `trafficLong` with `head()` and `tail()`; we find,

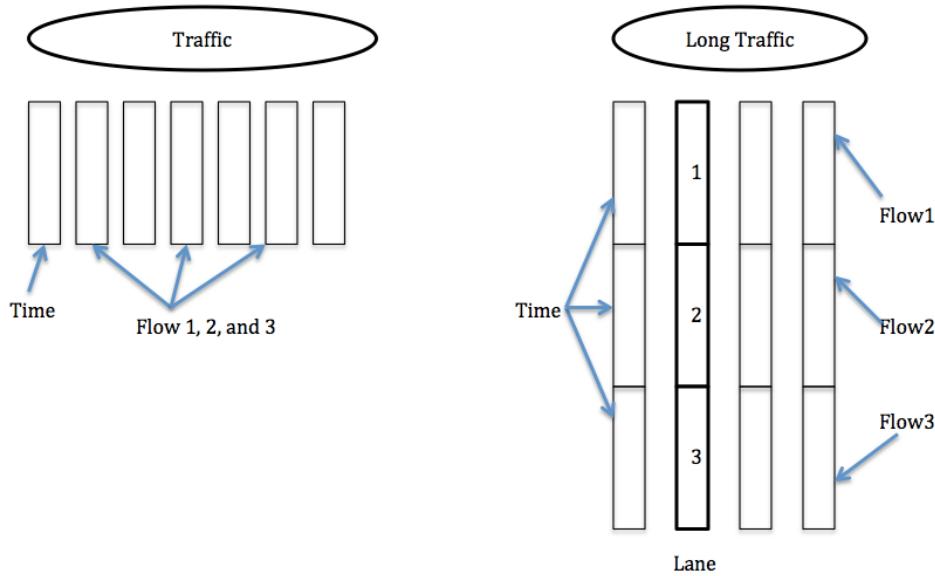


Figure 2.9: Diagram of Stacking Tables. The `traffic` data frame (left) contains 3 variables with flow information, one variable for each lane. The data frame on the right is a ‘stacked’ version, where these 3 variables are combined into one. This data frame also has a stacked version of occupancy. We need to create a new variable (`lane`) so that we can keep track of the lane for flow and occupancy. Similarly, we need to stack 3 copies of `time`.

```

    lane flow      occ          time
1 left   14 0.0100 2003-03-14 00:00:00
2 left   18 0.0133 2003-03-14 00:05:00
3 left   12 0.0088 2003-03-14 00:10:00
4 left   16 0.0115 2003-03-14 00:15:00
5 left   8 0.0069 2003-03-14 00:20:00
6 left   11 0.0077 2003-03-14 00:25:00
...
    lane flow      occ          time
5215 right 18 0.0199 2003-03-20 00:30:00
5216 right 9 0.0059 2003-03-20 00:35:00
5217 right 18 0.0234 2003-03-20 00:40:00
5218 right 13 0.0206 2003-03-20 00:45:00
5219 right 8 0.0063 2003-03-20 00:50:00
5220 right 12 0.0105 2003-03-20 00:55:00

```

It appears that our long data frame has been correctly constructed.

An example of the benefit of the long version of the traffic data frame is in the creation of the line plot in Figure 2.10. We create this from `trafficLong` with the `xyplot()` function in the `lattice` package, e.g.,

```

library(lattice)
xyplot(flow ~ time, data = trafficLong, groups = lane,
       type = "l", xlab = "", ylab = "Flow",

```

```
auto.key = list(corner = c(0.2, 1), points = FALSE,
               lines = TRUE))
```

Notice how the formula `flow ~ time` expresses the desired relationship; that is, the formula indicates that we want to visualize how flow depends on time. We also specify `groups = lane` to indicate that we want to compare this relationship across lanes, and as a result, we obtain 3 (color-coded) line plots in the plotting region. This plot reveals structure in the data that is not evident in Figure 2.8 because here we have the opportunity to observe the relationship between flows in all 3 lanes in time. Specifically, we see the ordering of flow between the 3 lanes change with time. In heavy traffic, the left lane has the greatest flow and the right lane is lowest, but immediately after a peak in flow, the left lane's flow dips sooner and faster to levels below the middle lane.

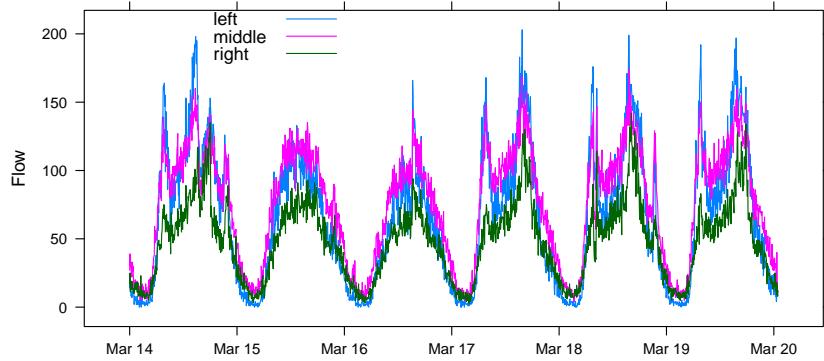


Figure 2.10: Comparison of Traffic Flow Across Lanes Over Time. *These three line plots show the patterns in throughput for 3 lanes of traffic. The 3 lanes have the same general shape, but the time of the peaks differ slightly and the swings from high to low are greater for some lanes than others.*

An alternative approach to stacking the columns uses the `gather()` function in the `tidyverse` package [12]. The `gather()` function performs the same task as `stack()` but can be more convenient with its `key` and `value` arguments that allow us to name the vectors in the resulting data frame. We do not take advantage of this feature here because we prefer to create our own factor with labels: left, center, and right. In any event, we can use `gather()` as follows:

```
library(tidyverse)
flow = gather(traffic[, c("flow1", "flow2", "flow3")])$value
occ = gather(traffic[, c("occ1", "occ2", "occ3")])$value
```

We have introduced `gather()` because other functions in `tidyverse` can be quite convenient; in particular the `spread()` function helps us create a wider, shorter data frame in Section 2.5.2.

2.5.2 Rearranging World Bank Country Statistics

The data in `GenderStat_Data.csv` come from the World Bank's Open Data Website described in Q.2-3 (page 52). At that site we can choose from a collection of prepared data

files or use an interface to select particular variables and countries of interest. We downloaded the collection of gender related variables in csv format. These data include annual measurements from 1960 through 2015.

To get a sense of the format of the data, we can view it in a plain text editor, or we can open it in Excel. With Excel, the software arranges the values into a spreadsheet, much as `read_delim()` arranges values into a data frame. The spreadsheet view of the data can be easier for us to examine the organization of the data than a plain text editor or data frame in R because the columns are aligned. Also, the file is too large for the data viewer in RStudio.

A screenshot of a portion of the Excel spreadsheet appears in Figure 2.11. We immediately notice several characteristics that are important to working with and analyzing these data: a) some of the entries for countries are actually conglomerates, e.g., Arab World; b) the column labeled 'Indicator Name' contains what appear to be variable names so one country occupies many rows (one row for each variable); c) the columns labeled 1960, 1961, etc. contain the annual values of the variable listed under Indicator Name; d) there appear to be many empty cells. For this example, we simplify the data and concern ourselves with one year only. We pick 2010 because it is relatively current and, by observation, appears to have fewer empty cells.

	A	B	C	D	E	F	G	BB	BC	BD
1	"Country Name"	Country Code	Indicator Name	Indicator Code	1960	1961	1962	2009	2010	2011
2	Arab World	ARB	Access to anti-re SH.HIV.ARTC.FE.ZS							
3	Arab World	ARB	Access to anti-re SH.HIV.ARTC.MA.ZS							
4	Arab World	ARB	Account at a fina WP_time_01.3							
5	Arab World	ARB	Account at a fina WP_time_01.2							
6	Arab World	ARB	Adolescent ferti SP.ADO.TFRT	133.560907	134.164419	134.861018	49.803885	49.7035477	49.5798955	
7	Arab World	ARB	Age at first marr SP.DYN.SMAM.FE							
8	Arab World	ARB	Age at first marr SP.DYN.SMAM.MA							
9	Arab World	ARB	Age dependency SP.POP.DPND	87.7976012	89.2206214	90.5019657	62.7324195	62.045774	61.7973355	
10	Arab World	ARB	Age population, SP.POP.AG00.FE.IN							
11	Arab World	ARB	Age population, SP.POP.AG00.MAIN							
12	Arab World	ARB	Age population, SP.POP.AG01.FE.IN							
13	Arab World	ARB	Age population, SP.POP.AG01.MAIN							
14	Arab World	ARB	Age population, SP.POP.AG02.FE.IN							
15	Arab World	ARB	Age population, SP.POP.AG02.MAIN							
16	Arab World	ARB	Age population, SP.POP.AG03.FE.IN							
17	Arab World	ARB	Age population, SP.POP.AG03.MAIN							
18	Arab World	ARB	Age population, SP.POP.AG04.FE.IN							
19	Arab World	ARB	Age population, SP.POP.AG04.MAIN							
20	Arab World	ARB	Age population, SP.POP.AG05.FE.IN							
21	Arab World	ARB	Age population, SP.POP.AG05.MAIN							
22	Arab World	ARB	Average number SL.TIM.DWRK.FE							
23	Arab World	ARB	Average number SL.TIM.DWRK.MAIN							
24	Arab World	ARB	Birth rate, crude SP.DYN.CBRT.IF	47.6978881	47.4553181	47.1985791	27.0762781	27.0921925	27.038467	
25	Arab World	ARB	Births attended I SH.STA.BRTC.ZS							77.6463271

Figure 2.11: Excel Spreadsheet of World Bank Data. *This screen shot displays a portion of a World Bank csv file that has been opened in Excel.*

Even with this simplification, the data have an unusual organization, i.e., the variable names are in one column rather than as names across the top of several columns. To get a better idea as to how we might reshape our data for analysis, we consider a simplified version of this format. In the data frame below, we have 3 countries (A, B, and C) and 2 variables (V1 and V2). Each country has 2 records in the data frame, one for each variable. The column, `vals`, contains the measurement of a variable for a country, and we can tell

which variable the measurement is associated with by examining the contents of `varName`, e.g., country C has a value of 29 for variable V2.

```
ctry varName vals
1   A      V1    11
2   A      V2    20
3   B      V1    12
4   B      V2    24
5   C      V1    14
6   C      V2    29
```

We want to turn this data frame into a wider, shorter data frame where there is one row for each country, and a column for each variable (we want variables `ctry` and `V1` and `V2`). The `spread()` function in the `tidyverse` package can do exactly this with

```
spread(x, key = varName, value = vals)
```

```
ctry V1 V2
1   A 11 20
2   B 12 24
3   C 14 29
```

Now we have a 3×3 , rather than 6×3 , data frame.

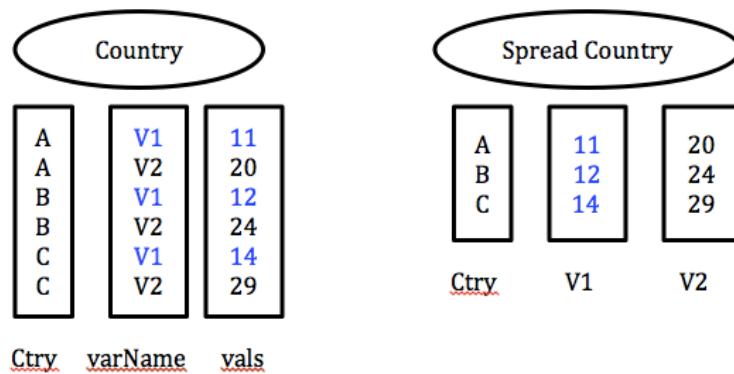


Figure 2.12: Conceptual Diagram of a 3-Dimensional Array. *This diagram shows a conceptual representation of a 3-dimensional array for handwritten digits. It is a collection of 1000 matrices that have 28 rows and 28 columns. Each matrix represents the pixel values for one handwritten image.*

We can apply this same technique to our World Bank data. We begin by reading the data into `R` with `read_delim()`, i.e.,

```
wb = read_delim("worldBank/GenderStat_Data.csv", delim = ",")
```

Then we drop the first 16534 rows because they contain aggregate data for regions rather than countries. We do this with subsetting by exclusion, e.g.,

```
wb = wb[-(1:16534), ]
```

We want to reshape our data as we did with the toy example. To do this, we need to identify the columns corresponding to the country name, variable name, and variable values. From the spreadsheet, we see that these are "Country Code", "Indicator Name", and "2000". We call `spread()` with the reduced `wb` data frame as follows:

```
wbWide = spread(data = wb[, c("Country Code", "Indicator Name",
                             "2000")],
                 key = "Indicator Name", value = "2000")
```

An alternative approach uses the `select()` function in `tidyverse`, rather than the square-bracket subsetting of `wb`, e.g.,

```
spread(data = select(wb, c(2, 3, 45)),
       key = "Indicator Name", value = "2000")
```

Or, we can also use the `one_of()` helper function in `dplyr` to specify the column names rather than positions, e.g.,

```
spread(data = select(wb, one_of("Country Code",
                                 "Indicator Name", "2000")),
       key = "Indicator Name", value = "2000")
```

Some users find the `select()` and `one_of()` functions in `dplyr` easier to use as they do not involve the `[` operator. However, we will see in Section 4.2.1 that `[` is more versatile.

Let's compare the shapes of `wb` and `wbWide` with

```
nrow(wb)
[1] 107213

dim(wbWide)
[1] 214 502
```

The variable names in `wbWide` are cumbersome to work with as they are long and include blanks and special characters, e.g., the first 6 variable names are:

```
head(names(wbWide))
[1] "Country Code"
[2] "Access to anti-retroviral drugs, female (%)"
[3] "Access to anti-retroviral drugs, male (%)"
[4] "Account at a financial institution, female (% age 15+) [ts]"
[5] "Account at a financial institution, male (% age 15+) [ts]"
[6] "Adolescent fertility rate (births per 1,000 women ages 15-19)"
```

Let's rename the variables to shorter names that are easier to type.

For simplicity, we also limit the variables to explore. We are interested in gross domestic product (in column 200), health expenditure per capita (214), life expectancy (259), literacy rate of adult female (260), infant mortality rate (289), total population (340), prevalence of female obesity (347), and public spending on education (374). And, of course, country code (1). We create a vector of short names for these variables, and we create a subset that contains only these few variables. We do this with

```
varNames = c("ctry", "gdp", "healthPC", "lifeExp", "litRateF",
           "infantMort", "pop", "obesityFemale", "edSpending")

wbSub = wbWide[ , c(1, 200, 214, 259, 260, 289, 340, 347, 374) ]
names(wbSub) = varNames
```

We view the first few records in this data frame to confirm that our subsetting worked as expected, e.g.,

```
head(wbSub)

Source: local data frame [6 x 9]

  ctry      gdp healthPC lifeExp litRateF infantMort      pop
  (chr)    (dbl)    (dbl)    (dbl)    (dbl)    (dbl)    (dbl)
1 ABW 1.873e+09       NA  73.72  97.07       NA  90858
2 ADO 1.402e+09 1954.98       NA       NA       3.9  65399
3 AFG       NA       NA  55.13       NA  95.4 19701940
4 AGO 9.130e+09   91.14  45.20       NA 128.3 15058638
5 ALB 3.632e+09  248.44  74.27       NA  23.2 3089027
6 ARE 1.043e+11 1882.34  74.45       NA      9.6 3050128

Variables not shown: obesityFemale (dbl), edSpending (dbl)
Source: local data frame [6 x 8]
```

It appears that we have properly read and reshaped the data and that many countries have a missing literacy rate.

We examine the summary statistics for all variables with

```
summary(wbSub)

  ctry          gdp          healthPC
Length:214      Min. :1.37e+07      Min. :  6
Class :character 1st Qu.:1.73e+09  1st Qu.: 93
Mode  :character Median :8.04e+09  Median :307
                  Mean  :1.66e+11  Mean   :665
                  3rd Qu.:5.22e+10 3rd Qu.: 754
                  Max. :1.03e+13  Max.  :4818
                  NA's  :16        NA's   :28

  lifeExp         litRateF        infantMort
Min.  :38.7      Min.  :12.8      Min.  :  3.1
1st Qu.:60.0     1st Qu.:55.4     1st Qu.: 10.9
Median :70.3     Median :81.6     Median : 26.7
Mean   :67.0     Mean   :72.9     Mean   : 39.7
3rd Qu.:74.5     3rd Qu.:92.3     3rd Qu.: 64.5
Max.   :81.1     Max.   :99.8     Max.   :143.3
NA's   :13        NA's   :171      NA's   :22

  pop          obesityFemale    edSpending
Min.  :9.42e+03  Min.  : NA      Min.  : 0.80
1st Qu.:6.20e+05 1st Qu.: NA      1st Qu.: 3.00
Median :5.10e+06  Median : NA      Median : 4.17
Mean   :2.85e+07  Mean   :NaN     Mean   : 4.38
3rd Qu.:1.65e+07 3rd Qu.: NA      3rd Qu.: 5.44
Max.   :1.26e+09  Max.   : NA      Max.   :11.49
NA's   :214       NA's   :214     NA's   :93
```

From this numerical output, we make several observations about the data: all values for `obesityFemale` are missing, over 3/4 of `litRateF` is missing, and about half of `edSpending` is missing; the range of `gdp` and `pop` cover 6 orders of magnitude, and the range in `healthPC` spans 3 orders of magnitude; country code is character and since it is unique across rows we do not convert it to a `factor`.

We can get a better sense of the distributions of the variables that have only a few missing values by examining density plots. For example, let's explore life expectancy. From the summary statistics, we see that life expectancy ranges from 38 to 81 with half of the countries having a life expectancy over 70. We make a density plot of life expectancy with

```
library(ggplot2)
ggplot(data = wbSub, aes(x = lifeExp)) + geom_density() +
  geom_rug(col="darkred", alpha = .4) +
  labs(x = "Life Expectancy") + xlim(c(35, 85))
```

Figure 2.13 reveals a skew-left distribution with the great majority of countries having a life expectancy of 70 to 80 years. Yet in some countries, average life expectancy falls well below 65 years. The rug plot along the x-axis allows us to see the concentration of values for the countries. We continue our exploration of these data in Section 2.7.2.

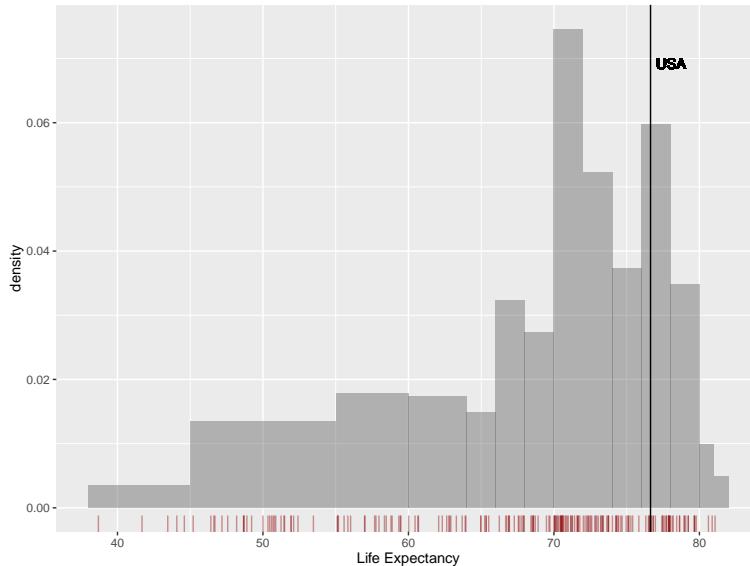


Figure 2.13: Average Life Expectancy for 214 Countries. *This histogram of average life expectancy shows a highly skewed distribution with many countries having a life expectancy below 65. The average life expectancy in the US is marked by a vertical line at 76. The individual country values are displayed in the rug plot below the histogram. These data are for the year 2000 as reported by the World Bank.*

2.6 Merging Data Tables

At times the variables that we want to analyze are not contained in a single data table and we need to merge two (or more) tables together to obtain a structure suitable for analysis. The variables may be divided across tables so some information is contained in one table and other information appears in a second table. In order to use information in both tables, we need to combine them into one table. However, the observations may not line up across tables, meaning that the order of the observations may be different from one table to the other. In order for us to correctly consolidate the variables into one table, we need a way to match observations between the tables. Typically, we have a uniquely identifying ‘key’ for each observation, and we use this key to match the observations across tables. (See Figure 2.14.)

A further complication sometimes arises where the two tables do not contain the exact same observations. That is, there is overlap in the records, but each table may contain observations (or keys) that do not appear in the other. When this happens, we need to determine whether we want only those observations that appear in both tables, or we want all those that appear in one table even if that means there will be missing values for the records where we do not find a match. As long as we have a key to match the records we can merge the data in any of these ways (see Figure 2.14).

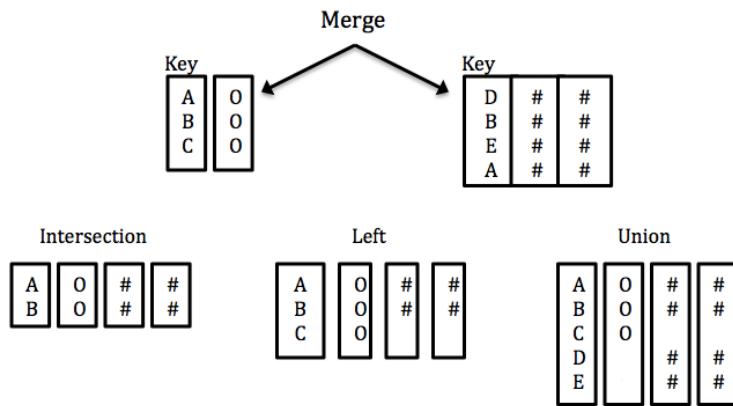


Figure 2.14: Diagram of Types of Merges. *The two data frame in the top row can be merged in several ways. The key for merging is the first column in each data frame. The merged data frame can contain only records where a match is found in both (bottom left), records only from the left data frame (bottom middle) or records from either data frame (bottom right). Missing values (blanks in the diagram) occur when a key appears in one file but not the other.*

We mention one last type of merge. We can have a case where the key is unique in one file but not in the other. For example, with the Clinton email data, we find in the next section that there is a 2-column table that contains an identification number and name for all individuals who sent or received an email. There is one record in this file for each identification number (or key). In the original table that we examined, we have an identification number for the sender of the email. This file has one record per email, and many people send more than one email so it’s possible for there to be many records with the same identification number. Indeed we saw that Hillary Clinton sent thousands of emails,

and her identification number (80) appears thousands of times in this file. Although the key is not unique in the email table, we can merge these 2 tables together in order to add the sender's name to email. This is one of the data merges that we carry out in the next section.

2.6.1 Merging Names and Emails

We discovered in Section 2.3.3 that the variable `MetadataFrom` was not a good source for the name of the email sender. When we wanted to count how many emails each person sent, we did not get accurate counts because some individuals had multiple representations of their name. Instead, it is more accurate to use the identification number that is also supplied in that table. That is, we can tally the emails and examine the top 10 senders with

```
sort(table(emails$FrId), decreasing = TRUE)[1:10]
```

80	81	32	87	194	116	170	124	10	180
1993	1437	1318	871	374	341	159	155	131	118

In order to figure out who is person #194, we need to do some sort of look up. We can look at the values for `MetadataFrom`, e.g.,

```
head(emails$MetadataFrom[ emails$FrId == 194 ])

[1] NA                 NA                 NA
[4] "Blumenthal, Sidney" NA                 "Blumenthal, Sidney"
```

Some of the entries are NA while others list Sidney Blumenthal as the sender. We see again why `MetadataFrom` is not very clean.

The file we downloaded from Kaggle was a zipped file that contained several csv files. In addition to `Emails.csv` that we read into *R* as the `emails` data frame, the zipped file also contained `Persons.csv`, which has a unique name for each identifier, and `EmailReceivers.csv`, which we describe shortly. When we view `Persons.csv` in a plain text editor (or Excel or RStudio viewer) we see that it contains 2 variables, `Id` and `Name`. When we scroll though this file, we see that there is one name for each identification number. We can match the identification number used in an email record to the identification number in this file and pick up the person's name to add to the record. Then we can tally the emails by this clean version of the sender's name.

To do this, we read in `Persons.csv` with

```
persons = read_csv("Persons.csv", col_names = TRUE)
str(persons)

Classes 'tbl_df', 'tbl' and 'data.frame': 513 obs. of 2 variables:
 $ Id : int 1 2 3 4 5 6 7 8 9 10 ...
 $ Name: chr "111th Congress" "AGNA USEMB Kabul Afghanistan" ...
```

We confirm that the unique identifier is called `Id` and the individual's name is held in `Name`. Next, we use `merge()` to combine the `emails` and `persons` data frame with

```
emails = merge(emails, persons, by.x = "FrId", by.y = "Id",
               all.x = TRUE)
```

In addition to providing `merge()` with the 2 data frames, the arguments we have specified give the names of the variables in each data frame that contain the key to merge on and the type of merge to perform. The variable `FrId` in `emails` and `Id` in `persons` hold the unique identifiers. Since `emails` is the first data frame provided in the function call, it is referred to as `x` in `by.x`, and `by.y` refers to the second data frame, i.e., `persons`. The expression `all.x = TRUE` indicates that we want to keep all of the records in `emails` so the value in `Name` in `persons` is added to all records where we find a match in the keys. When a record has a value for `FrId` that doesn't match any value in `Id`, then NA is assigned to `Name` for that record.

After merging the data frames, when we tally the emails, we can use `Name`. We find that the top email senders are

```
sort(table(emails$name), decreasing = TRUE)[1:6]
      Hillary Clinton          Huma Abedin          Cheryl Mills
                  1993                  1437                  1318
      Jake Sullivan    Sidney Blumenthal        Lauren Jiloty
                  871                   374                   341
```

Now, all of the emails corresponding to Jacob J Sullivan are counted together and appear under the label 'Jake Sullivan' and similarly for Sidney Blumenthal, etc.

What happens when we try to merge these two data frames and specify `all.y = TRUE` rather than `all.x = TRUE`? Or, what if we include both arguments, i.e., `all.x = TRUE, all.y = TRUE`? Try it and examine the dimension of the resulting data frames to convince yourself of how the merge is working. Be careful not to assign the merged file to `email` because we can get unwanted records. For example, we can have a record where all of the variables except `Name` are NA. Who are these people? The `persons` data frame contains identifiers and names of people who sent or received emails so anyone who received an email from Clinton but did not send an email to her will have a record added to the resulting data frame. All of the variables in this record (except `Name`) are missing.

Let's take our investigation one step further and consider the senders and recipients of email. We are interested in who sends and who receives emails and how often any two people exchange email. In other words, we're interested in the connections between people. The file `EmailReceivers.csv` contains the recipients of each email. Let's read in this file and examine it

```
recipients = read_csv("EmailReceivers.csv", col_names = TRUE)
```

We can examine a few records in our file and find emails with multiple recipients. For example,

```
recipients[16:25, ]
   Id EmailId PersonId
   (int) (int)     (int)
1    16     15       80
2    17     16       80
3    18     17       80
4    19     17       32
5    20     17      229
6    21     17      170
7    22     17       87
8    23     18       80
9    24     18       32
10   25     18      230
```

The variable `EmailID` holds the identifier for the email record in `emails`. Notice that there are 5 entries in the `recipients` data frame that have a value of 17 for `EmailID`. For these records, we have 5 different values for `PersonId`, including #80, #32, and #87 whom we already know from our previous investigation are Clinton, Mills, and Sullivan, respectively. We can examine the `MetadataTo` and find that it shows only H for Clinton, but the `ExtractedTo` variable has the string:

```
H; Mills, Cheryl D; Sullivan, Jacob J; Nuiand, Victoria J;
Reines, Philippe
```

These 5 recipients all appear in the string.

What kind data structure do we need in order to track who sent and received email to whom? What if we take the `recipients` data frame and augment it with the sender information? We can also add the name of the recipient, just as we added the name of the sender to `email`. We want to merge `recipients` and `emails`, which we can think of as senders. Let's start by creating a `senders` data frame that contains only the information we need in the merge. This is, the email identifier and the sender's identifier and name, we can do this with

```
senders = emails[ , c("Id", "FrId", "Name")]
```

Now `senders` is a 3-column data frame. Recall that `recipients` has 3 variables, `Id`, `EmailID`, and `PersonId`. To avoid confusion, let's rename `PersonId` to `ToId` and drop `Id`. We can do this with

```
names(recipients)[3] = "ToId"
recipients = recipients[ , -1]
```

We can add the person's name to this file with a merge, similar to the merge we already performed between `emails` and `persons`, i.e.,

```
recipients = merge(recipients, persons,
                    by.x = "ToId", by.y = "Id", all.x = TRUE)
```

Again, we have 2 data frames at different levels of granularity. The `recipients` is at the level of email recipient and `senders` (and `emails`) is at the level of email message. Since an email message can have multiple recipients and only one sender, we have more rows in `recipients` than in `senders`. Again, we want to merge the data in such a way as to keep all records in `recipients` and add to each record the sender ID and name. We can do this with

```
ToFrs = merge(recipients, senders,
              by.x = "EmailID", by.y = "Id", all.x = TRUE,
              suffixes = c("To", "Fr"))
```

We added one more argument to the call to `merge()`, i.e., `suffixes = c("To", "Fr")`. This argument is used to differentiate between variables with the same name in the 2 files. We have `Name` in both files, and after the merge, we have `Name.To` for the recipient's name and `Name.Fr` for the sender's name.

Now that our data are merged into 1 data frame, we can tally the communications between the people, i.e.,

```
emailCts = with(ToFrs, table(NameFr, NameTo))
```

It's difficult to study these data because `emailCts` has more than 100 rows and columns. Researchers developed a 2-dimensional line plot called BioFabric [1] where individuals are represented by horizontal lines and vertical lines represent exchanges between these individuals (an *R* implementation is available in the `RBioFabric` package [6]). Figure 2.15 is an example of this type of plot. What we can see here is that Clinton exchanges email with just about everyone (which makes sense since it is her email that we are studying), and her staff seem to be on email exchanges with smaller subgroups of people. Later, in [?] we consider these and other advanced visualizations. We include it here to hint at the possibilities in designing specialized visualizations for particular data structures.

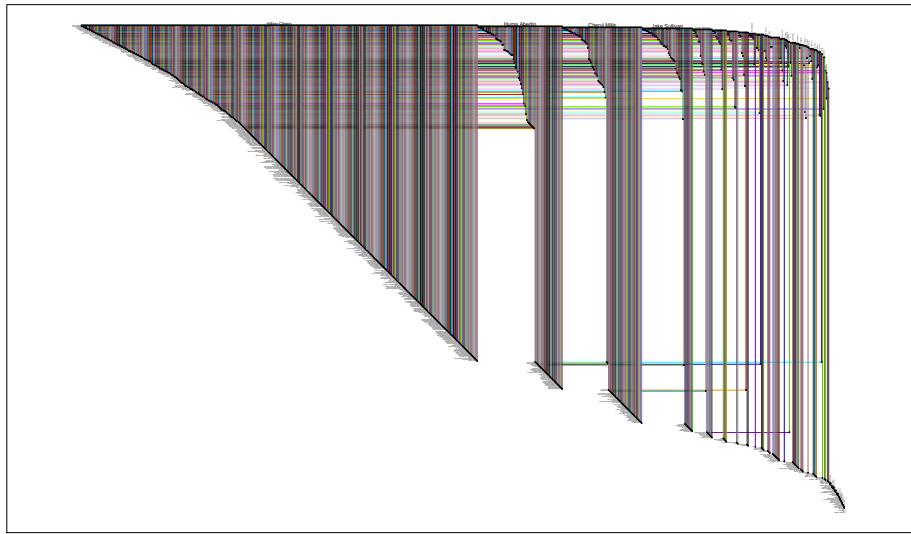


Figure 2.15: Clinton Emails Sender and Recipients. *This plot was designed to represent complex graphs. A node in the graph appears as a horizontal line segment, and edges between nodes are represented as vertical line segments that terminate at the two rows associated with the endpoint nodes.*

The `dplyr` package offers the functionality of `merge()` that employs the terminology common to relational databases. For example, the `full_join()` function finds the union of the two data frames, i.e., it works like `merge()` when both `all.x` and `all.y` are TRUE, and `inner_join()` is equivalent to them both being FALSE. Similarly, when only one of these arguments is TRUE, then we are performing a `left_join()` (`all.x = TRUE`). The `dplyr` package also offers other ways to join 2 data frames together. We examine how to merge tables in relational databases in Chapter 9.

2.7 Exploratory Data Analysis

We have provided examples of exploratory data analysis (EDA) throughout this chapter as we have read and cleaned various data sets. This section contains a more comprehensive treatment of EDA. EDA is like detective work. We aim to have an open mind about what me

might uncover and a willingness to find something surprising that points us in a direction for analysis that we had not previously considered. We examine the structure in the data in an iterative fashion; as we uncover aspects of our data, this can lead us to re-examine our understanding of the data and continue in our exploration. This approach typically relies on plotting our data in multiple ways, e.g., changing the scale to zoom in or out, transforming a variable to straighten or flatten, and consider a different type of plot to see the data from another view point. With EDA we try to remain flexible in the techniques that we use to examine the data. According to Tukey, the founder of EDA, “exploratory data analysis is actively incisive rather than passively descriptive, with real emphasis on the discovery of the unexpected.” He further explains that “‘Exploratory data analysis’ is an attitude, a state of flexibility, a willingness to look for those things that we believe are not there, as well as those we believe to be there.” Importantly, we also consider the manner in which the data were collected and how this can impact the conclusions drawn from our explorations. In the next sections, we explore more deeply the traffic data, country statistics, and emergency room survey results as examples of how we might carry out EDA.

2.7.1 Exploring Traffic on California Freeways

Recall from Q.2-1 (page 52) that we have an extract of traffic data from PeMS (the California freeway Performance Measurement System). These data consist of 5-minute summaries of the number of cars (flow) passing over a loop detector and the proportion of time a car was above the loop (occupancy). These 5-minute summaries are measured over 6 days for one set of loop detectors on 3-lanes of a freeway. In Section 2.2.1.1 we read the data into a data frame with 7 variables: flow and occupancy for the 3 lanes of traffic, plus time. We also changed the type of `time` from `character` to `POSIXct` so that we can use it in plotting. In Section 2.5.1 we re-shaped this data frame by stacking the measurements of flow for the 3 lanes into one variable (`flow`) and the 3 sets of occupancy values into one variable (`occ`). We also created a new variable (`lane`) so that we can distinguish which measurements came from which lane. With this format, we separated the information about the lane in which the traffic is measured from the variable name and created a variable that we can use more easily in analysis (see Figure 2.9). The `trafficLong` data frame contains the results from this data manipulation. It consists of 4 variables, `time`, `lane`, `flow`, and `occ`. As we carried out these actions on the data frame, we made several plots. In this section, we continue to examine the traffic data, but our focus is on exploration rather than data validation.

The main variables of interest are `flow` and `occ` because we are interested in `time` and `lane` only to help explain the relationship between flow and occupancy. As a first step, we examine the distribution of flow and of occupancy. Occupancy can be viewed as a measure of congestion. Since it is a numeric variable we can summarize it with

```
summary(trafficLong$occ)
```

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
	0.0000	0.0183	0.0466	0.0579	0.0680	0.5100

The mean is about 25% larger than the median indicating that the distribution is skewed right. To get a better sense of the shape of the distribution of occupancy values, we examine a density curve. The density curve is a smooth version of a histogram (see the plot on the left of Figure 2.16). We create this curve with

```
ggplot(data = trafficLong) +
  geom_density(mapping = aes(x = occ)) +
  labs(x = "Occupancy")
```

By examining a density curve, we can glean additional features about a distribution that are not from in the summary statistics. From a density curve we can see which values are more common (high density) or rare. Most noticeably, the density curve for occupancy shows a bimodal distribution with two sharp peaks at about 0.02 and 0.06. We also see that the distribution has a long right tail when occupancy reaches 0.20 – 0.40. To get a better view of the main portion of the distribution, we zoom in to the region from 0 to 0.15 and shrink the bandwidth, which controls the amount of smoothness of the curve, so it follows the data more closely. The resulting curve is displayed on the right in Figure 2.16. There the 2nd mode appears to contain 3 small peaks, but this may be simply an artifact of a small bandwidth, which tends to yield curves that follow the data values too closely. However, since there are 3 bumps, it raises the question of how occupancy varies with lane.

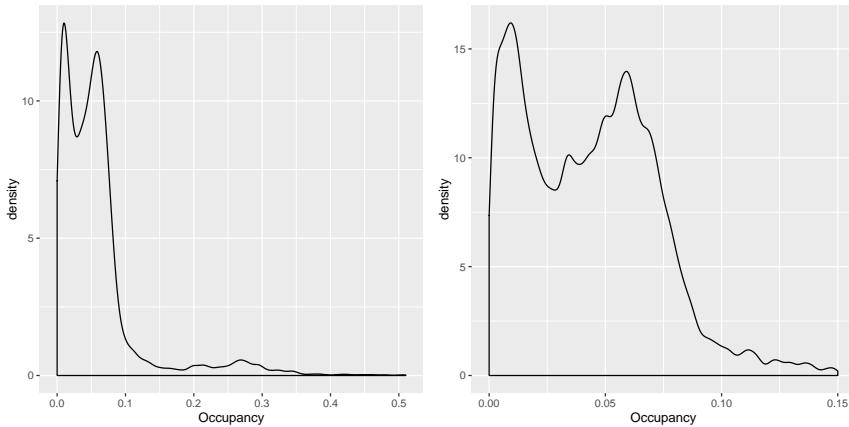


Figure 2.16: Distribution of Occupancy. *The density curve on the left shows a bimodal distribution for occupancy. The first mode is more peaked and the distribution has a long right tail. The plot on the right shows the distribution of occupancy over the range from 0 to 0.15. Here, the peakedness of the 1st mode and the larger spread in the 2nd mode are more apparent.*

In order to tease out the contributions from the 3 lanes to the distribution of occupancy, we create 3 density curves, one for each lane, and overlay them on the same plotting region, e.g.,

```
ggplot(data = trafficLong) +
  geom_density(mapping = aes(x = occ, fill = lane, color = lane),
               alpha = 0.3) +
  labs(x = "Occupancy") + xlim(c(0, 0.15))
```

This plot appears in Figure 2.17. We can see that each lane has a bimodal distribution, but the locations and peakedness of the modes differ. The middle lane tends to have a higher occupancy than the other lanes; we see this in the smaller first peak and in the location of the second peak at a higher occupancy than the other lanes. Additionally, the left lane appears to be more skewed than the other lanes with its large right tail. Further, the two modes in the right lane are closer together and the distribution is less spread.

Let's now examine the distribution of flow. Since the distribution of occupancy varied with the freeway lane, we examine flow as a function of lane too. The lefthand plot in Figure 2.18 displays side-by-side box plots of flow for the 3 lanes of traffic. These box and

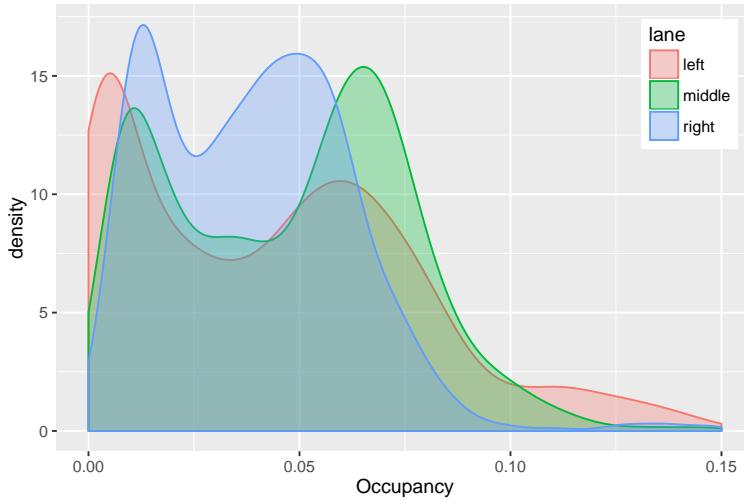


Figure 2.17: Distribution of Occupancy for 3 Lanes of Traffic. *This plot shows the distribution of occupancy separately for the 3 lanes of traffic. The distribution varies with lane where each lane exhibits 2 levels of occupancy with a sharper initial peak and a broader 2nd peak. Additionally, all 3 density curves have long right tailss.*

whisker plots provide a visual display of the quantiles of the data. The ends of the box mark the upper and lower quartiles, the line through the interior locates the median, and whiskers are drawn from the box to the largest data value within 1.5 of the inner quartile range (IQR) beyond the upper (or lower) quartile. Values beyond this are denoted individually. These plots reveal more information about a distribution than the summary statistics we computed earlier, but they are not typically as informative as a density curve or histogram because they cannot, for example, show modes and gaps. We do see that the flow in the right lane tends to be smaller than the flow in the other lanes as at least 3/4 of the values fall below the medians of the left and middle lanes. It also appears that the left lane has a long right tail and appears more skewed than the other lanes. The violin plot on the righthand side of Figure 2.18 is similar to the boxplot except that a density curve of the data (and its mirror image) is plotted along each vertical line. The violin plot reveals the bimodal nature of the distribution of flow. We make these two sets of plots with

```
boxFlow = ggplot(data = trafficLong) +
  geom_boxplot(aes(x = lane, y = flow)) +
  labs(x = "Lane", y = "Flow")
vioFlow = ggplot(data = trafficLong) +
  geom_violin(aes(x = lane, y = flow), bw = 5) +
  ylim(c(0, 200)) + labs(x = "Lane", y = "Flow")
grid.arrange(boxFlow, vioFlow, ncol = 2)
```

These density curves, box plots, and violin plots do not address questions about how flow in the lanes vary together, how flow varies over time, and how flow (which is considered a measure of throughput) and occupancy (regarded as a measure of congestion) vary together. In the first case, we can, for example, make a scatter plot of flow in the left and right lanes. We made this plot earlier (see Figure 2.8). In that plot, a point corresponds to a particular 5-minute time interval so there are 1740 points plotted. We saw that flow in the left and

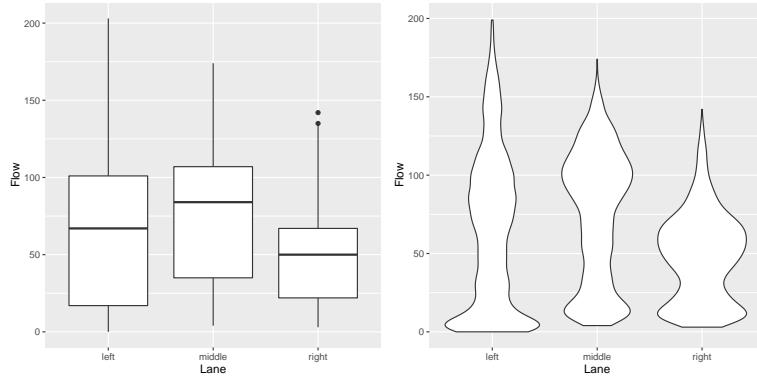


Figure 2.18: Distribution of Flow for 3 Lanes of Traffic. *The box plots on the left and the violin plots on the right compare flow for 3 lanes of traffic. We can see the bimodal nature of the distributions in the violin plot.*

right lanes are correlated in the sense that when flow is low in one lane then it tends to be low in the others. However, we also can see that flow in these two lanes is roughly linear with slope near 1 for flow under 50, and after that, the flow in the left lane increases more quickly than the right, and the variability also increases. This change is connected with the locations of the second modes in these distributions. Also the longer right tail of flow in the left lane is evident in the increase in variability at higher flow.

To understand how flow changes in time, we want to plot flow against time. We typically place time on the x-axis and make a line plot, e.g., we connect the flow measurement at one time to the measurement 5 minutes later and so on. Given the differences we have already observed between the behavior of flow across lanes, we super-pose the line plots of flow against time for each lane. We made this plot in Figure 2.10. As mentioned earlier, we see the ordering of flow between the 3 lanes change with throughput. In heavy traffic, the left lane has the greatest flow and the right lane is lowest, but immediately after a peak in flow, the left lane's flow dips sooner and faster to levels below the middle lane. Again, we can connect our observations to the location of the modes of the distributions of flow for the 3 lanes.

In order to see how throughput depends on congestion, we can plot one against the other for each of the 3 lanes. We do this with

```
ggplot(data = trafficLong) +
  geom_point(mapping = aes(x = occ, y = flow), alpha = 0.2) +
  facet_grid(lane ~ .) +
  labs(x = "Occupancy", y = "Flow")
```

Here we juxtapose the scatter plots to avoid over plotting and to make it easier for us to compare the relationship between flow and occupancy. We see the same pattern of points in each plot, but there are also important differences. The pattern shows that when there are few cars on the road, flow is small and so is congestion. The linear collection of points indicates that adding more cars increases congestion and flow also increases. However, at some point this linear relationship breaks down and as congestion increases traffic slows and flow decreases (i.e., we have a traffic jam). Two key differences between these 3 scatter plots are the slope of the line and the place where the breakdown occurs. The relationship between flow and occupancy breaks down soonest in the right lane (in terms of level of

congestion) then in the middle and last in the left lane. The pre-breakdown slope is smaller in the right than the middle and left lanes, which appear to have about the same slope. If we work with the simple assumption that all vehicles have the same length, then this slope is proportional to velocity.

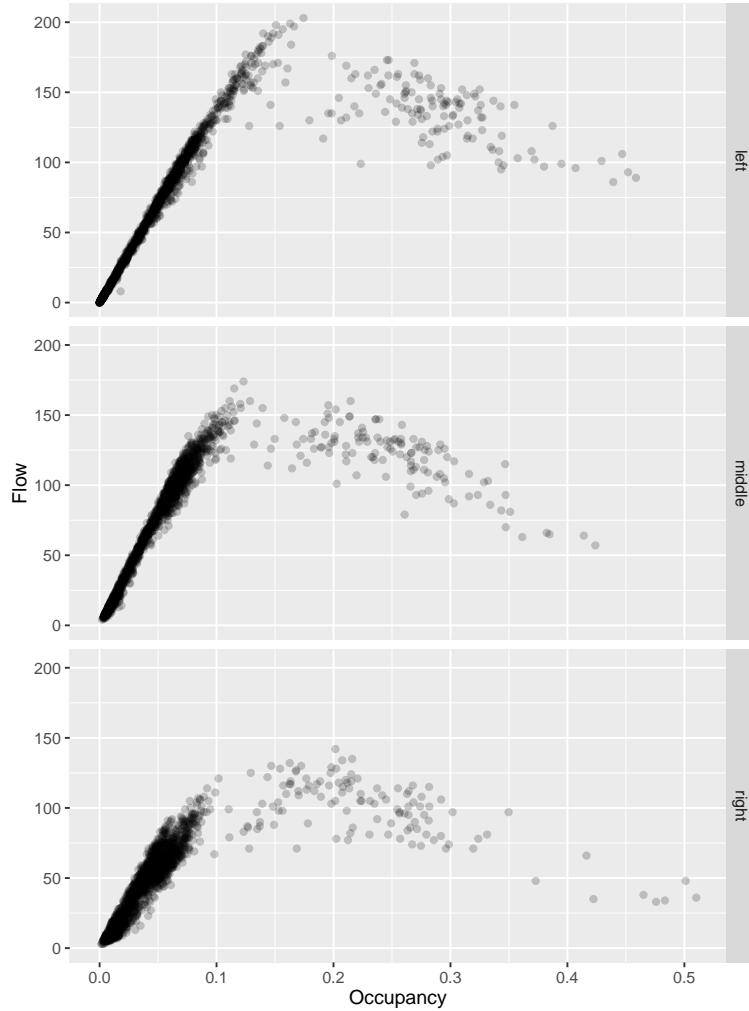


Figure 2.19: Flow vs. Occupancy for 3 Lanes of Traffic. *These 3 juxtaposed scatter plots display flow and occupancy measurements in 5-minute intervals for 3 lanes of traffic. The linear relationship between flow and occupancy breaks down at different occupancies for the 3 lanes.*

These exploratory plots have revealed many insights into the data. For example we have seen that the lanes appear to behave similarly in general but somewhat differently in specifics, and that the linear relationship between flow and occupancy breaks down. These insights help us in our more formal analysis. For example, we probably want to model the relationship between flow and occupancy with a smooth curve or a piecewise linear curve, and we want to take the lane into account in estimates for the breakdown point.

Finally, with this traffic example, we have collected all measurements at one location in

a particular week. If we want to generalize the relationships we have discovered between flow and occupancy or flow and day of the week, then we need to consider how representative is this location of the California freeway system and how similar are the other weeks of the year to this particular week in March.

2.7.2 Exploring Country Statistics

The World Bank data provide summary statistics for nearly all countries in the world. With these data we can examine relationships between these various statistics. As an example, we explore the relationship between health care expenditures per capita and life expectancy at birth. Both variables are quantitative. We earlier examined the distribution of life expectancy (see Figure 2.13), where we found a skewed left distribution. That is, most countries have an average life expectancy of 70 to 80 years, but a substantial number of countries have low life expectancies that range from about 40 to 65 years.

A histogram of health care expenditures per capita for these 200+ countries appears in Figure 2.20. The dark red line segments along the bottom of the plot show the individual values for the countries. The color is somewhat transparent so that when segments overlap they show a darker color and denser regions display thicker clumps. These red ‘threads’ create a rug plot. Again, we have a skewed distribution, but this time it’s right skewed. The vast majority of countries spend less than \$500 per person. This distribution has some unusually large values; it appears that one country spends about \$5000, another about \$4000, and a handful of countries spend between \$2000 and \$3000 per capita. We make this histogram with

```
ggplot(data = wbSub, aes(x = healthPC)) +
  geom_histogram(aes(y = ..density..), alpha = 0.4) +
  geom_rug(col="darkred", alpha = .4) +
  labs(x = "Health Expenditure per Capita")
```

We are interested in relationships between variables and these two histograms do not reveal this connection. We imagine that health care expenditure (per capita) is positively correlated with a longer life expectancy, but we can’t glean this information from the two univariate distributions. We make a scatter plot where each country corresponds to one point so we can see how life expectancy and health care expenditures vary together. We create the scatter plot in Figure 2.21 with

```
ggplot(data = wbSub) +
  geom_point(aes(x = healthPC, y = lifeExp)) +
  labs(y = "Life Expectancy",
       x = "Health Care Expenditure per Capita")
```

In this figure, we see a highly curved relationship with the countries that spend less than \$500 per capita having a great range in life expectancy. Also, the curve flattens out over a wide range of high-expenditure countries.

If we can transform the variables and straighten the curvature, then this helps us better ascertain the relationship between the 2 variables. In this case, we take the logarithm of health care spending to bring in the large values and make the distribution more symmetric.

```
ggplot(data = wbSub) +
  geom_point(aes(x = healthPC, y = lifeExp)) +
  scale_x_log10() +
  labs(y = "Life Expectancy",
       x = "Health Care Expenditure per Capita")
```

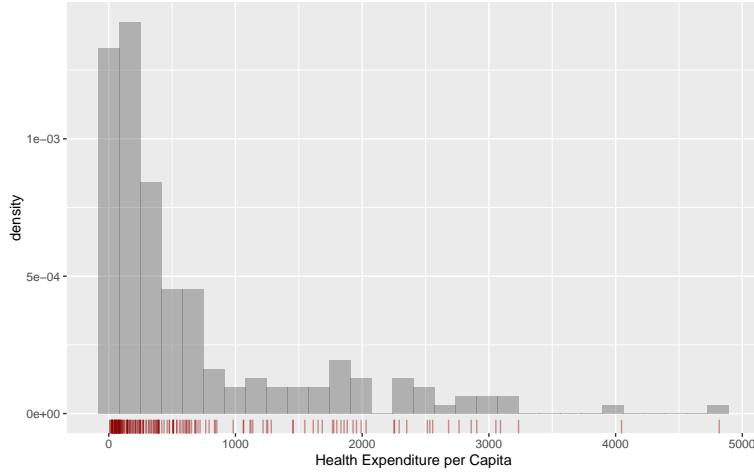


Figure 2.20: Histogram of Health Expenditure per Capita. *This histogram and accompanying rug plot show the distribution of average health expenditure per capita in 2000 for countries, as reported by the World Bank. The distribution is highly skewed to the right with a few countries spending over \$2000 per capita, and there appears to be two outliers at \$4000 and \$5000.*

Now we see in Figure 2.22 a linear association between life expectancy and the log of health care expenditures. The correlation coefficient is a measure of linear association and we find a strong linear association of about 0.78 with

```
with(wbSub, cor(log(healthPC), lifeExp, use = "complete.obs"))
[1] 0.7836
```

Although this relationship is roughly linear, the variability in life expectancy for low-expenditure countries is greater than for high-expenditure countries. Note, we can compute the correlation even when the variables do not display a linear association. For example, the correlation between the life expectancy and the untransformed health expenditure is 0.62, despite our original scatter plot showing a clear lack of a linear association. It's a good idea to not rely on correlation alone to assess linear association.

The scatter plots we have examined show all countries, which have a tremendous range in prosperity. When we zoom in to examine a more homogeneous subset of countries, we can get a closer (and possibly better) view of the relationship between life expectancy and health care expenditure. For example, let's examine the countries with the highest gross domestic product (GDP). In this case, there are few enough points that we can label them with country codes (see Figure 2.23). We create this scatter plot with

```
topGDP = order(wbSub$gdp, decreasing = TRUE)[1:50]

ggplot(data = wbSub[topGDP, ]) +
  geom_point(aes(x = healthPC, y = lifeExp)) +
  geom_text(aes(x = healthPC, y = lifeExp, label = ctry),
            size = 3) + scale_x_log10() +
  labs(y = "Life Expectancy",
       x = "Health Care Expenditure per Capita")
```

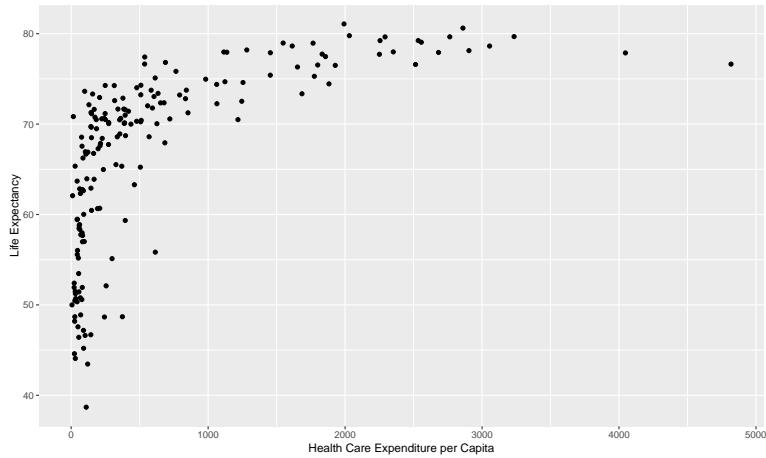


Figure 2.21: Life Expectancy vs. Health Care Expenditures per Capita. *This scatter plot displays the average life expectancy and health care expenditure per capita in 2000 for countries reported by the World Bank. The pattern of points is highly curved and expenditures range over several magnitudes.*

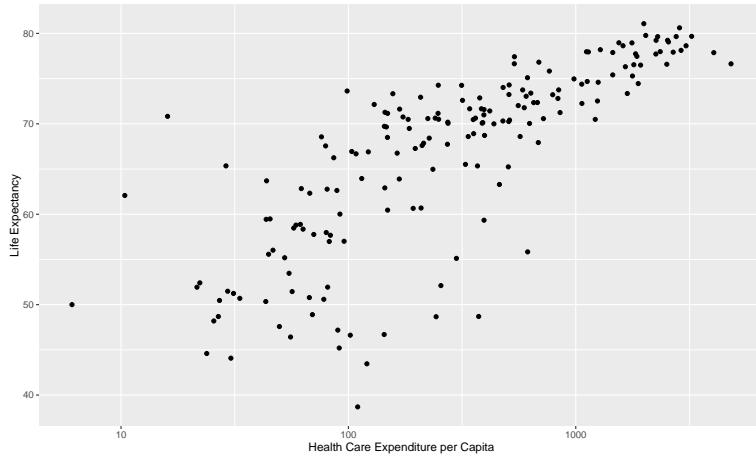


Figure 2.22: Life Expectancy vs. Health Care Expenditures per Capita (Log Scale). *In contrast to Figure 2.21, this scatter plot shows health care expenditure per capita on a log scale. The relationship appears roughly linear with a large variability in life expectancy for countries spending under \$1000.*

The correlation between life expectancy and health expenditures for this subset of data remains high,

```
with(wbSub[topGDP, ], cor(log(healthPC), lifeExp,
  use = "complete.obs"))
```

```
[1] 0.8029
```

In some situations, the correlation drops because the subset has a smaller range of values;

this is not the case here because the strong curvature and high variability are gone when we exclude the low-GDP countries.

This scatter plot has several interesting features. South Africa (ZAF) has a much lower life expectancy compared to all other top countries, especially those with similar expenditures per capita. The US spends 10 times more than any other country on health care but is outstripped in terms of life expectancy by 20 countries or more.

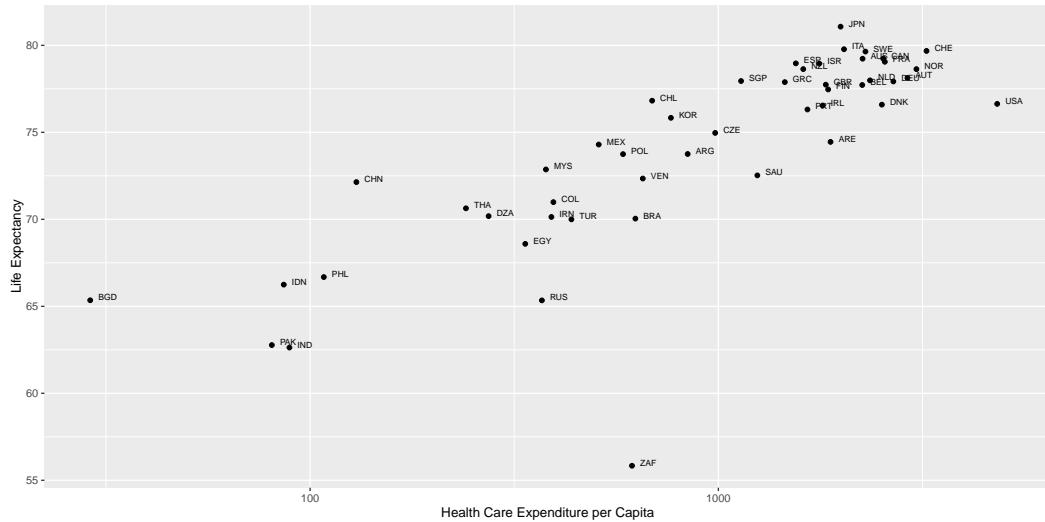


Figure 2.23: Life Expectancy vs. Health Care Expenditures per Person (Top Countries). This scatter plot zooms in on those countries with GDP in the top 50 of all countries in the World Bank (see Figure 2.22 for plot with all 214 countries). Points are identified with their country ID. South Africa (ZAF) has unusually low life expectancy given its expenditures.

Despite these interesting findings, we must be careful when interpreting them for reasons that are related to the kind of data. The relationships observed are based on rates and averages, e.g., the infant mortality rate in the country and the average years of education. Measurements such as these tend to overstate the strength of an association. For example, if we have data on individual's income and education, the variability between these tends to increase and as a result the correlation tends to be lower than for the group averages. Also these data are observational, not experimental so although we expect that spending more on health care leads to an increase in life expectancy, we can imagine situations where this wouldn't occur. For example, in the US many people go without adequate health care so increasing expenditures may not lead to increased life expectancy if funding for the underserved does not change. Lastly, if we model this and other relationships, we must bear in mind that the variability in average life expectancy is not constant across expenditures.

2.7.3 Exploring Emergency Room Visits due to Drug Abuse

The DAWN survey data (Q.2-2 (page 52)) studies carefully selected emergency room visits. A probability scheme was used to choose the visits for inclusion in the survey. In particular, all hospitals with 24-hour emergency departments in the US were divided into regional groups and within each group a sample of hospitals was chosen at random. The chance a hospital was selected to participate depended on the size of the hospital and the region. Then, for each hospital selected, a random sample of emergency department medical records

was taken from those with a drug-related problem. More information may be found at www.samhsa.gov/data/. Although a probability sample, every possible subset of ER visits is not equally likely because of the complex sampling scheme. This means that we need to incorporate each record's probability of selection into the analysis. As an example, if the proportion of ER visits in Los Angeles were sampled at twice the rate of those in other regions, then LA is over-represented in the sample and we need to correspondingly down weight their responses. Before setting up the survey design for our analysis, we show a simple example of why it's important to consider how the subjects are selected for the survey.

As mentioned in Section 2.3.2, most, if not all, of the variables in this survey are categorical, some of which are ordered. This means that we summarize and plot the data differently from the previous examples. Let's begin by comparing the reason for the ER visit for men and women. The variable `allabuse` is binary (dichotomous) and simply records whether or not the purpose of the visit was entirely drug-abuse related. Since this information is qualitative, we convert `allabuse` into a factor with

```
dawn$allabuse = factor(dawn$allabuse, labels = c("no", "yes"))
```

We make a 2-way table of `allabuse` and `sex` with

```
totRow = table(dawn$sex)
100 * table(dawn$sex, dawn$allabuse) / as.numeric(totRow)
```

	no	yes
male	31.98	68.02
female	53.99	46.01

Note that we normalized the proportions in the 2-way table by row (which we calculated and saved in `totRow`) so the row proportions add to 1. This way, we control for an unequal number of males and females in the sample. These data show that roughly 68% of the male and 46% of the female visits to the ER for drug-abuse related reasons were solely for drug abuse. However, when we take the probability of selection into account, these numbers change to 57% and 40%, respectively. That is, both percentages decrease and they are closer together.

The DAWN data includes information about the survey design and weights to adjust for the unequal representation of ER visits. We do not go into the details of the sampling scheme, but note that we can use the `survey` package to describe the design and incorporate the design into tallies and plots. We set up the design with

```
library(survey)
dawndsgn = svydesign(id = ~ strata + psu, weights = ~ wt,
                      data = dawn)
```

Then, we can compute the tables with `svytable()`, which uses the design to adjust the tallies, e.g.,

```
totSex = svytable(~ sex, dawndsgn, Ntotal = 100)
totSexAbuse = svytable(~ sex + allabuse, dawndsgn, Ntotal = 100)
percentWinSex= 100 * totSexAbuse / as.numeric(totSex)
percentWinSex
```

```
allabuse
sex      no    yes
male    42.40 57.60
female  59.58 40.42
```

A statistical graph of a table can be much easier for us to make comparisons. We don't get bogged down in the particular digits and instead get a general understanding of the differences between groups. For example, in the bar plot of Figure 2.24, the heights of the bars match the percentages in the table above. These are color-coded to make it easier to distinguish between males and females, and the placement of the two bars for 'not all abuse' are close to each other to further facilitate this comparison. We make this plot with

```
barplot(percentWinSex, beside = TRUE,
        col = c("lightblue", "mistyrose"),
        ylab = "Percentage",
        xlab = "ER Visit Solely for Substance Abuse")
legend("topleft", fill = c("lightblue", "mistyrose"),
       legend = rownames(percentWinSex), bty = "n")
```

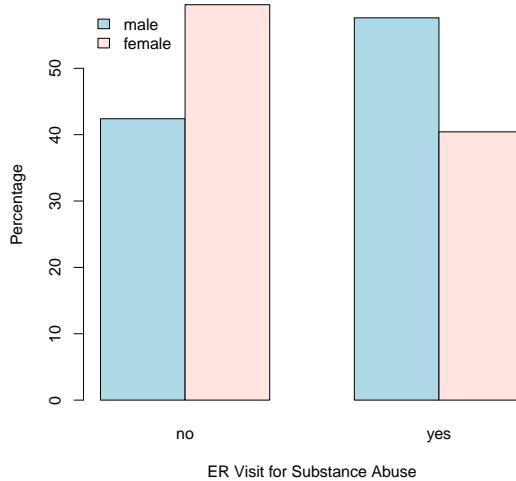


Figure 2.24: Comparison of Reason for ER Visit. *This bar plot compares males and females according to whether the ER visit was entirely abuse related (yes) or due to a combination of reasons including drug abuse (no).*

The mosaic plot offers another kind of visual comparison. We create the mosaic plot on the left in Figure 2.25 with

```
plot(svytable(~ sex + allabuse, dawndsgn), main = "")
```

This plot makes clear the comparison between males and females. To create it, a rectangular region is divided vertically according to the proportion of males and females in the sample. Then, each new rectangle is divided horizontally according to the proportion of yeses and noes within that subgroup. As a comparison, the mosaic plot on the right in Figure 2.25 ignores the survey design. We make this with

```
with(dawn, mosaicplot(table(sex, allabuse), main = "", ylab = ""))
```

These plots make it apparent that we need to include the design of the survey in our analysis or we risk creating an incorrect picture of the results.

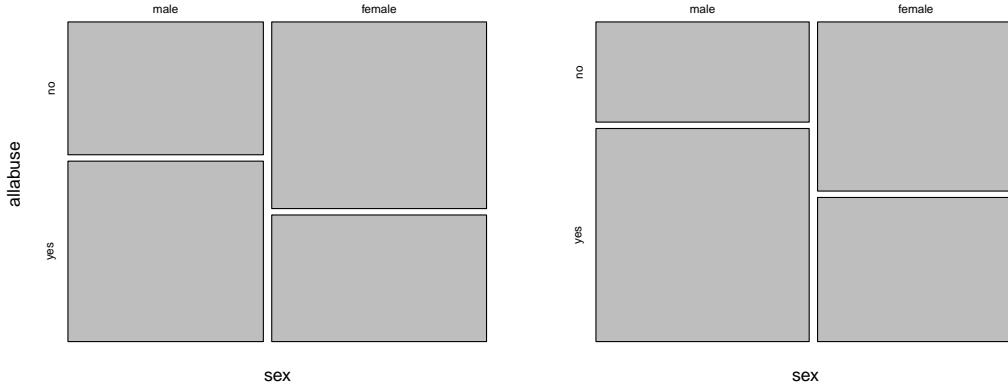


Figure 2.25: ER Visit Due Solely to Substance Abuse by Sex. *The mosaic plots on the left compares males and females according to whether the ER visit was entirely abuse related (yes) or due to a combination of reasons including drug abuse (no). The plot on the right was made without taking into account the sampling design. The unweighted mosaic plot overstates the yes-category for both sexes.*

Another variable in the data provides the reason for the ER visit. This can be one of the following: suicide attempt, seeking detox, alcohol only (for those under 21), adverse reaction, overmedication, malicious poisoning, accidental ingestion, and other. Again, to examine the relationship between sex and the reason for the visit, we can make a bar plot or mosaic plot. Instead we use a dot chart, which is a simplification of the bar plot. Since there is no meaning attributed to the width of the bars in a bar plot, we can shrink the bar and simply mark its length with a dot along a line (see Figure 2.26). We organize this dot chart to make the comparisons between males and females for each category of reason by computing the percentage for a reason within males (and similarly for females) and plotting the male and female figures side by side for each reason. We do this with

```
totSexType = svytable(~ sex + type, dawndsgn, Ntotal = 100)
percentTypeWinSex= 100 * totSexType / as.numeric(totSex)

dotchart(percentTypeWinSex,
         xlab = "Percent of ER Visit Type within Sex")
```

We see in Figure 2.26 that the distribution of reasons for males and females are very similar with 2 striking exceptions – adverse reactions and the ‘other’ category. A greater proportion of females visited the ER due to adverse reactions, and a greater proportion of the males visited the ER for a reason not captured by the 7 possibilities given. This large fraction of males in the other category (40%) indicates a problem with the coding of this variable that the survey researchers may want to further investigate.

We can also create line plots for these data (see Figure 2.27). Like the dot chart, the line plot represents a proportion with a point. However, these points are plotted one above or below the other for males and females for each reason, and the points for each sex are connected by line segments. We make this line chart with

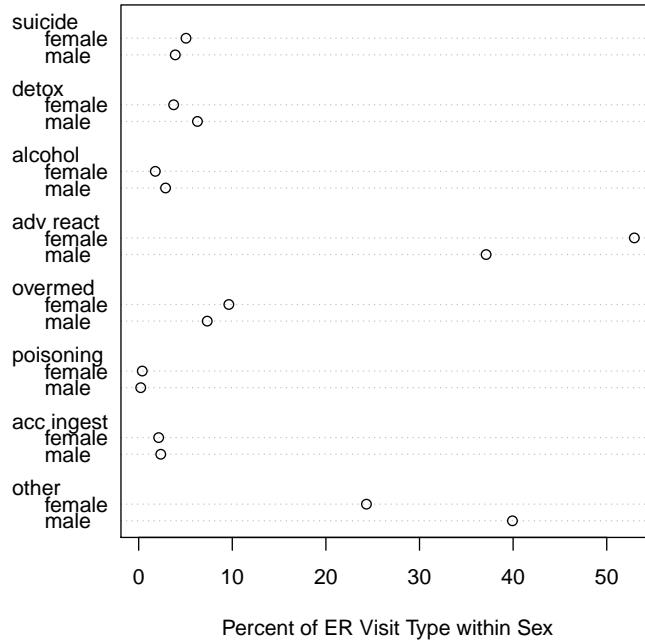


Figure 2.26: Reason for ER Visit by Sex. *This dot chart compares males and females according to the reason for visiting the ER (all visits are drug-abuse related). The percentages for males and females are quite close for all reasons except adverse reaction, where a greater proportion of females visit the ER for this reason, and the ‘other’ category where a greater proportion of males visit for this reason.*

```

plot(percentTypeWinSex["male", ] ~ I(1:8), type = "l",
      ylim = c(0, 55),
      ylab = "Percentage of ER Visit Type w/in Sex",
      xlab = "Type of Abuse", axes = FALSE)
points(I(1:8), percentTypeWinSex["female", ], type = "l", lty = 2)
box()
axis(side = 2)
axis(side = 1, at = x, labels = colnames(cage))
legend("topleft", legend = rownames(cage),
       border = NULL, bty = "n", lty = c(1, 2))

```

This layout helps us see the male values together and in comparison to the female values. The differences between males and females for these two categories is quite striking, especially because they make up a large fraction of the responses.

We are also interested in exploring the 3-way relationship between sex, age, and the reason for the ER visit. Particularly, we might want to know whether the reason for the ER visit changes with age and whether we observe a difference between males and females for these age groups. To do this, we can create a mosaic plot with

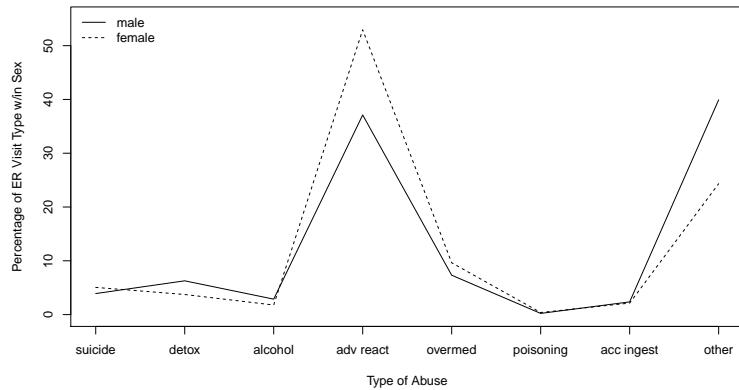


Figure 2.27: Interaction Plot for the Reason for ER Visit by Sex. *This interaction plot displays the same information that is in the dot chart in Figure 2.26. Here, the reasons are arranged along the x-axis rather than the y-axis and the values for each sex are connected by line segments.*

```
plot(svytable(~ sex + age + type, dawndsgn), main = "",  
      ylab = "", xlab = "", las = 2)
```

In this case, the rectangular area is first divided vertically according to sex, then horizontally according to age within sex, and lastly, vertically within each sex-age region according to the type of visit. We see in Figure 2.28 how the proportion of ER visits due to an adverse reaction changes by age with it being the predominant reason for the young and old. Also striking is the large proportion of suicide attempts for teenage girls. There are many other interesting observations we can make from this mosaic plot. We also note that if we change the order of the variables in the formula `~ sex + age + type`, then the order of division of the regions changes and a very different looking plot appears. The order determines the sequence to use in slicing up the rectangles. Each rectangle is subdivided based on the proportion within the corresponding group. This order can be important for facilitating particular comparisons.

2.7.4 EDA Summary

In this section, we summarize the basic concepts of EDA and considerations in creating visualizations that we demonstrated in the previous examples. More details on how to construct informative statistical graphs appears in Chapter 3.

How were the data collected?

It's important to keep in mind where the data come from because this impacts whether and how we can generalize the findings of our exploration and analysis. One question to consider is: do the subjects under study form a census or a sample? And, if the data are a sample, how were the subjects selected for the sample? We also consider whether these data represent a snapshot in time, and if repeated measurements were made over time. We saw with the traffic data that we have only measurements at one location for 6 days so if we want to generalize from these data to traffic on California freeways, we must consider how representative is this location and time period. The World Bank data consists of averages and rates for all

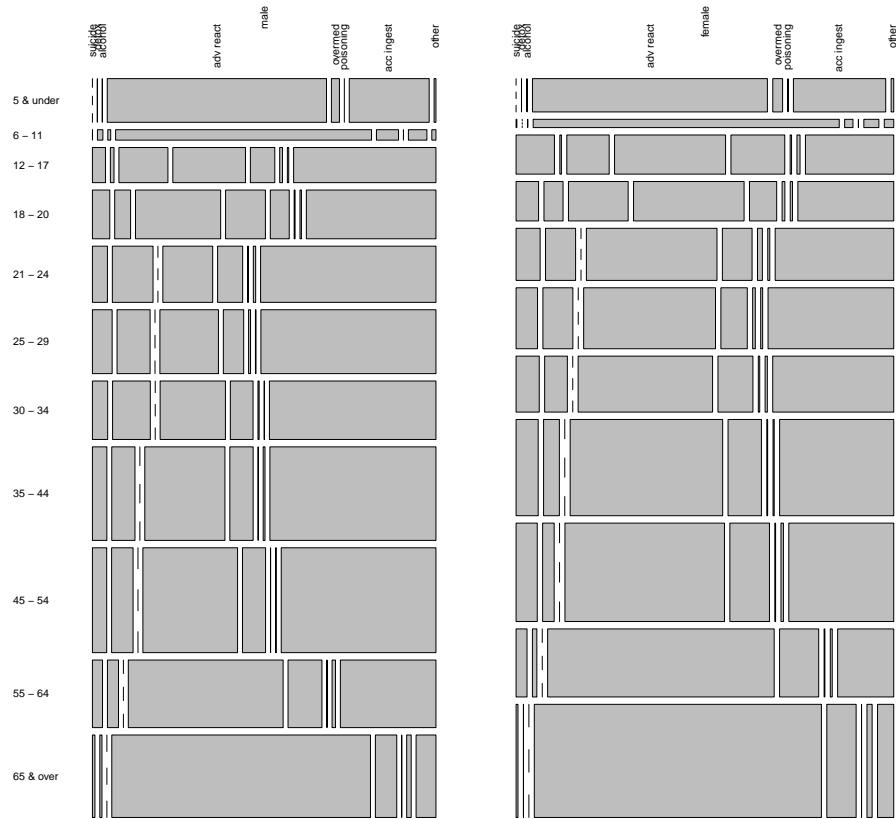


Figure 2.28: Type of Substance Abuse by Sex and Age. *This mosaic plot compares age and sex for different types of substance abuse. Males are on the left and females on the right. We see the type of abuse changes for different age groups.*

countries in a specific time period. Here, we must be careful not to overstate the strength of the relationships found when considering implications for individuals, assumptions of causality, and heterogeneity in variances. For the DAWN survey, a probability method was used to select the subjects (ER visits) for the survey and we must take care to include the design in our analysis so that relationships discovered are representative of the population.

What type of data are we exploring?

An important first step in making a statistical graph is to recognize the data type of the variable(s). We need to ascertain whether a measurement is quantitative or categorical because the approaches to viewing and summarizing quantitative information differs from qualitative. In the DAWN survey, we worked solely with qualitative variables, such as the sex of the patient and the reason for the ER visit. Additionally, age is qualitative in this study because it is reported in categories from '5 and under' to '65 and over'. These categories are naturally ordered, unlike with the categories for sex and reason for the ER visit so it may be tempting to treat this categorized age variable as quantitative. However, this can be problematic because the distance from one category to the next is not the same number of years and a person in, say, category 6 is not twice as old as someone in category 3. Alternatively, the sex variable in the World Bank survey is quantitative because it is the

proportion of males in the country, which ranges between 0 and 1. The measurements in the World Bank are generally quantitative as they are averages, e.g., health care expenditures per person, and rates, e.g., infant mortality per 1000 live births. Finally, the measurements of flow and occupancy over the loop detectors in the traffic data are quantitative measures, as is the time when a measurement was taken. (Here time is an ‘interval’ and not a ‘ratio’ variable because it makes sense to look at differences in time but not multiples.) Lastly, the lane in which the measurements were taken is qualitative.

What do we look for in a distribution?

To examine the distribution of values for a quantitative variable, we create a histogram or density curve. In histograms, the area of the bar represents the proportion of values that fall in that range. For this reason, the bins of a histogram need not be all the same width. (For density curves it’s the area under the curve within the range of x values that represents the proportion). With a histogram or density curve, we get a sense of the following characteristics of the distribution: symmetry and skewness, number and location of modes (high frequency regions), length of tails, gaps where there are no observations, and unusually large or anomalous values. For qualitative data, the bar plot serves a similar role to the histogram in that we can visualize the ‘popularity’ or frequency of different categories. In a bar plot, the frequency of a category is represented by the height of the bar and the width carries no information about the distribution. For this reason, a dot chart is an alternative graphical display for qualitative data.

Summary statistics such as the mean and standard deviation (SD) or the median and interquartile range (IQR) give us an idea of the center of a quantitative variable’s distribution and the spread of typical values. However, the density curve and histogram give us a more complete sense of the data distribution. The mean and SD are often useful summary statistics for symmetric distributions and the median and IQR are typically used with skewed distributions. Box and whisker plots provide a visual display of these quantile-based summaries. Box and whisker plots typically reveal more information about a distribution than the quartiles, but they are not as informative as a density curve or histogram because they cannot, for example, show modes and gaps.

Summary statistics for qualitative data amount to tables of categories with their frequencies or proportions, i.e., the same information conveyed in a bar plot and dot chart. The advantage of the plots is in the ease of making relative comparisons. For this reason, e.g., bars in a bar plot are often ordered according to their height, unless the ordering of the categories has some meaning, e.g., with education level.

Method of Comparison

We often want to make comparisons with our data. We may want to compare the distribution of a variable to an historical value or a benchmark of some kind; we can do this by adding a reference point or line to a plot (e.g., Figure 2.13). Alternatively, we may want to compare the relationship between two quantitative variables to a line with, say, slope 1, which we can do by simply adding the reference line to a scatter plot (e.g., Figure 2.8). Or, we may want to split our data into subgroups, according to the value of another variable and compare the resulting distributions through side-by-side box plots, violin plots (e.g., Figure 2.18), or super-posed density curves (e.g., Figure 2.17), side-by-side bar plots (e.g., Figure 2.24), or mosaic plots (e.g., Figure 2.25). The method of comparison can help provide context to a problem or insight into a distribution or relationship.

Looking for Relationships between Variables

Similar to the decision to make a histogram or a bar plot, we must consider data types when choosing a plot to visually explore the relationship between two variables. When both variables are quantitative, we can examine their relationship through a scatter plot. We

may want to consider transformations of the variables to straighten the relationship, or we may want to add a locally smoothed curve to the plot that shows the average value of the y variable for neighboring x values. Also, if one of the variables is time, then we typically place this variable on the x-axis and make a line plot that connects the y-values. With two qualitative variables, we make grouped bar plots, dot charts, and line plots, as well as mosaic plots. And in the case where we have one quantitative and one qualitative variable, we make side-by-side box plots and overlaid density curves.

Of course, these are general guidelines for the selection of a plot type, not strict rules. There can be situations where a particular graph is appropriate even though it doesn't fit into these guidelines. For example, suppose that we want to look at the age distribution in the DAWN survey. Although age is a categorical variable in this survey, a histogram is a natural way to summarize an age distribution. In this case, if we align the bins of the histogram with the categories used for age, then we can make a histogram from these categorical data. As another example, flow and time are numeric quantities, but it is reasonable to compare the distribution of flow across days of the week. To do this, we can make box plots of flow for each day of the week, essentially converting time into a categorical variable.

2.8 Summary

Tables with rows that correspond to observations and columns to various measurements on each observation are a convenient and common representation for data. Moreover, tabular data are often stored in plain text files where each observation appears on its own line in the file and a delimiter separates the values for the observations or the values appear in the same fixed location for all lines in the file.

The data frame is a natural structure for tabular data as the vectors in the data frame correspond to variables and these can be of different data types.

It is a simple process to read tables into a data frame. However, things can go wrong and data validation and cleaning are important steps when reading data into R. Simple data summaries and plots are often helpful in this validation step and can serve as the beginning of exploratory data analysis. EDA is a key part of the data analysis process. In addition to validating data, it serves to examine the properties of the variables and relationship between variables.

Facility with computations help us in this process. For example understanding data types and impact of EDA lead to data type conversions and decisions as to the best data structure to use, e.g., data frame, matrix, array, etc. Understanding the concept of a table helps us in determining the best shape for our data table and gives us skills for reshaping tables. Understanding formats for plain text files helps us choose the appropriate function for reading the data and providing the appropriate arguments so the data can be discerned.

We have not provided examples of the many versions of functions that can be used to read data into R. Instead, our focus is on understanding the computational and statistical aspects in determining how to read, clean, and organize the data. For more details on reading data see

EDA is like detective work where we actively seek clues about our data that can help direct us in later stages of analysis. It is important to not lose sight of where the data come from in our EDA and more generally. EDA is an important aspect of data analysis that tends to have limited development in statistics textbooks. Moreover, we learn the philosophy and goals by example. For more information on EDA see Tukey's original introduction to the subject [10]. See also [11] and [7] for more examples of EDA.

2.9 Functions for Reading and Exploring Data

This chapter introduced many of the functions available in *R* for working with data that are arranged in table-like formats. These are summarized below. A summary of the plotting functions appear at the end of Chapter 3.

`read_delim()` (in `readr`) Read a text file with values in a tabular arrangement and return a data frame with one row for each row in the table. The delimiter is specified with `delim`; common delimiters are white space (specified by `" "`), the tab (`"\t"`), and the comma (`", "`). To argument `col_names` is a logical to specify that the first line of the file contains column names (TRUE is the default); to skip the first `n` rows of the file (not include them in the data frame), set `skip`; specify the class of each vector in the data frame with `col_types` (`factor` is not accepted as a data type); and provide the values to be converted into NAs in a character vector to `na`.

`read.table()` Similar to `read_delim()`. The delimiter is specified with `sep` (default: `" "`). To specify that the first line of the file contains column names, set `header` to TRUE; to skip the first `n` rows of the file (not include them in the data frame), set `skip`; specify the class of each vector in the data frame with `colClasses` or avoid the character vectors from automatic conversion into factors by setting `stringsAsFactors` to FALSE; and provide the values to be converted into NAs in `na.strings`.

`readBin()` Read a binary file from the connection specified in `con` and return a vector of type specified by `what`. Specify the maximum number of records to be read in with `n` and the the number of bytes per element with `size`. To specify that the data being read in is an unsigned integer, set `signed` to FALSE.

`read_fwf()` (in `readr`) Read in fixed width formatted data and return a data frame with one row for each row in the table. The columns positions are specified with `col_positions`, which takes in output from the functions `fwf_empty()`, `fwf_widths()`, or `fwf_positions()`. For `fwf_positions()`, provide the starting position, ending position, and optional column names with `start`, `end` and `col_names`, respectively. Other arguments match those of `read_delim()`

`read.fwf()` Similar to `read_fwf()`. The `widths` of the columns are specified as an integer vector. To specify that the first line of the file contains column names, set `header` to TRUE. Columns can be skipped by supplying a negative integer for a column width. Other arguments are similar to those of `read.table()`.

`read.dcf()` Read in a Debian Control File (DCF) format and return a character matrix with one row for each record and one column for each field. One of the defining features of a DCF format is that each entry in the dataset is a key: value pair. The `fields` argument is used to specify fields to be read in. If there are multiple occurrences of a field, the default behavior is to read its last occurrence; to gather all occurrences, specify `all` to be TRUE.

`with()` Evaluate *R* expression on the supplied data object, usually a list or data frame. This is a convenience function to refer to elements in the data object without needing to refer to the data object itself in the expression, e.g., `with(dataframe, cor(x, y))` is equivalent to `cor(dataframe$x, dataframe$y)`.

2.10 Guided Practice

Saratoga

The file `Saratoga.txt` contains data that are a random sample of 1,728 homes taken from public records from the Saratoga County in New York. The data was collected by Candice Corvetti (Williams '07) and made available at <http://community.amstat.org/stats101/home>.

1. Examine the file `Saratoga.txt` to determine its format. Read the file into a data frame in *R* and name it `saratoga`. Examine the resulting data frame to determine its structure and whether the data have been read in properly.
2. Create a new factor variable called `fp` that indicates whether or not a house has a fireplace.

Weather

The file `weather.txt` contains daily temperature recordings for San Francisco in 2015. In the following exercises, we read the dataset into *R* and check that the data are properly read. This includes some data clean-up and some sanity checks.

1. Examine the file to determine the format and then read it into a data frame.
2. What are the names of the variables? Rename them to short informative names. (We suggest `month`, `day`, `low`, `high`, `nlow`, `nhigh`, `rlow`, `rhigh`, `year.rl`, `year.rh`, `precip`, `rprecip`, and `year.rp`.)
3. Explore the variables. What is the class of each variable? Do the values of each variable seem reasonable?
4. Why is `precip` not `numeric`? Make a table of the values in precipitation. The `T` stands for 'Trace'. Set it to 0 and convert to the `precip` variable to `numeric`.
5. Convert the month and day variables into a `POSIXct` formatted variable. Include the year in the format. Begin by creating a string of the year, month, and day, where each element is separated by a dash. Then use this string to create a `POSIXct` formatted date variable. The help file of `strptime()` is useful here.
6. On the same plot, make line plots of daily high and low temperatures and record high and low temperatures. Make sure to appropriately set the limits of the y-axis!

Olympics 2012

The file `olympics2012.txt` contains the country medal tallies and number of men and women competing for countries that participated in the Summer 2012 Olympics.

1. Examine the file to determine its format. Read the file into *R* as a character matrix. Make sure the data are read in properly.
2. Transform the matrix to a data frame. Convert the variables to `numeric` that should be.
3. Examine the relationship between the number of athletes competing and the number of medals. To do this, create a new variable which is the sum of the male and female athletes. Make a scatter plot of the number of medals won against the number of athletes. Describe the relationship.

4. Consider transformations to straighten the relationship between the number of athletes competing and the number of medals. Two common transformations for count data are the log and square root transformations. The variation in the counts tends to increase with the square root of the count. Compare the log and square root transformation of medal counts. Which does a better job of straightening the relationship?
5. We want to include data about the size and wealth of the countries in our analysis. Use the World Bank data and create a data frame with 2010 data on the countries. Follow the code from the chapter to read and reshape the World Bank data. Feel free to keep additional variables.
6. Merge these two data frames. Since we are only interested in countries that competed in Summer 2012 Olympics, we do not want countries in the World Bank that did not compete.
7. Examine the relationship between two variables in the World Bank data. Color the countries, i.e., the points in the scatter plot, by the number of medals won by the country, e.g., no medals, a few, etc. To use these colors, create a factor variable from the total number of medals using the `cut()` function. Consider transformations and normalizations, e.g., does it make sense to plot GDP or GDP per person?

Binary data

The file `data3` in the `digitsBinary` directory contains 1,000 binary images of handwritten digits for the number three.

1. Using the code in Section 2.4 as a guide, read `data3` into R. Reshape the data into an array of 1,000 28×28 matrices.
2. Explore the data by making an image plot like the one shown in Figure 2.7. This time try different values for the quantile. How does the plot change?

2.11 Exercises

Bibliography

- [1] BioFabric. <http://www.biofabric.org/>, 2014.
- [2] Henrik Bengtsson, Andy Jacobson, and Jason Reidy. `R.matlab`: Read and Write MAT Files and Call MATLAB from Within R. <http://cran.r-project.org/web/packages/R.matlab>, 2016. R package version 3.6.0.
- [3] Caltrans. Caltrans Performance Measurement System. <http://pems.dot.ca.gov/>, 2014.
- [4] Ben Hammer. Hillary Clinton's Emails. <https://www.kaggle.com/kaggle/hillary-clinton-emails>, 2016.
- [5] Tim Hanrahan, Martin Burch, and Katie Marriner. *Search Hillary Clinton's Emails*. The Wall Street Journal, Mar 1, 2016.

- [6] William Longabaugh. RBioFabric: BioFabric network visualization tool. <https://github.com/wjrl/RBioFabric>, 2013. R package version 0.3.
- [7] Wendy L. Martinez, Angel R. Martinez, and Jeff Solka. *Exploratory Data Analysis with MATLAB, second edition*. Chapman and Hall/CRC, London, 2010.
- [8] SAMHSA: Substance Abuse and Mental Health Services Administration. Drug Abuse Warning Network. <http://www.samhsa.gov/data/emergency-department-data-dawn>, 2010.
- [9] The World Bank. World Bank Open Data. data.worldbank.org/, 2016.
- [10] John W. Tukey. *Exploratory Data Analysis*. Addison-Wesley, Boston, 1977.
- [11] Paul F. Velleman and David C. Hoaglin. *Applications, Basics and Computing of Exploratory Data Analysis*. Duxbury, Pacific Grove, CA, 1981.
- [12] Hadley Wickham. tidyverse: Easily Tidy Data with 'spread()' and 'gather()' Functions. <http://cran.r-project.org/web/packages/tidyr>, 2016. R package version 0.6.0.
- [13] Hadley Wickham, Jim Hester, Romain Francois, Jukka Jylinkki, and Mikkel Jorgensen. readr: Read Tabular Data. <http://cran.r-project.org/web/packages/readr>, 2016. R package version 1.0.0.
- [14] Hadley Wickham and Evan Miller. haven: Import 'SPSS', 'Stata' and 'SAS' Files. <http://cran.r-project.org/web/packages/haven>, 2016. R package version 0.2.1.

3

Visualization

CONTENTS

3.1	Introduction	107
3.1.1	Composing a Graph of Voter Registration Trends	108
3.2	Data Types and Plot Choice	110
3.2.1	Terminology	111
3.2.2	The Kaiser Family Data	111
3.2.3	Univariate Plots	113
3.2.4	Bivariate Plots	117
3.2.5	Conveying Relationships between 3 or More Variables ..	122
3.2.6	Scalability – Real Estate Data with 500,000 records	125
3.2.7	Measurements in Time	126
3.2.8	Geographic Data	128
3.3	Guidelines	129
3.3.1	Scale	135
3.3.2	Position	135
3.3.3	Shape	136
3.3.4	Aggregates	136
3.3.5	Color	136
3.3.6	Context	137
3.3.7	Over Arching Considerations	137
3.4	Iterative process	138
3.5	Rs Graphics Models	138
3.5.1	Painter’s Model in Base R	139
3.5.2	Grammar of Graphics Model in <code>ggplot2</code>	142
3.6	Creating Unique Plots	145
3.7	Summary	146
3.8	Exercises	146
	Bibliography	146

3.1 Introduction

Statistical graphs can offer very effective means for formally presenting the findings from a data analysis or simulation study. They are also an essential part of exploratory data analysis as we saw in Section 2.7. In this chapter, we consider the question of how to visually present data in informative and effective ways. We begin with an overview of the basic types of statistical graphs and how to select the appropriate graph given the type of variable(s) we are analyzing (Section 3.2). We also provide a framework and a set of guidelines for making effective plots (Section 3.3); these include considerations for making the data stand out in the visual presentation, ways to facilitate important comparisons

between groups or against a benchmark, and approaches for augmenting a visualization to create a context for interpreting the visual display. We demonstrate these concepts by providing several example visualizations, including addressing the issue of how to create visualizations for large data. These examples are presented without the code we used to create them. Later in Section 3.5, we introduce the two primary models in *R* for creating graphs – the painter’s model in base *R* and the grammar of graphics model in *ggplot2*.

Implicit in the guidelines for making statistical graphs is that we are attempting to convey a message in as clear and concise a manner as possible. This means that an important aspect of data visualization is discerning the message that we have discovered in our data and selecting an effective graph for conveying that message. Before we begin with an overview of plot types, we demonstrate the process of creating a visualization that best represents the message in the data.

3.1.1 Composing a Graph of Voter Registration Trends

There is no shortage of poorly designed statistical graphs. We use one as an example here because we can clearly separate the process of improving a graph by following the basic principles of plotting from the more nuanced considerations of how to best convey the underlying message in our data. These data are from online voter registration summaries published by the California Secretary of State at <http://www.sos.ca.gov/elections/voter-registration/voter-registration-statistics>. Figure 3.1 shows a bar plot created from data available at this site. A version of this plot first appeared online at swivel.com, which is no longer an active site.

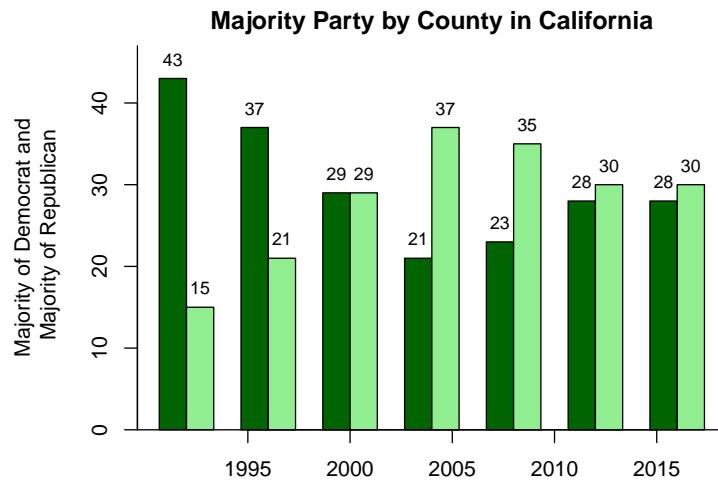


Figure 3.1: Distribution of California Voters by County. *The bar chart shown here imitates one that appeared on swivel.com. It has many flaws. Figure 3.2 fixes these problems, but does not address the question of how well the plot conveys the story in the data. Figure 3.3 replaces the bar chart with a more informative line plot that better conveys the message about changes in voter registration from 1992 to 2016.*

There are many obvious stylistic problems with this plot, e.g.,

- Tick marks: X-axis tick marks are at 5-year intervals and do not line up with the

locations of the bars so the viewer has to work too hard to figure out that the bars correspond to measurements made at 4-year intervals.

- Color: Atypical use of light and dark green for the Democratic and Republican parties (traditionally represented with blue and red, respectively).
- Legend: The lack of a legend means that we cannot discern which color represents which party.
- Axis Label: Y-axis label does not indicate what are the units of measurement.
- Title: Confusing title does not illuminate the content of the plot, i.e., what is meant by ‘Majority Party by County in California’?

The numbers on top of each bar help elucidate what are the data. We see that the heights of each pair of light and dark green bars sum to 58, which we determine (with an Internet search) to be the number of counties in California. Furthermore, we can visit the voter registration page where the data come from to determine which colors represent the Democratic and Republican parties. With this additional information in hand, we can address all of the concerns listed above. The revised bar chart appears in Figure 3.2. In this plot, we have located the year labels (1992, 1996, etc.) below each pair of bars, used the traditional red and blue for the 2 parties, added a legend to associate color with party, replaced the y-axis label with one that specifies the units of measurement, added an x-axis label to indicate that the years are presidential election years, and modified the title slightly (with all of the other changes the title now seems adequately informative).

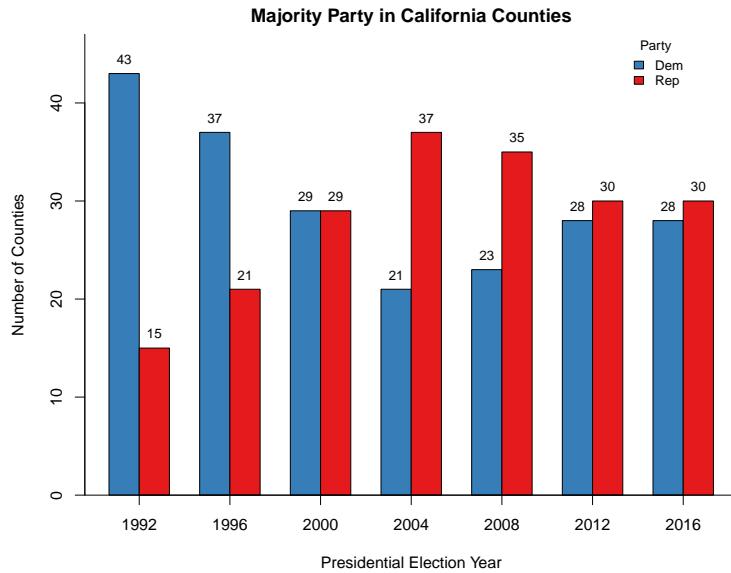


Figure 3.2: Bar Chart of Majority Party in California Counties – Revisited. *This bar chart addresses many of the problems found with the bar chart in Figure 3.1, including the inaccurate y-axis label, ill-positioned tick marks on the x-axis, poor choice of colors, and lack of legend.*

Before we declare Figure 3.2 a success, let’s ask ourselves what message was the creator of this plot trying to convey and is this the appropriate plot for doing so? It seems that

the graph is trying to show the change in voter registration over the past 7 presidential elections. However, it's people who register to vote, not counties. County size is a lurking variable—small counties tend to be rural and conservative—so counting counties overstates the Republican presence. Rather than count counties, let's make a plot that tallies voter registration. To do this, we revisit the registration Web site to obtain these figures. There we find that voters can decline to affiliate with a party and there are several other parties with which a voter can register. To effectively observe the trends in registration, we need to include these other possibilities in our plot because they are not insignificant. What kind of plot should we make? We have registration figures over time so a line plot seems appropriate. Also, given that the California population has grown dramatically in the past 25 years, rather than compare raw registration numbers, we scale them by each year's total registration and compare percentages.

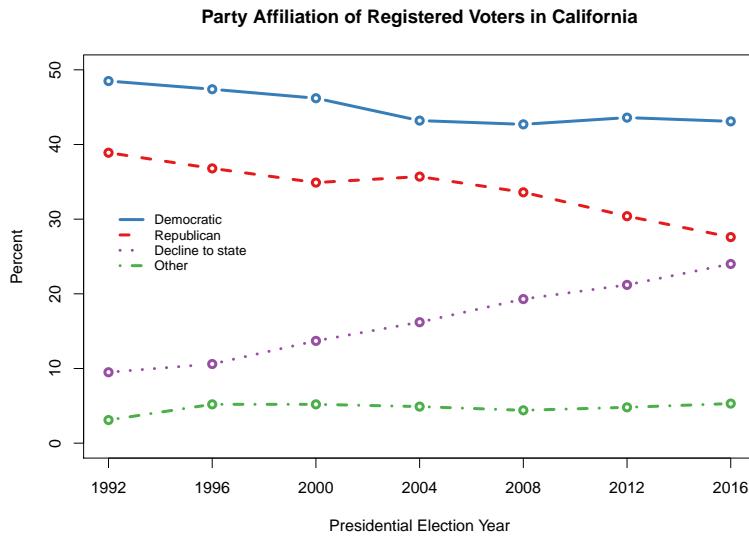


Figure 3.3: Distribution of California Voters by Party. *The line chart addresses the essential problem with the bar chart in Figure 3.2, i.e., we are interested in the change in voter registration over the years, not in the number of counties that are majority Republican or Democratic. Here we see that the percentage of registered Democrats and Republicans have declined over this 25-year period, the percentage of unaffiliated voters has dramatically increased, and that the gap between Democrats and Republicans has grown from about 10% to 15%.*

We have entirely overhauled the plot (see Figure 3.3). From this new graph, we get a more interesting and accurate depiction of the voter registration trends in California. We see that: the percentage of registered Democrats and Republicans have declined over this 25-year period; the percentage of Democrats was about 10% higher than the Republicans in the earlier years but the spread has grown recently to about 15%; and the percentage of unaffiliated voters has dramatically increased from about 10% to about 25% over this period. This aspect of making meaningful statistical graphs that accurately convey the story in the data follows from experience and experimentation, in addition to abiding by graphics guidelines (Section 3.3).

3.2 Data Types and Plot Choice

Chapter 2 introduced many of the basic plots within the context of cleaning and formatting data and carrying out exploratory data analysis. There we connected the choice of plot to the data type, and discussed how to read and interpret the various kinds of plots. In this section, we provide a brief overview of these basic plots. We do this in the context of one set of data that includes a range of variables and data types. These data are described in Section 3.2.2. We conclude this section by examining a second set of data with 100s of thousands of records, in order to address some of the considerations that arise when creating visualizations of relatively large amounts of data (Section 3.2.6). We begin with an introduction of the terminology we use to reference various components of a plot.

3.2.1 Terminology

The common terms we use to describe various pieces in a plot are shown in Figure 3.4. They include: the plot title, x and y axes and their respective labels, tick marks, and tick mark labels; within the plotting region, plotting symbols, reference lines, and labels on these lines and symbols; and a legend with its title, keys, and labels.

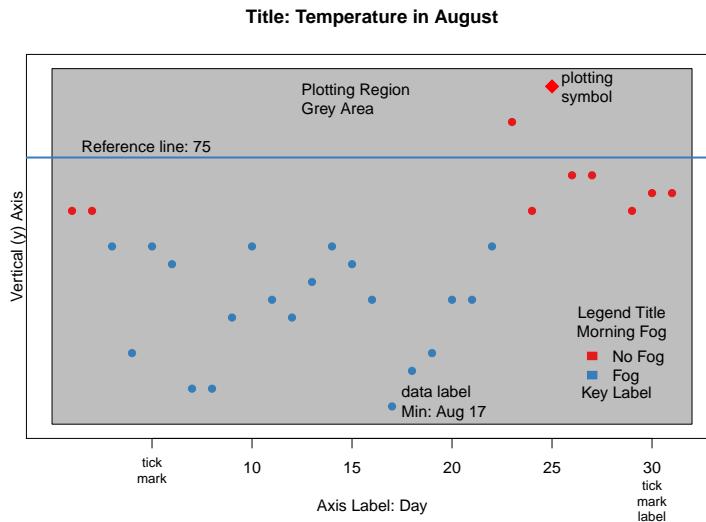


Figure 3.4: Graph Terminology. *This annotated plot provides a reference for the terminology we use in describing and critiquing plots.*

3.2.2 The Kaiser Family Data

The Child Health and Development Studies (CHDS) is a comprehensive investigation of all pregnancies that occurred between 1960 and 1967 among women who received prenatal care in the Kaiser Foundation Health Plan in the San Francisco–East Bay area and delivered at any one of the Kaiser hospitals in northern California. Over 15,000 families participated in the CHDS. The babies and their parents were followed through adolescence. The study had

TABLE 3.1: Infant Health Data Dictionary

Variable	Definition
id	identification number
date	birth date where 1096 = January1,1961
gestation	length of gestation in days
wt	birth weight in ounces (999 unknown)
parity	total number of previous pregnancies including fetal deaths and still births, 99 = unknown
race	mother's race 0-5 = white, 6 = mexican, 7 = black, 8 = asian, 9 = mixed, 99 = unknown
age	mother's age in years at termination of pregnancy, 99 = unknown
ed	mother's education 0 = less than 8th grade, 1 = 8th-12th grade – did not graduate; 2 = HS graduate-no other schooling; 3 = HS + trade; 4 = HS + some college; 5 = college graduate; 6 & 7 = trade school HS unclear; 9 = unknown
ht	mother's height in inches to the last completed inch, 99 = unknown
wt	mother prepregnancy wt in pounds, 999 = unknown
drace	father's race, coding same as mother's race.
dage	father's age, coding same as mother's age.
ded	father's education, coding same as mother's education.
dht	father's height, coding same as for mother's height
dwt	father's weight coding same as for mother's weight
marital	1 = married, 2 = legally separated, 3 = divorced, 4 = widowed, 5 = never married
inc	family yearly income in \$2500 increments 0 = under 2500, 1 = 2500-4999, ..., 8 = 12500-14999, 9 = 15000+, 98 = unknown, 99 = not asked
smoke	mother's smoking status: 0 = never, 1 = smokes now, 2 = until current pregnancy, 3 = once did, not now, 9 = unknown
time	If mother quit, how long ago? 0 = never smoked, 1 = still smokes, 2 = during current pregnancy, 3 = within 1 year, 4 = 1 to 2, 5 = 2 to 3, 6 = 3 to 4, 7 = 5 to 9, 8 = 10+ years ago, 9 = quit and don't know, 98 = unknown, 99 = not asked
number	number of cigarettes smoked per day for past and current smokers 0 = never, 1 = 1-4, 2 = 5-9, 3 = 10-14, 4 = 15-19, 5 = 20-29, 6 = 30-39, 7 = 40-60, 8 = 60+ cigarettes, 9 = smoke but don't know, 98 = unknown, 99 = not asked

several goals, one of which was to examine the effect of the mother smoking during pregnancy on the baby. The `babies` data frame provides a subset of this information collected for 1236 babies—baby boys born during one year of the study who lived at least 28 days and were single births (i.e., not one of a twin or triplet). The information available for each baby, including how the variables are coded, is provided in Table 3.1.

For background, the gestation period is reported in days and the typical gestation is 40 weeks, or 280 days. In the CHDS, the start date of the pregnancy is self-reported by the mother. Additionally, the father's height, weight, smoking status, and education are reported by the mother. At the time of the study, little was known about the adverse effects

of smoking on health. For example, the link between smoking and lung cancer was first reported by the Surgeon General in 1964, the first warnings appeared on cigarette packages in 1965, and the effects of smoking on the unborn had not been widely studied.

We have prepared these data for analysis by, e.g., converting values of 99 and 999 into NAs, formatting variables such as smoking status and education as factors, and collapsing levels with only a few observations into other levels. The cleaning and formatting process we carried out is outlined in the exercises of Chapter 2.

3.2.3 Univariate Plots

We described in Chapter 2 that when selecting a plot to create a visualization, we need to consider the data type. In particular, we determine whether or not the variable represents a quantitative or qualitative measurement. Although there are exceptions, the data type tends to dictate the kinds of plots most appropriate for the data values. We use plots to visualize the distribution of observations across the variable's values.

Quantitative Variables

With quantitative data, we want to know about the basic locations and number of high-density regions (i.e., those regions where a large fraction of the observations crowd together), whether or not there are a few observations with unusually large or small values, gaps where no data are observed, the size of tails, and the symmetry or skewness of the distribution. On the other hand, with qualitative variables, we tend to simply summarize the proportion (or counts) of observations that take on each possible category.

Take for example the baby's birth weight (measured in ounces) in the Kaiser study. Figure 3.5 shows 4 different statistical graphs of this variable, including (clockwise from top left) a rug plot, histogram, density curve, and normal-quantile plot. We describe each of these in turn.

Rug Plot

With only a few observations, the rug plot provides a simple representation of the distribution of a quantitative variable. In a rug plot, each observation is marked by a tick along the x-axis, i.e., 'a yarn in the rug'. With more than a handful of observations, the rug plot is typically not adequate for conveying the distribution because there's too much over plotting (ticks plotted on top of one another) or just too many ticks marks to see the shape of the distribution, i.e., we have trouble distinguishing between high and low density regions. There are more than 1200 babies in our data frame, and the rug plot in Figure 3.5 reveals little about the distribution of birth weight. Instead, we want to create a more informative representation of the distribution with a histogram or density curve.

Histogram

To make a histogram of the birth weights, we divide the x-axis into intervals that span the range of the data values. For each interval (or bin), we find the percentage of observations with values that fall into this bin and create a bar over the interval with an area that equals this percentage. Essentially, we are smoothing the observations evenly across each bin, i.e., we don't know the exact location of observations in a bin. The histogram of birth weight shows a unimodal distribution (one high-density region) that is centered at about 120 ounces. There do not appear to be any gaps or unusually large or small data values, the distribution looks roughly symmetric, and the tails appear neither long nor short.

The intervals in the birth weight histogram in Figure 3.5 are each 5 ounces wide. However, the intervals in a histogram need not be all the same width. As mentioned already, the defining property of the histogram is that the area of a bar equals the proportion of observations in the corresponding bin. Note that we can recover this percentage/proportion

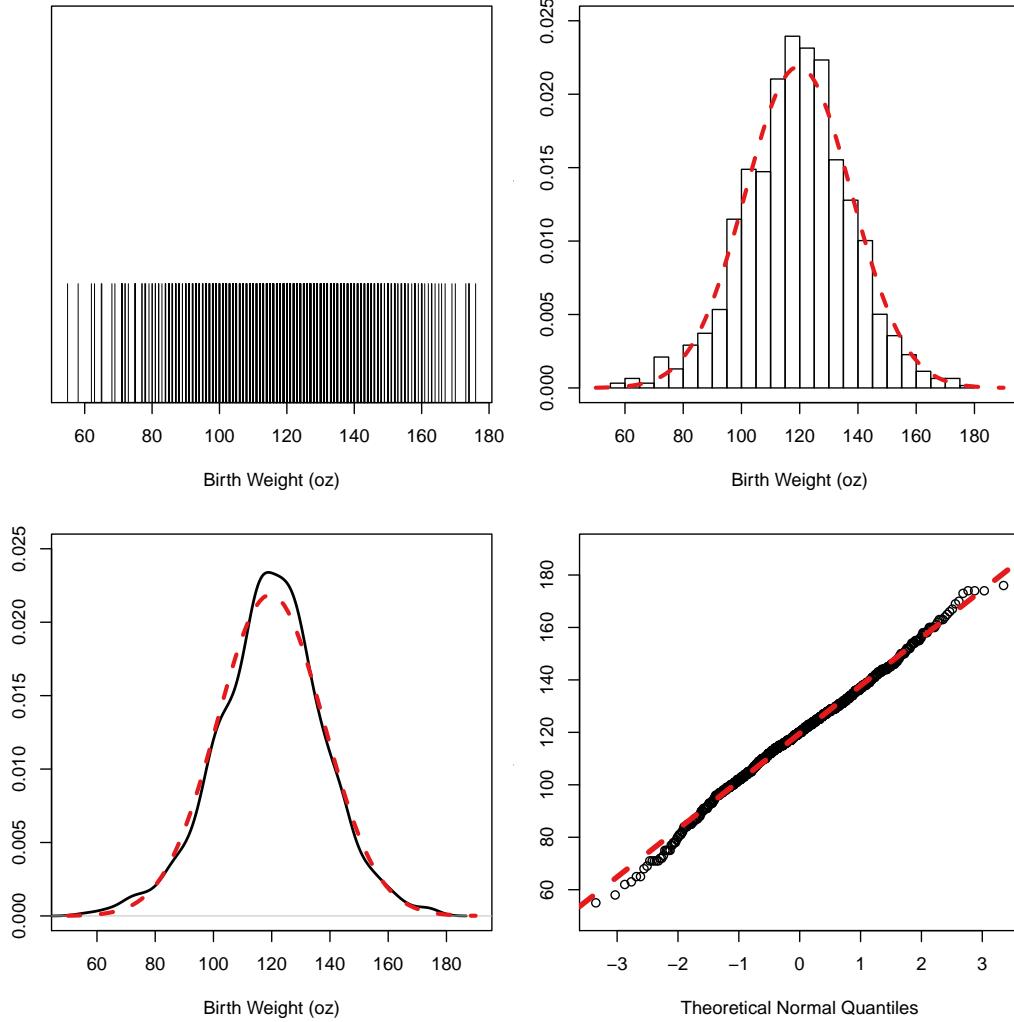


Figure 3.5: Distribution of Birth Weight. These 4 plots show the distribution of birth weight for a subset of the babies in the CHDS. These are (top left to bottom right) a rug plot, histogram, density curve, and normal-quantile plot. The rug plot shows values of individual observations but with so many records it's difficult to see the general shape of the distribution. The histogram and density curve reveal a similar shape—a unimodal, symmetric distribution. A normal curve is overlaid on the histogram and density curve to show that the distribution is roughly normal. However, the normal quantile plot offers a better visual comparison to the normal. Generally, the data quantiles and normal quantiles follow a straight line which indicates the data closely follow the normal curve. The downward curve on the left indicates the empirical distribution has a slightly longer left tail than the normal.

by multiplying the height of the bar by the width of the bar. In other words, the units of the bar's height is a density, such as percent per ounce.

Figure 3.6 provides another example of a histogram for the parity of the pregnancy, i.e., the number of previous pregnancies. Here 0 corresponds to the woman's first pregnancy, 1 to her second, etc. We describe the distribution of parity as unimodal with a peak at 0, skewed right (there is more area to the right of the mode than the left), a long right tail with some mothers having parity of 6 or more, and short left tail since it's not feasible to have values below 0. Notice that in this histogram, most bins are 1 unit wide, but the 3 rightmost bins are wider. They are 2, 3, and 3 units wide, respectively. We often use wider bins in the tails of a distribution to further smooth the data. The mother's **parity** is a discrete quantitative variable because only integer values are possible, i.e.,

```
table(babies$parity)
```

0	1	2	3	4	5	6	7	8	9	10	11	13
315	310	238	168	83	52	32	16	8	7	4	2	1

We have used a bin that combines the counts for 11, 12, and 13 in the histogram because the proportions for these values are low and we want to smooth them out over the distribution's tail.

Additionally, to make it clear that **parity** can take only nonnegative integer values, we center the bins on the integers, e.g., the bin from 1.5 to 2.5 contains those mothers with a parity of 2. For baby's birth weight, since the measurements are to the nearest ounce, we need to know the interval convention, i.e., whether the bin from 120 to 125 is open on the left and closed on the right or closed on the left and open on the right, in order to determine whether the bin includes the babies with a birthweight of 120 or 125. This distinction is less important because these measurements ideally can be measured to a finer precision.

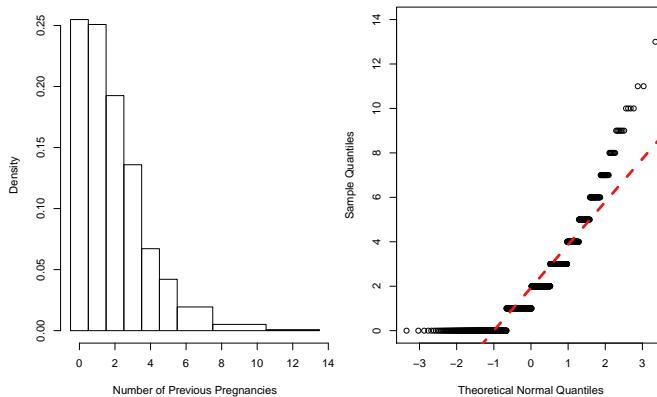


Figure 3.6: Histogram of Parity of the Pregnancy. *Parity has a discrete distribution (only integer values are possible). The histogram (left) reveals the distribution is unimodal, skew right, with a long right tail where a few women had 6 to 13 previous pregnancies. The steps in the normal-quantile plot (right) are due to the discreteness in the distribution. The very long initial step indicates the empirical distribution has no left tail and the curvature on the right indicates a long right tail.*

Density Curve

The density curve provides a smooth representation of a distribution. With the density

curve, the area under the curve for a particular interval approximates the proportion of values in that interval. (The total area under the curve is 1.) The density curve for birth weight (bottom left in Figure 3.5) has a similar shape as the histogram in that figure. We again see a unimodal distribution that is centered at about 120 ounces, symmetric, with neither long or short tails. Note that we do not make a density curve for parity because of the discrete nature of the values for this variable.

Normal Curve

We overlaid a normal curve on both the histogram and density curve for birth weight in Figure 3.5. The normal curve is often used as an idealization of an empirical distribution. It can align well with the distribution of a quantitative characteristic of a population, such as the height of fathers, width of Dungeness crab shells, weight of locally grown tomatoes, and circumference of trees. When we match a normal curve to data, we choose from a family of similarly shaped curves, where each curve is characterized by its center (point of symmetry) and spread. The normal curve that is super-posed on the histogram and density curve in Figure 3.5 has a center that matches the average birth weight and a spread that matches the SD (standard deviation) of birth weight. (Recall that the SD is a measure of spread, which is defined as the square root of the average of the squared deviations from the observations to the average, i.e., in R it is

```
sqrt(mean( (bwt - mean(bwt))^2 ))
```

These 2 quantities, center and spread, completely determine a normal curve. Also, when we describe the length of a distribution's tails, we often compare them to the tails of a normal curve.

Normal-Quantile Plot

The normal curve that is layered over the histogram and density curves in Figure 3.5 appears to follow the histogram and density curve quite closely. However, a normal-quantile plot offers a better visual comparison (lower right, Figure 3.5). In this plot, we see that the points roughly follow a line, which indicates that the data roughly follow the normal curve. Generally, a normal-quantile plot compares the quantiles of the data to those of the normal curve. That is, the scatter plot consists of quantile pairs: (q th quantile of the idealized normal curve, q th quantile of the data), for many q s between 0 and 1. If the data's distribution is close to the normal curve, then the points roughly fall on a line. Deviations from a line indicate differences between the distributions. We have added a line to the plot with intercept and slope that match the mean and SD of our data because the normal quantiles used in the plot are for the standard normal (mean 0 and SD 1).

As another example, we make a normal-quantile plot for parity (on the right in Figure 3.6). Most noticeable are the steps in the plot. These are due to parity taking only discrete values. The birth weight measurements are discrete in the sense that weight is measured to the nearest ounce, but the steps from the duplicate values are not noticeable given the range of data values. We saw in the histogram that this distribution is clearly not normal because it is skewed right and has a long right and short left tail. These features appear in a normal quantile plot via curvature; specifically, the short left tail is indicated by the long step at the lower left, and the long right tail is evident from the upward turn on the right end of the curve.

More generally, we can make a quantile-quantile plot that compares the quantiles of any two distributions, including comparing observed quantiles from two sets of data (see Figure 3.10) and comparing the empirical quantiles from data to the quantiles of a theoretical distribution other than the normal. See the exercises for more practice with reading quantile plots.

Qualitative Variables

A qualitative variable takes on a fixed and finite set of possible values, where each value corresponds to a category. For example, in the Kaiser study, mother's smoking status can belong to one of 5 possible categories : she never smoked, smoked during pregnancy, smoked until she was pregnant, smoked once but not now, or has an unknown smoking status. We are interested in the distribution of smoking status, i.e., the proportions in each category. (We exclude those mothers with an unknown smoking status). We can compute these proportions with

```
table(babies$smoke) / sum(table(babies$smoke))
```

	Never	Current	Until	Once
	0.444	0.395	0.077	0.084

These proportions can be arranged in several different visual formats, such as a bar plot, dot chart, and pie chart, as shown in Figure 3.7.

Bar Plot

Unlike a histogram, the area of the bar in a bar plot has no meaning—only the height of the bar conveys the distributional information. For this reason, the 2 bar plots in the top row of Figure 3.7 are equivalent. In both we see that there are nearly equal numbers of mothers who never smoked and those who smoked during pregnancy. We also see that less than 10% of the mothers quit smoking when they became pregnant and about the same proportion quit smoking before becoming pregnant.

Dot Chart

The dot chart (bottom right of Figure 3.7) takes the lack of meaning in the width of a bar in a bar chart to its extreme conclusion and eliminates the bars entirely. In a dot chart, the proportion of mothers in each category are located along the category's respective line. Again, we see clearly that the smokers make up slightly less than 40% of the mothers and the never smokers about 44%. The bars in a bar chart and dots in a dot chart can be arranged vertically or horizontally. The dot chart here is arranged horizontally while the bars are arranged vertically in the 2 bar plots.

Pie Chart

The pie chart (bottom left of Figure 3.7) conveys these proportions through angles in the pieces of pie. For example, the slice of pie corresponding to the Never group makes up 44% of total pie. Comparisons between slices in a pie can be difficult to make, e.g., it's difficult to discern that the Current slice of pie is smaller than the Never piece. In general, we can more accurately compare lengths of bars and locations of dots on a line than angles in slices of pies so bar charts and dot charts are typically preferred over pie charts.

3.2.4 Bivariate Plots

When we have more than one variable, we typically want to examine the relationship between variables. For example, we may want to observe the relationship between the heights of mothers and fathers to see if, for example, mothers who are above average in height tend to have partners who are above average in height.

We may also want to examine the relationship between 2 categorical variables to see, for example, if the distribution of never and current smokers is the same across different levels of education. In general, more highly educated mothers tend to live healthier lives which can impact the health of the new born and we want to see if this relationship is born out in the data.

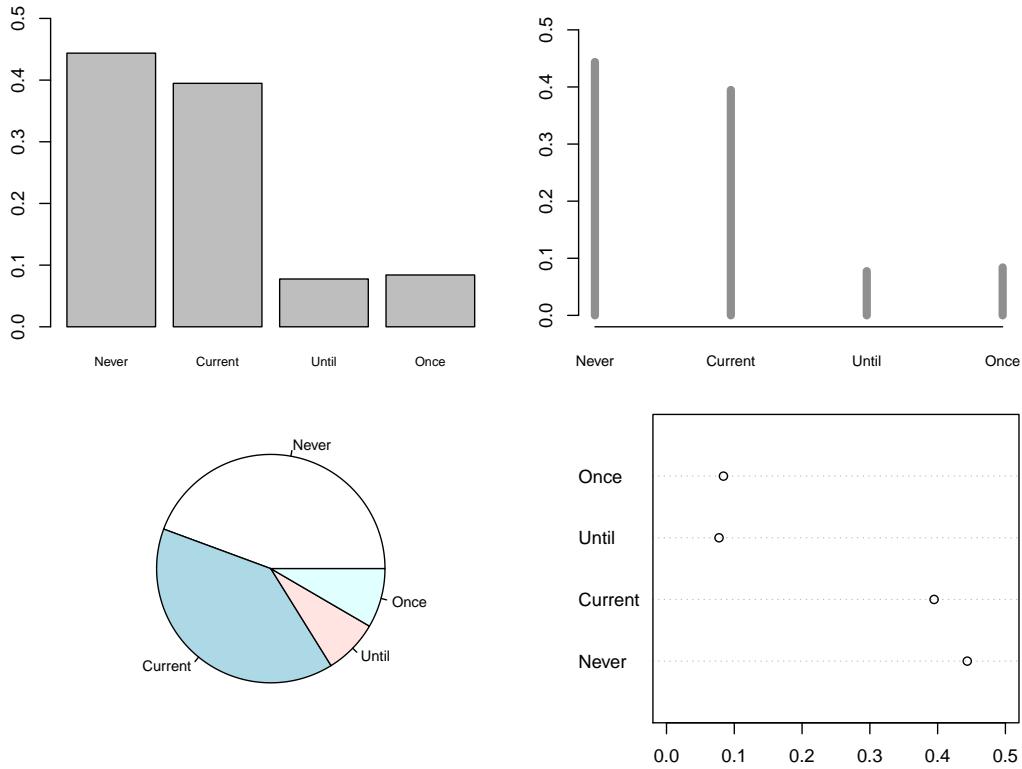


Figure 3.7: Distribution of Mother's Smoking Status. *These 4 plots display the proportion of mothers in each smoking status via (top left to bottom right) bar charts with wide and narrow bars, pie chart, and dot chart. The labels denote whether the mother Never smoked, Currently smokes (in pregnancy), smoked Until she became pregnant, or smoked Once and quit before pregnancy. The bar and dot charts use length to represent the proportion of each type of smoker. The pie chart uses angles which are typically harder to compare accurately.*

A core interest of ours is in the relationship between the mother's smoking status and her baby's birth weight; that is, we want to examine the relationship between a qualitative and quantitative variable. To do this, we can compare the distribution of birth weight for the group of smokers and never smokers to see if these distributions are the same. If not, we want to know how they differ. We provide examples of bivariate plots that can begin to address these various situations.

Scatter Plot

Mother's and father's height are both quantitative variables. With quantitative variables, we typically use a scatter plot to examine their relationship. For the scatter plot in Figure 3.8, each point corresponds to a (mother, father) pair. The scatter of points display a weak positive association; in other words, mother's above average in height tend to be associated with father's who are also above average in height, but there are many exceptions to this 'rule'. We can compute the linear correlation between these variables with

```
cor(babies$ht, babies$dht, use = "complete.obs")
```

```
[1] 0.34
```

Correlations range between -1 and +1 with values of ± 1 indicating a perfect linear relationship and a value of 0 indicating the lack of a linear relationship. The correlation of 0.34 is weak, but it does indicate that there is a small positive association. However, since the correlation is small, there is a great deal of variability in the father's height among the mothers with the same height. We also note that a correlation coefficients can be strong even when the relationship between 2 variables is clearly not linear so it is good practice to plot the data in order to view the shape of the relationship.

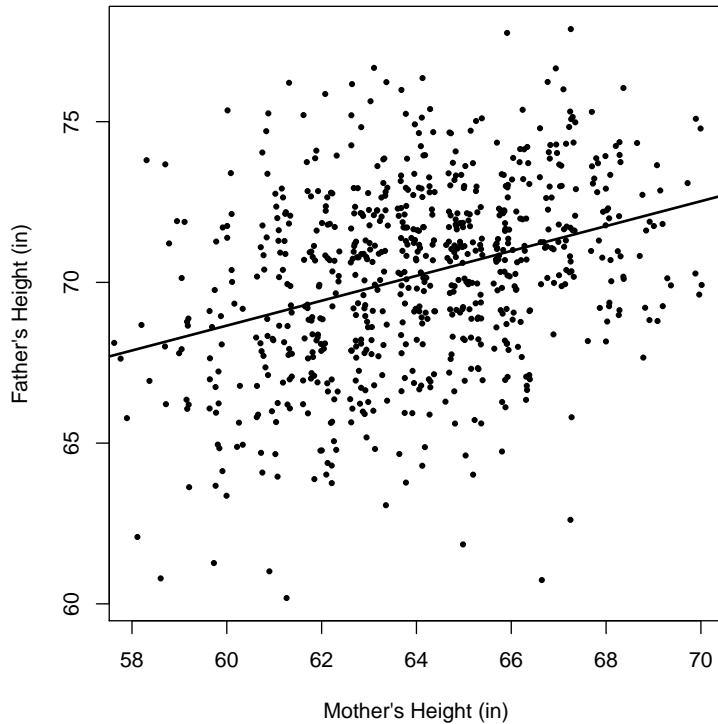


Figure 3.8: Heights of Mothers and Fathers. *This scatter plot shows a weak linear association between the heights of mothers and fathers in the Kaiser study. The line fitted to these points has a slope of about 0.39 and an intercept of about 45. According to this line, a mother who is 64 inches tall tends to be with a father 70.2 inches tall on average, and a mother 2 inches taller tends to be with a father 71 inches tall on average. The correlation is 0.34, indicating the variability of father's height for mothers who are, e.g., 64 or 66 inches tall is quite large.*

Two Qualitative Variables

When we examine the relationship between 2 qualitative variables, we examine proportions of one variable within the subgroups defined by the other variable. For example, with education and smoking status, we can compare the proportion of never, current, until pregnant, and once smokers within each education level with

	< 12th	HS	Trade	Some	Col	College
Never	0.345	0.459	0.371	0.466	0.498	
Current	0.530	0.400	0.486	0.345	0.297	

Until	0.065	0.091	0.043	0.071	0.082
Once	0.060	0.050	0.100	0.118	0.123

Alternatively, we can compare the proportion of some high school, high school, trade school, some college, and college educated mothers across each smoking status with

	< 12th	HS	Trade	Some	Col	College
Never	0.126	0.371	0.048	0.254	0.200	
Current	0.219	0.364	0.070	0.211	0.135	
Until	0.137	0.421	0.032	0.221	0.189	
Once	0.117	0.214	0.068	0.340	0.262	

Which of these tables of proportions is most helpful? The answer depends on what we consider to be the important comparison. If we condition on education level and examine the distribution of smoking status for each level of education, then we can compare education levels across smoking status. We associate education level with healthy life choices, e.g., alcohol consumption, so we want to see how education level varies across smoking status. This comparison corresponds to the proportions in the first table. In Figure 3.9, we have created (clockwise from top left) a mosaic plot, side-by-side bar chart, line plot, and stacked bar chart from this table.

Mosaic Plot and Stacked Bar Chart

The stacked bar chart is the most difficult to read because, for example, the top and bottom of each sub-rectangle that corresponds to a particular education level moves up and down across smoking status. The mosaic plot has a similar issue but it has several advantages over the stacked bar plot. The width of the vertical slices correspond to the proportion of smoking levels among the entire population under study; the gaps between the vertical and horizontal slices assist in the comparison; and the shading of a rectangle indicates whether the proportion is more (blue) or less (red) than expected under the assumption of there being no relationship between the two variables.

Side-by-Side Bar Chart and Line Plot

The line and side-by-side bar chart are very similar in how they convey information. In the side-by-side bar chart, the bars for education status are grouped together for each level of smoking. Also, since education is an ordered qualitative variable, the bars are arranged according to this order. The line plot connects dots for each education level across smoking status. The line plot has two main advantages over the bar chart: the dots for education level within each smoking status appear directly above/below one another, and the dots for the same education level are connected across smoking status. Both these features make it easier for us to make comparisons. We can more easily see whether the ordering of education level remains the same across the smoking status and if the differences between education grow or shrink with smoking status.

One Qualitative and One Quantitative Variable

When we examine the relationship between one quantitative and one qualitative variable, we use the qualitative variable to divide the data into groups and compare the distribution of the quantitative variable across the groups. For example, we can compare the distribution of birth weight for babies born to mothers with different smoking statuses.

Quantile-Quantile Plot

We can compare 2 quantitative distributions with a quantile-quantile plot. That is, we can plot pairs of quantiles from 2 empirical distributions (this is in contrast to the normal-quantile plot where we plot the quantiles of an empirical distribution against those of the

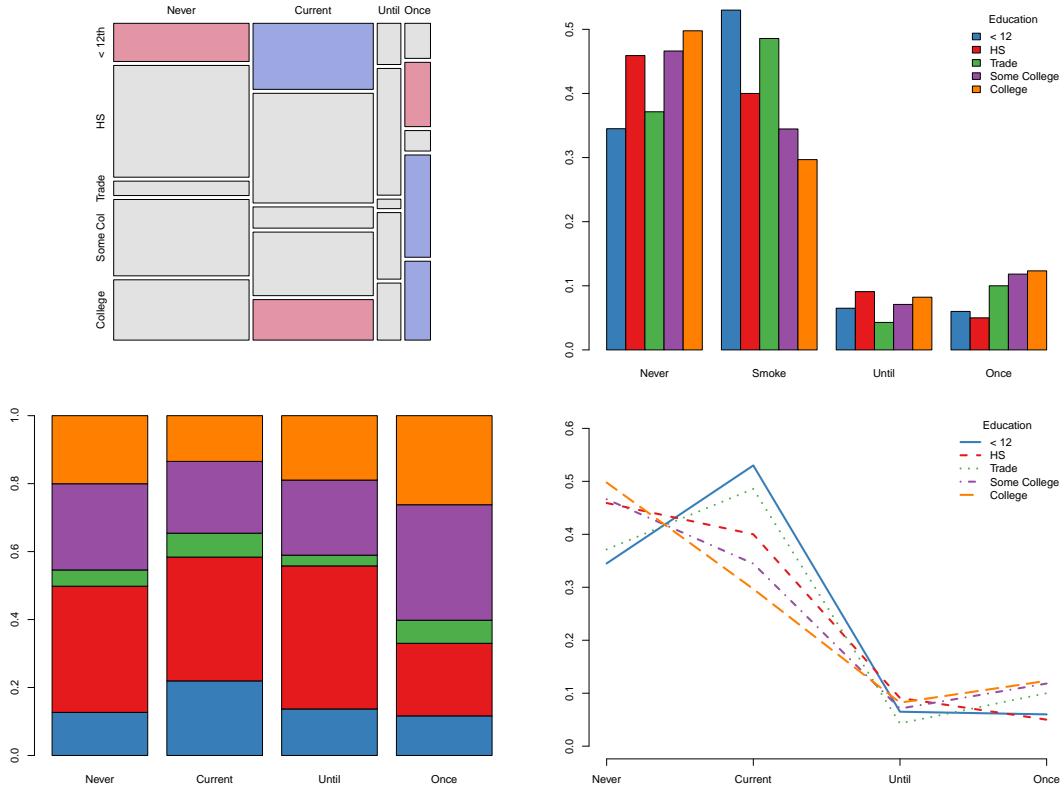


Figure 3.9: Mother's Smoking Status and Education. The 4 statistical graphs in this figure offer alternative approaches to comparing education level for different types of smokers. These are (from top left to bottom right) the mosaic plot, side-by-side bar chart, stacked bar chart, and line plot. Each plot displays the proportion of mothers with a particular education level within smoking status, i.e., the proportions for high school education add to 1 across the 4 smoking statuses. The line plot has advantages over the bar chart due to the close proximity of education level values within a smoking status and to the connecting line segments that help compare education level across smoking status. The stacked bar plot is problematic because the rectangles for a particular education level are not aligned, i.e., both the lower and upper sides of the rectangles move up and down across smoking status.

theoretical normal). For example, in Figure 3.10 (top left), we compare the distribution of birth weight for never smokers and current smokers. Here we see that the points roughly fall on a line, which indicates the distributions have roughly the same shape. However, this line is shifted down from the reference line that has intercept 0 and slope 1, which points to a shift left in the distribution of weight for the smokers' babies in comparison to the never smokers.

Super-posed Density Curves

Alternatively, we can super-pose (place on the same plot) the density curves for each subgroup (Figure 3.10, top right). There we confirm that the density curve for the smokers is shifted to the left of the curve for the never-smokers. It also appears that the spread in birth weight is larger for the smokers and that the never-smoker distribution has longer tails than the smoker group. How do these features appear in the quantile-quantile plot? Note the slope of the points and the curvature at the extremes. We see that the smaller spread for never-smokers is reflected in the slightly steeper line of points, and the unusually large and small values for the never-smokers appears in the curvature of the points at the two ends of the birth weight values. However, the shift seems the most pronounced difference between the never and current smokers.

Side-by-Side Box Plots and Violin Plots

Yet another approach compares summary statistics for the subgroups with side-by-side box plots. Recall that the box marks the lower and upper quartiles and the line in the interior of the box denotes the median. Whiskers are drawn to the nearest observation that is within 1.5 IQRs of the quartiles and points beyond that are marked individually. Figure 3.10 contains box plots of birth weight for all 4 smoking statuses. We see that the IQR for the never smokers is about 5 ounces smaller than the IQR for the current, until pregnant, and once smokers. Also evident are the large number of outliers in the never smoked group; there are many unusually small and large birth weights in this group. Alternatively, we can juxtapose 4 density curves in a violin plot. In the bottom right of Figure 3.10, the density curves for 4 subgroups of mothers are plotted vertically and reflected about their respective axes to create violin-shaped plots that are similar in purpose to box plots.

3.2.5 Conveying Relationships between 3 or More Variables

We can use the plots described already to create 2-dimensional visualizations for 3 (or more) variables. For example, if we have 3 or more qualitative variables, we can subdivide the data according to the combinations of levels of these variables and compare proportions with line plots, dot charts, side-by-side bar charts, and mosaic plots as in Figure 3.9. We typically organize the lines, dots, and bars into groups according to a combination of levels from two (or more) variables. We can examine the relationship between one quantitative variable and 2 or more qualitative variables with side-by-side box/violin plots or overlaid density curves. Again, the box/violin plots are organized according to the combination of categories of the qualitative variables. The specific organization depends on which comparison we want to focus on, similar to the decision made when we compared education levels across smoking status (Figure 3.9). With 2 quantitative variables and 1 or more qualitative variables, we often make scatter plots where the points are color coded according to the categories of the qualitative variable or where the shape of the plotting symbol denotes the category. Additionally, we juxtapose scatter plots in a grid (keeping the range of the axes the same across the various plots).

Figure 3.11 is an example where we examine three variables: birth weight, mother's height, and smoking status. The scatter plot displays the relationship between the baby's birth weight and mother's height. As with the scatter plot of mother's and father's height,

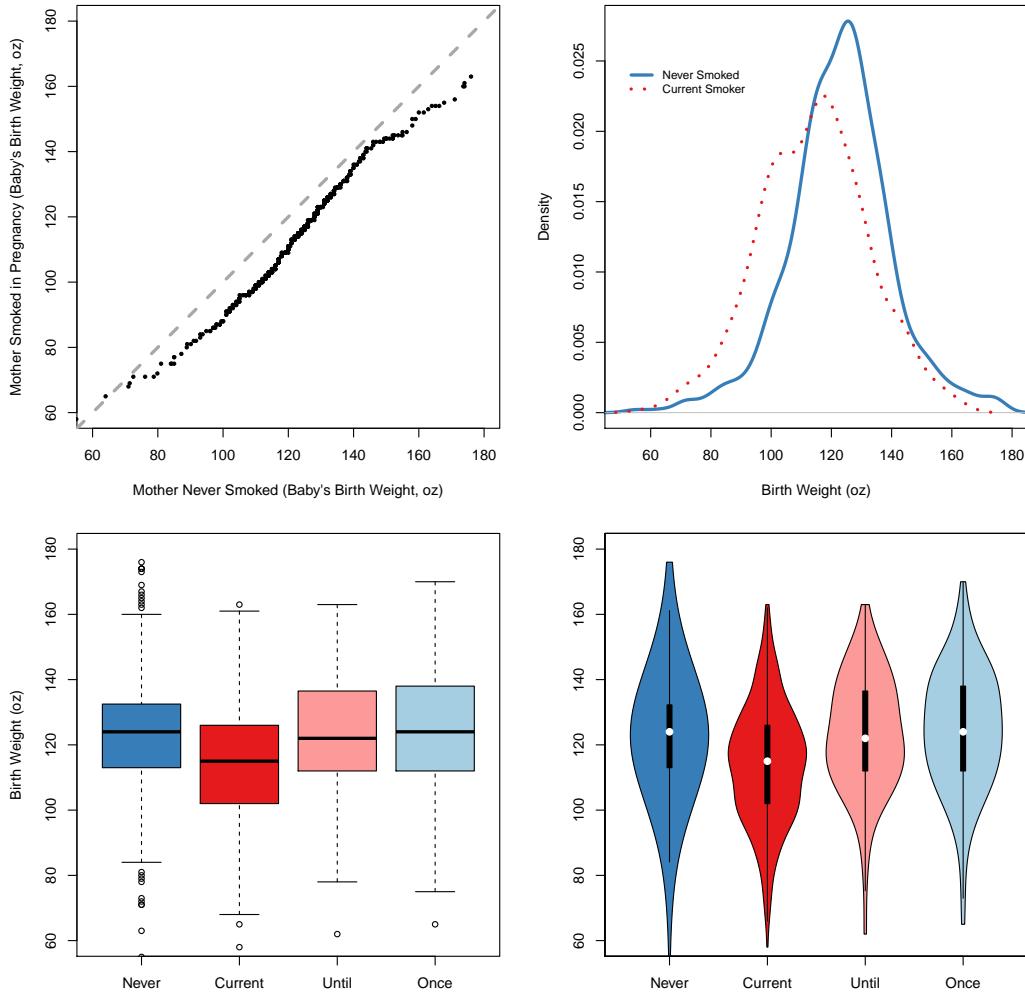


Figure 3.10: Mother's Smoking Status and Baby's Birth Weight. These plots compare the distribution of birth weight for mothers with different smoking statuses. A comparison of the birth weight of babies born to mothers who never smoked versus those who smoked during their pregnancy is made with a quantile-quantile plot (top right). The quantiles roughly fall on a line which indicates similar distributional shape. The straight line added to this plot has slope 1 and intercept 0 and can be used to discern differences between the distributions of these two groups. Particularly, the downward shift of the points in comparison to this line indicates a shift to lower birth weights for smokers. The plot on the top right overlays (or super-poses) the density curves for these 2 groups and confirms the noted shift. The birth weight distribution for all 4 smoking statuses are provided in side-by-side box and violin plots (bottom left and right, respectively). It appears that those who quit smoking have a distribution with a median that matches the non-smokers but a slightly larger variability.

we find a weak linear association. The points in the scatter plot have been color coded according to whether the mother never smoked (blue) or currently smokes (red). We have added two curves to this plot, the red dashed line for the smokers and the blue solid line for the never smokers. For each group of mothers (never or current smokers) the curve displays the average birth weight of the babies born to mothers of the same height. We see that the both curves are roughly linear, indicating the taller mothers tend to have heavier babies. We also see that the curves have roughly the same slope but curve for the smokers is about 10 ounces lower than for the never smokers.

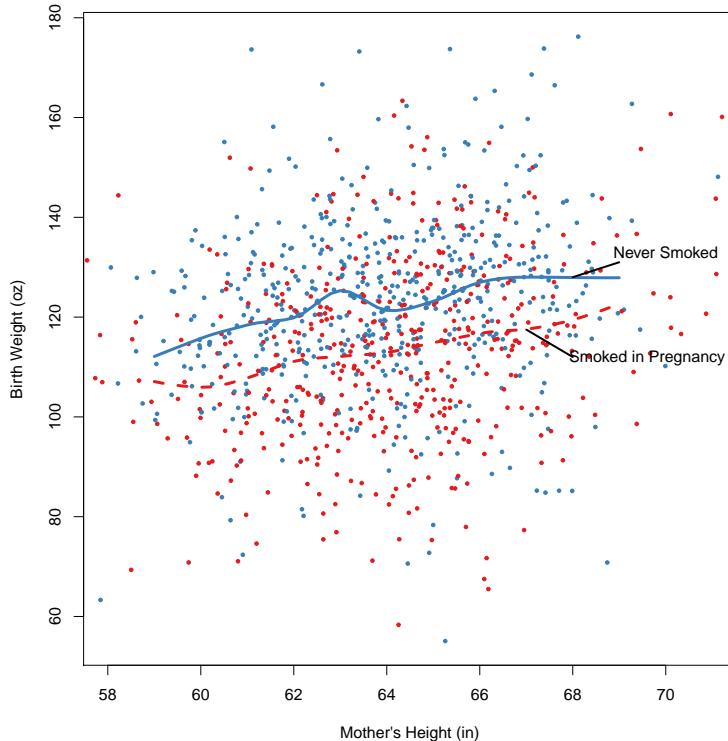


Figure 3.11: Baby's Birth Weight and Mother's Height. *Mother's height and baby's birth weight have a weak linear relationship with a correlation coefficient of about 0.20. The points in this scatter plot are color-coded according to the mother's smoking status (blue for never smoked and red for smoked during pregnancy). Two curves added to this plot show the average birth weight for babies born to mothers with the same height (red dashed is the average for mothers who smoked during pregnancy and solid blue for mothers who never smoked). We see that the relationship between birth weight and mother's height is roughly linear for both groups of mothers and babies born to never-smokers consistently weigh about 10 oz more than those born to current-smokers for all heights.*

Visualizations of 3 or more quantitative variables are more difficult to make. We often create scatter plots of all pairs of variables. However, plots of pairs of variables may not be adequate for revealing higher dimensional relationships. We can also create a scatter plot with 2 of the variables and color the points according to the level of the 3rd variable. Another technique, multi-dimensional scaling, creates a scatter plot that tries to maintain

TABLE 3.2: San Francisco Housing

Variable	Definition
county	County name
city	City name
zip	Zip code
street	Street address
price	Sale price in dollars
br	Number of bedrooms
lsqft	Size of lot (ft^2)
bsqft	Size of building (ft^2)
year	Year house was built
long	Longitude of house location
lat	Latitude of house location
wk	Week sale reported by SF Chronicle, e.g., "2003-04-21"

the distances between all pairs of observations, where the distance between two observations is computed with all of the variables. (See Section 13.9 for an example).

3.2.6 Scalability – Real Estate Data with 500,000 records

When we have a large number of observations, some of the plots we have described can be problematic for displaying the relationship between variables. As an example, we examine a catalog of houses sold between April 27 2003 and November 16 2008 in the San Francisco Bay Area. For each of the 500,000 sales, we have the sale date and price along with the house location, lot size, building size, number of bedrooms, and year it was built. See Table 3.2 for a description of the variables and their units of measurement. These data were scraped from the San Francisco Chronicle web site [2]. (See also [13] for an analysis of these data, [1] for an example of how to scrape these data from the Web, and the exercises in Chapter 2 for how to clean and format the `housing` data frame for analysis.)

Over plotting is the main problem with large amounts of quantitative data. For this reason, rug plots are not feasible. Instead, we create a histogram or density curve to view a variable's distribution. We can alleviate the problem with over plotting in a scatter plot in a variety of ways. We can: divide the data into sub groups and juxtapose scatter plots of these groups; examine a subset of the data where particular variables are held constant; or make plots of summaries or aggregates of the data, rather than individual points.

For example, we can select one year of data, say 2004, and a subset of cities, say those within 10 miles of Berkeley. With this reduced collection of sales, the data are likely to be more similar because housing prices don't change as much in 1 year in comparison to 8 years and the houses are in a smaller geographic area. Figure 3.12 shows side-by-side box plots of sale price in the 12 neighboring cities. Rather than arranging the box plots alphabetically according to city name, they have been arranged according to median sale price. Notice that we have also plotted price on a log scale in order to accomodate the tremendous range in prices. We might consider removing a few outliers so that we can zoom in on the bulk of the data. Nonetheless, it is clear that houses in Piedmont are clearly the most expensive (Piedmont is a small city surrounded that has many large, stately homes and schools with high test scores.)

With this same subset, we examine the relationship between the sale price and size of

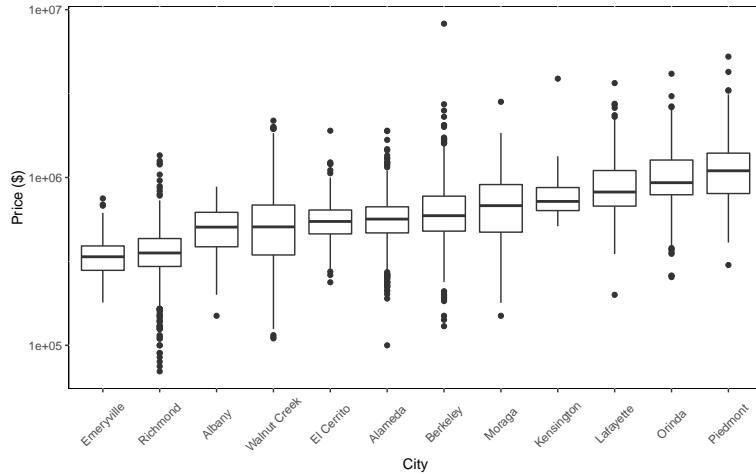


Figure 3.12: Housing Sale Prices by City. These side-by-side box plots of sale price are arranged in order according to the median sale price in the city. Price is plotted on a log-scale. Included here are sales for 2004 from 12 cities within 10 miles of Berkeley, California.

the house with a set of scatter plots. Rather than examine sale price directly, we compute the price per square foot of the building. We have juxtaposed 12 scatter plots, one for each city, in a 3 by 4 array in Figure 3.13. Notice that the range of the x- and y-axes remain the same across all 12 scatter plots to help us compare cities.

We also added a smooth curve that shows the relationship between price per square foot and building size. The curve averages price per square foot given building size for houses in all 12 cities (so the curve is identical on all 12 plots). This curve shows a general trend and helps compare cities across plots. The curvature indicates that smaller homes cost more per square foot than large ones, i.e., there is an entry cost to buying a home and the cost of additional space is cheaper.

Furthermore, we made the color of the points in the scatter plot partially transparent so that overplotting is ameliorated to some extent. The color corresponds to the number of bedrooms. It's evident that the 1 bedroom houses are more prevalent in the cities with lower sale prices.

For another approach, we can include all of the 2004 data for all 12 cities in one plot, as in Figure 3.14 (left plot). Now there are too many observations to color code the points by the number of bedrooms. Instead, we color code the points according to density. This way, we visualise the shape of the bivariate distribution of price per square foot and building size. That is, the deep red/purple area has the greatest density of sales; the peak is located at about 1250 square feet and \$350 per square foot (which corresponds to prices from about \$350,000 to \$400,000).

To make yet another kind of summary, we can plot smooth curves rather than individual points. The plot on the right of Figure 3.14 shows no individual points, and instead displays curves of the average price per square foot as a function of building size. We have superposed 12 curves, one for each of our cities. From these curves, we see that each city follows this pattern of a high entry point and decreasing cost per square foot for larger homes. We also see that the curves for all of the cities are roughly the same shape and those for the more expensive cities are shifted higher than the cheaper cities. The lumpiness in a couple of these curves, e.g., Moraga, may be due to the number of houses sold in that size range.

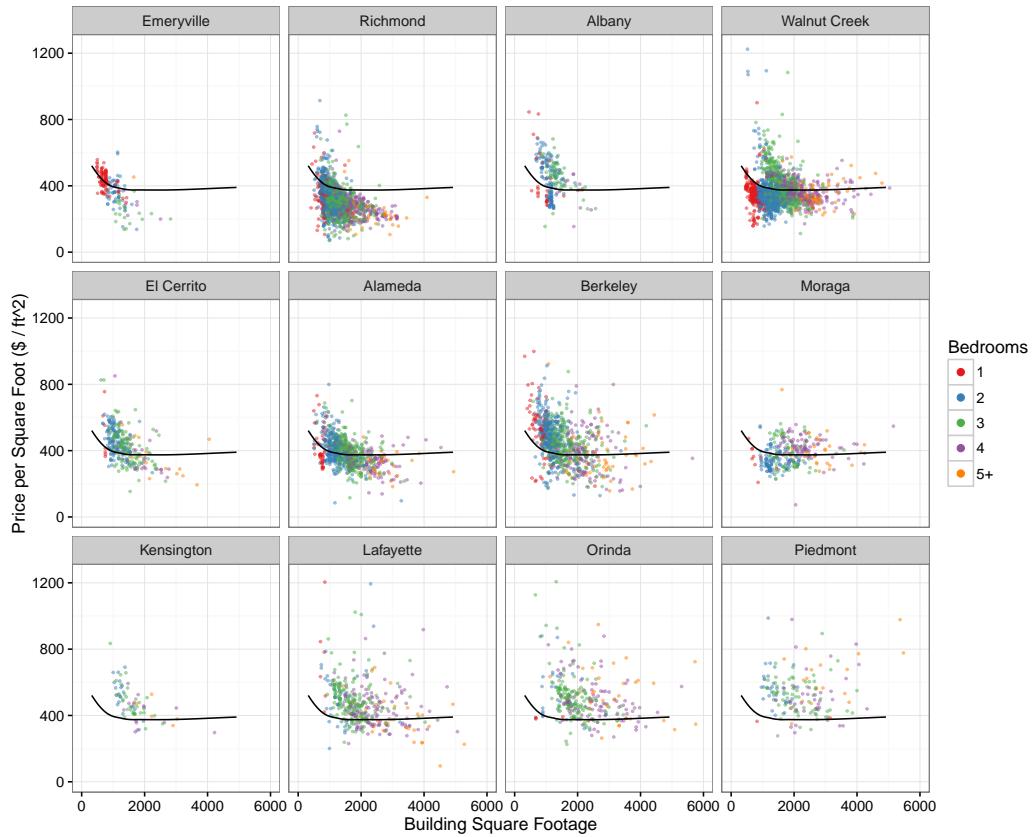


Figure 3.13: Sale Price and Building Size for 12 Cities. Plotted here are the price per square foot against the size of building for all houses sold in 2004 in 12 cities within 10 miles of Berkeley, California. The data for each city appear in separate (juxtaposed) scatter plots, which are arranged from least to most expensive. The plots have common x-axis and y-axis scales across the 12 plots, and the curve of average price as a function of building size for all cities is super-posed on each plot. The points are color-coded according to the number of bedrooms in the house.

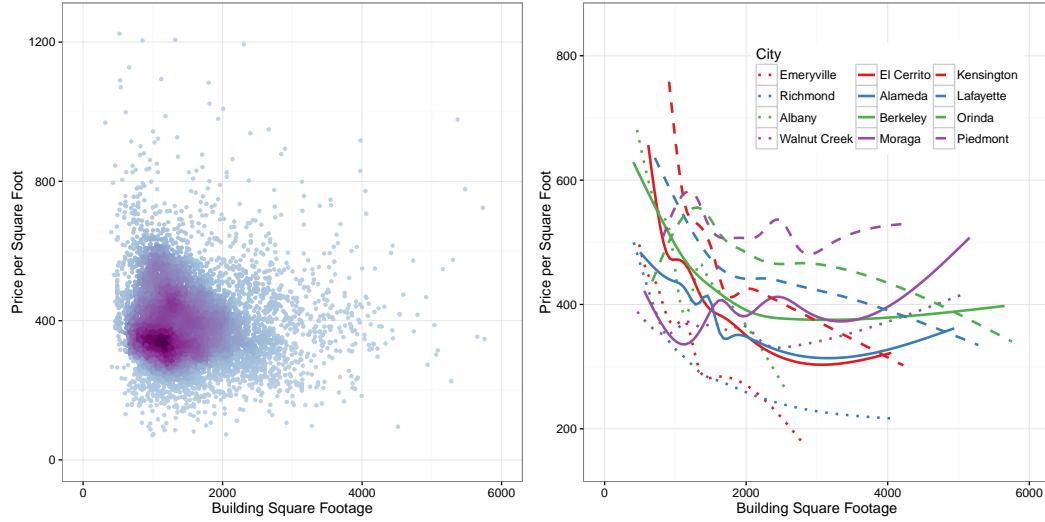


Figure 3.14: Sale Price and Building Size. The scatter plot (left) shows the relationship between the price per square foot and size of building for all houses sold in 2004 in 12 cities in the East San Francisco Bay Area. The points are color-coded according to the density of the number of houses in each price-size combination. The smooth curves (right) show the average price per square foot as a function of the building size for these 12 cities. The curvature in the relationship between price per square foot and building size is similar across all 12 cities with the cities with higher median sale prices remaining consistently above the other cities.

3.2.7 Measurements in Time

The original data set contains weekly sale prices for 6 years so we may want to observe trends in prices over time. When we include time as a variable in a plot, we typically place it along the x-axis and make a line plot that connects the y-values across time. In this case, we have multiple measurements for each time, i.e., all of the sales for the week, so we make a time plot with an aggregate of sale price for each week. Given the skewness of the distribution of sale price, we typically summarize sale price by a median rather than a mean. However, we can also examine other quantiles of the data. In Figure 3.15 (modeled after a figure in [13]), we make a line plot for each of the 9 weekly price deciles. That is, the 10th percentile, 20th percentile, ..., and 90th decile of sale price for each week's sales. Housing prices were seen to fluctuate tremendously over this short time period as the housing market boomed and crashed. One way to explore the impact on the different priced homes is to normalize the weekly deciles by their value at the start of our window of observation. In other words, we divide the weekly 10th percentile of sale prices by the 10th percentile on Apr 27, 2003, and similarly scale the other percentiles. These normalized percentile line plots appear in Figure 3.15. They are color-coded with a grey-scale that ranges from dark grey for the 10th percentile to light grey for the 90th percentile. It's evident from this plot that the lower priced homes suffered the greatest volatility in this period. They proportionally surpassed the higher-priced homes in 2006 and lost the greatest value in 2008.

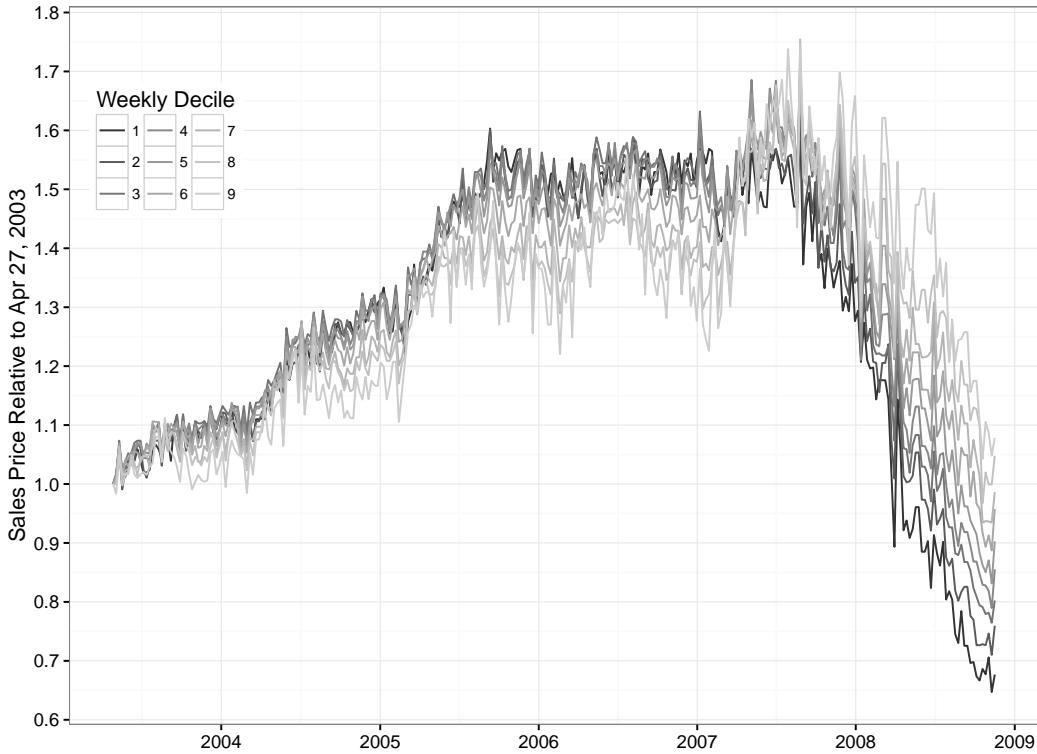


Figure 3.15: Deciles of Weekly Sale Prices. *These 9 line plots show the change in weekly housing prices over a 6-year period. Each line represents a decile in weekly house prices. These are normalized to the respective decile value for sales in the week of Apr 27, 2003 (the beginning of the data collection period). The least expensive houses show the greatest variability, rising the most and dropping the most, relative to their starting position.*

3.2.8 Geographic Data

The housing data include geographic information. In addition to street address, we have the latitude and longitude of each house sold. With this information, we can examine spatial relationships. For example, in Figure 3.16, we have plotted the upper quartile of sale price across a latitude-longitude grid. The upper quartile is coded as a color ranging from blue (\$100,000) to red (\$10,000,000). These colored tiles use alpha transparency so that we can discern the city names beneath them. Information is revealed in this map that is not apparent from the scatter plots, smooth curves, box plots, and time series plots we have examined already. Furthermore, the background map shows land masses, bodies of water, cities, and roadways; these details provide additional context for interpreting the patterns in housing prices. In Figure 3.16, we see that sale prices on the San Francisco peninsula are higher than those in the east bay; houses along the Golden Gate are the most expensive in the city; and housing in Marin County north of San Francisco also command high-prices. The regions with very few colored tiles include national, state, and regional park areas. Additionally, the bands of color that run along both sides of the bay indicate that houses in the hills command relatively higher prices than those in the flats.

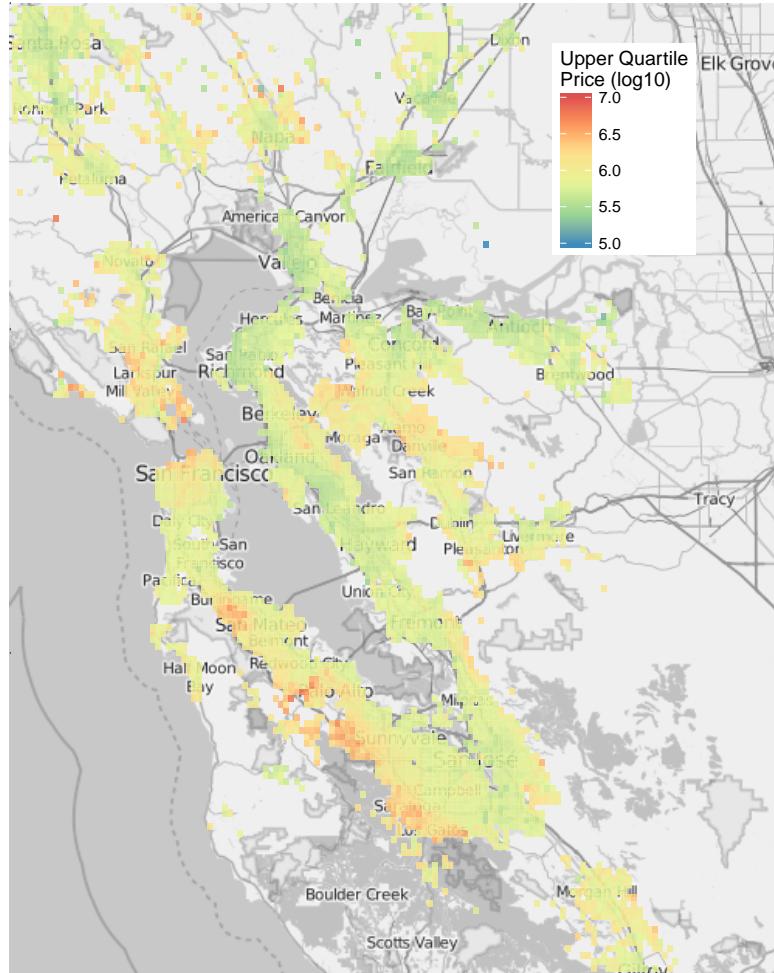


Figure 3.16: Housing Price Map. This map of the San Francisco Bay Area shows the locations of the houses sold in 2004. The points are color-coded according to the upper quartile in sale price at that geographic location. It is evident that prices on the peninsula (San Francisco and south) are higher than those in the east bay; houses in the Golden Gate area of San Francisco are the most expensive in the city; north of San Francisco is also a high-priced area; and houses in the east bay and on the peninsula with higher elevation command relatively higher prices. The city of Piedmont, which has the highest prices in the east bay appears as a small red area surrounded by Oakland.

3.3 Guidelines

In this section, we introduce guidelines for making effective statistical graphs through a collection of before-and-after plots. The ‘before’ plots are adapted from or inspired by plots we have encountered in the news and on the Web. For each, we describe several problems with the ‘before’ plot and how they are addressed in the ‘after’ version.

Word clouds such as the one shown in Figure 3.17 have gained tremendous popularity. The word cloud randomly arranges words found in text documents and scales them according to their frequency of occurrence. These visualizations are terrific for t-shirt designs, but they are not very effective at communicating the essential information in the data and in many cases can be misleading. The data underlying the pair of graphs in Figure 3.17 come from job listings on Kaggle. All postings for ‘data science’ or ‘data scientist’ were scraped from Kaggle’s job postings in January, 2015. The listed skills were extracted from these ads and tallied. The word cloud displays the top 20–25 terms in a random pattern, and the height of each word is proportional to the frequency of occurrence in the listings. The random arrangement makes it difficult to compare the frequency of skills, e.g., is statistics or python listed more often? Furthermore, short terms are not correctly represented. For example, based on the area that the letter R covers, it appears that *R* occurs in about 1 listing to 4 or 5 for *Python*. However, we see in the dot chart in Figure 3.17 that this is clearly not the case. In fact, *R* appears more often than *Python* in these listings. The problem is that in the word cloud the height of the letters in a term is proportional to the frequency so a term with the same frequency as another but with more letters covers a greater area and appears to have a greater frequency than the shorter term. Although the dot chart is not as eye-catching as the word cloud, it orders terms according to their frequency making them easy to compare accurately.

The line plot on the left of Figure 3.18 is a remake of a plot that was presented by Congressman Chaffetz (R-UT), chairman of the US House Oversight Committee in the 2015 hearings investigating federal funding of Planned Parenthood (<https://oversight.house.gov/interactivepage/plannedparenthood/>). This plot originally appeared in a report by Americans United for Life (<http://www.aul.org/>). It demonstrates a clear violation of good graphics principles. The lines are meant to show the change in the types of procedures carried out by Planned Parenthood from 2006 to 2013. At first impression it appears the number of cancer screenings plummeted while the number of abortions sky-rocketed in this 6-year period. However, close inspection reveals that the plot has no y-axis, and the lines for cancer screenings and abortions are drawn on different scales. That is, Planned Parenthood performed 327,000 abortions and 935,573 cancer screenings in 2013, yet cancer screenings appear below abortions in this plot. One way to fix this problem, is to use the same vertical scale for both lines. However, since the number of cancer screenings in 2006 is nearly 10 times the number of abortions (2,007,371 compared to 289,750), the increase in abortions from 289,750 to 327,000 appears roughly flat in this plot. Given that there are only 4 numbers in this plot (we do not have annual figures from 2006 to 2013), it may be more informative to report the proportion of the total number of procedures via a dot chart (on the right in Figure 3.18). We have grouped the dots according to the type of procedure to make it easier to compare the change from 2006 to 2013. There we see that cancer screenings dropped from 87% of procedures in 2006 to 74% in 2013, and this drop was mirrored by an increase in abortion procedures (13% to 26%). The actual changes are not nearly as dramatic as those depicted in the original plot.

The World Resources Institute (<http://www.wri.org/>) provides data on historical carbon dioxide (CO₂) emissions from fuel combustion (<http://cait.wri.org>). The

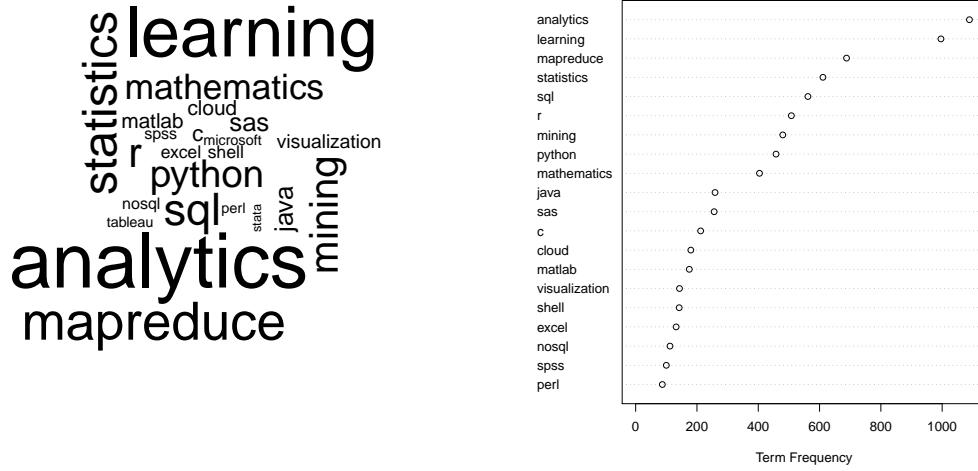


Figure 3.17: Kaggle Job Postings. The word cloud (left) is an attractive graphic but it is difficult to compare the frequencies of terms due to their random arrangement. In contrast, comparisons are easy and accurate with the dot chart (right). Moreover, the word cloud is visually misleading because the height, not the area, of each word is proportional to its frequency. These terms are from job postings for a data scientist on Kaggle in January 2015.

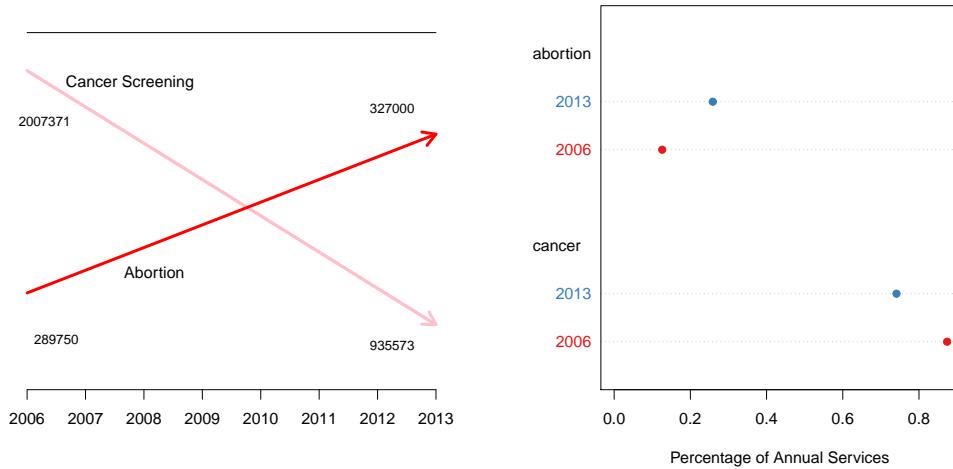


Figure 3.18: Planned Parenthood Services. The line plot (left) shows Planned Parenthood's decrease in cancer screenings (pink) and increase in abortion services (red) from 2006 to 2013. This plot gives a misleading comparison of the change because the two line segments are on different scales (note that the 327,000 abortions appears above 935,573 cancer screenings for 2013. On the other hand, the dot chart (right) shows the change in the proportions of services for these two years. There we see that the percentage of cancer screenings dropped from 87% to 74% and the abortions rose accordingly from 13% to 26%.

data that we have downloaded provide country annual CO₂ emissions dating back to 1850. We have plotted trends since 1950 for the 14 countries that emitted the greatest amount of CO₂ in 2012. The plot on the left of Figure 3.19 is an example of the colorful plot that we often see made with data like these. Since we have emissions over time, it is natural to make a line plot to see trends in emissions. However, stacking these line plots makes it difficult to compare country trends. Only the first country's data and the total for all 14 countries have a horizontal base that enables us to accurately perceive changes in emissions. For all of the other countries, the change from one year to the next is the length of a vertical segment where both the top and bottom of the segment move up and down from one year to the next, which makes it difficult to assess the trend and size of change. In contrast, the super-posed line plots (Figure 3.19, right) are not stacked, i.e., all values are rendered relative to the horizontal axis. We have also used a log scale. With this scale the data fill the plotting region, we can more easily compare countries with emissions that are different orders of magnitude, and we can see the kind of growth some countries have undergone in this period. We have also used only 7 colors in this plot, rather than 14, because most people have trouble distinguishing between more than about 7 colors. We use 2 different line types for each color to distinguish between countries, e.g. Brazil and Japan have the same color but Japan's line is dashed.

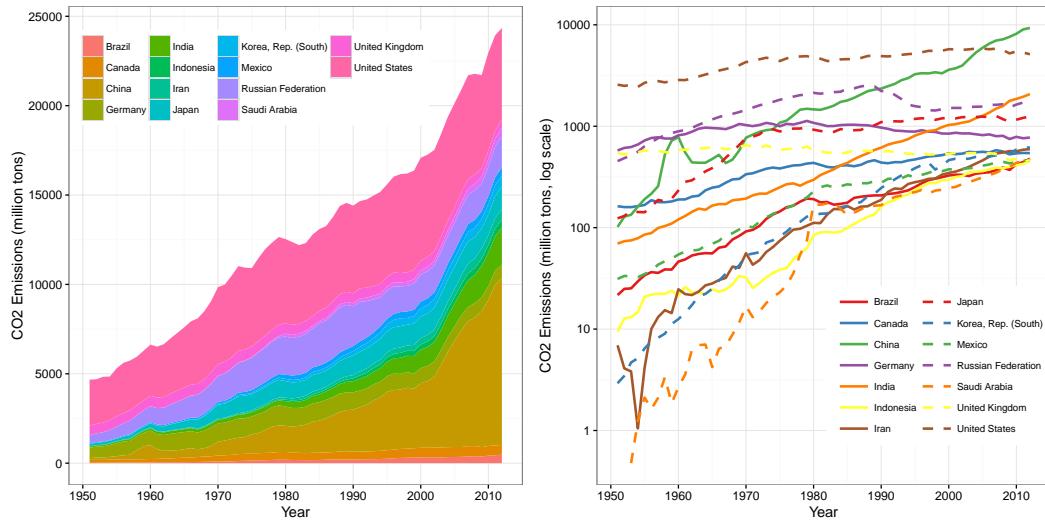


Figure 3.19: CO₂ Emission Trends. The stacked line plot (left) displays the rise in CO₂ emissions from 1950 to 2012 for the 14 countries with the highest emissions in 2012. It is difficult to compare the countries because the base line for each country jiggles up and down with the emissions of the country below it. Only the first country at the bottom of the plot has a straight base line, but the tremendous change in China's emissions over this 60-year period masks any changes for, e.g., Brazil and Canada. In contrast, the line plot (right) is not stacked so the countries can be directly compared. Also the emissions are displayed on a log scale making it easier to see how both the small and large countries are changing.

The graphs in Figure 3.20 examine the improvements in the manufacturing of the Intel chip for desktop computers since 1974, when Intel released its first microprocessor for a home computer (the 8080 chip). The plot on the left of this figure tracks the number of transistors on each new desktop model of microprocessor introduced from 1974 to 2004. (These data are from <http://computer.howstuffworks.com/microprocessor1.htm>.) Intel now

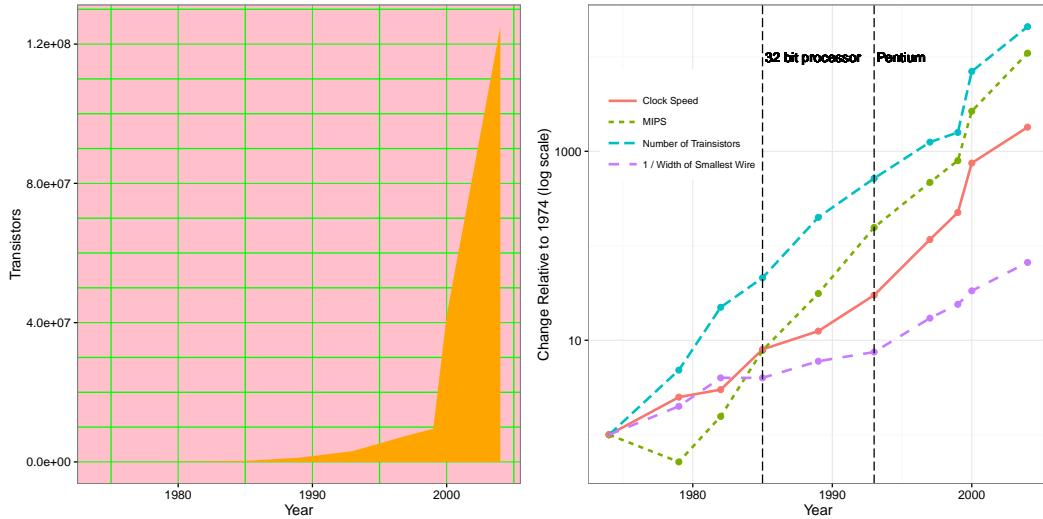


Figure 3.20: Intel Chip Benchmark Scores. *The line plot of the number of transistors on an Intel chip from 1972 to 2004 (left) demonstrates many flaws. The value for 2004 is orders of magnitude greater than the values from the 1970s through 1990s making it difficult to see how the relationship has changed over time. Furthermore, the use of color for the background, grid lines, and the region between the line and axis are too bright, distracting, and offer no insights. Alternatively, the plot on the left uses color to differentiate between the various measures of improvement, the y-axis is on a log-scale so that we can see how quickly these scores have grown, and two important design changes for the chip are denoted with reference lines.*

manufactures microprocessors with multiple cores and millions of transistors, but the new technology is not as easily compared to the earlier technology. For example, clock speeds in 2014 remain close to those of the Pentium 4, but the newer CPU can get more work done in one cycle so comparing clock speed is not informative. We made the visualization on the left in Figure 3.20 as an example of glaringly poor graphics design. It breaks many of the guidelines for good graphics:

- The data do not fill the plotting region because the number of transistors has grown so quickly that the most recent years take up most of the vertical range. This makes it very difficult to assess the kind of growth, e.g., we cannot tell whether the growth is exponential or slower/faster.
- The colors are very bright which makes it difficult to examine the plot closely. We typically want to be able to study a plot for minutes at a time so want to choose colors that facilitate careful inspection. Bright colors such as these have an after image effect that interferes with our inspection.
- Color should be used sparingly and should convey information. In this plot, orange fills the region between the data curve and the x-axis to no particular purpose. Also, the grid lines and background should use colors such as grey, white and black because they recede and do not interfere with the view of the data.
- The plot has obtrusive grid lines; they are too numerous, too thick, and too bright. We

want grid lines to assist us in reading off values from the data curve and not to dominate or interfere with our perception of the data.

- In addition to the number of transistors, our data include other aspects of the chip, such as the width of the smallest wire (in microns), the clock speed or the number of clock cycles a CPU can perform per second (in Mega Hertz or Giga Hertz), and the number of instructions the chip can execute per second (MIPS – millions of instructions per second). Ideally, we want our plot to be rich with information about these additional aspects of the chip.

The plot on the right of Figure 3.20 remakes the original to address these issues. The y-axis is on a log scale so that we can see the rate of improvement. All color has been eliminated, except for the color of the lines. These multiple curves show the changes in clock speed, size of the wire, and MIPS, in addition to the number of transistors. Since they are measured in different units, the values for each variable are scaled by its 1974 measurement. In addition, we have added 2 reference lines that mark major developments in the chip—the 32-bit processor and the Pentium processor.

These pairs of before-and-after plots have introduced many of the considerations that we take into account when designing a statistical graph that effectively conveys a story. We summarize the ideas presented in these examples into a set of topics and guidelines for good graphics.

3.3.1 Scale

When we choose the scale of an axis, we try to fill the plotting region with data. See for example, Figure 3.19, where the log transformation and limits on the y-axis create line plots that make it easy to see that some country emissions have remained flat and others have grown exponentially in this 60-year period. The log-transformation makes some lines appear at a 45-degree angle, which we call banking to 45 degrees. Banking makes it easier to ascertain a trend in the data. If we want to highlight some structure in the data that is not readily visible from the scale that we have chosen, then we can make a second plot for a subset of the data that uses a scale that brings out this additional feature. We can place this second plot in an inlay to the main plot or next to the original plot. Additionally, we may want to drop a few unusually large observations in order to get a better view of the main portion of the data. See for example Figure 3.14 where unusually expensive and large houses are not included in the scatter plot nor in the calculation of smooth curve. If we do not include all of the data in the graph, then we need to mention in the caption or on the plot itself that we have dropped some observations.

Depending on the measurements, it may not be necessary to include 0 on the axis, especially if including it makes it difficult to fill the data region. For example in the y-axis for the box plots in Figure 3.12, there is no need to include 0 as the lowest prices are near \$100,000. On the other hand, the bars in Figure 3.7 include 0 so the heights of the bars for 10% have the correct proportion when compared to the bars for 40%, i.e., they are 1/4 as tall.

3.3.2 Position

When we juxtapose plots that contain different subsets of the data, we want to use the same limits on the axes of the plots in order to facilitate comparisons across plots (see Figure 3.13). We want to arrange bars in a bar chart and dots in a dot chart in increasing order and side-by-side box plots in increasing order of the median (Figure 3.12). An exception to this

convention is when the groups are naturally ordered, such as with education level. If groups are defined by two qualitative variables, then we want to arrange the groups in a way that emphasizes the important difference; see for example the grouping of the bars for level of education within smoking status in Figure 3.9.

Comparisons are often easiest if we superpose plots. See for example dot plots in Figure 3.9 (bottom right), density curves in Figure 3.10 (top right), line plots in Figure 3.19 (right), and smooth curves in Figure 3.14 (right). If superposition is not feasible then side-by-side comparisons are preferable (Figure 3.9, top right) to stacking bars or curves because stacking makes it difficult to compare subgroups.

If many observations have the same values, we can add a small amount of random noise to these values in order to reduce the amount of over plotting and better see the underlying relationships (Figure 3.8).

3.3.3 Shape

Length is easier to compare and assess than angle, and for this reason barplots and dot charts are preferable to pie charts (Figure 3.7). However, stacked bars and line plots hinder comparisons as they lack a constant base line. Each dimension of a multidimensional plot (e.g., the depth in a 3-dimensional bar) should represent a variable, otherwise it is unnecessary. Similarly, maps that fill geographic regions can be misleading if the density of observations is not constant over these regions.

3.3.4 Aggregates

With large amounts of data, we often want to visualize aggregates of the data. Smooth curves or lines can be plotted in addition to or instead of scatter plots to make average relationships more apparent (Figure 3.14, right). Alpha transparency shows darker colors when symbols are over plotted, which can help reveal high density regions (Figure 3.13). Color can represent density in a smooth scatter plot (Figure 3.14, left).

When the data have been collected according to some sampling design and the observations represent different numbers of individuals, we need to incorporate the sampling design in any plots that we make (Figure 2.24).

3.3.5 Color

If we use color in a plot, the color should represent information and not be gratuitous. Depending on the type of information being represented, different color palettes are preferable. For categorical data, we want to use a collection of colors that are easily distinguishable and where one does not stand out more than another. For continuous data, we want to use a sequential gradation that emphasize one end of the spectrum of values over the other, or we want a diverging palette that emphasizes the two extremes of the spectrum over the middle. The choice between a sequential and diverging palette depends on the message being conveyed. For example, with cancer rates, we want to use a sequential palette that increases the brightness and saturation for high rates. On the other hand, with two-party election results we want a diverging palette with two distinct hues for low and high values. Both ends of the palette use bright saturated colors to distinguish between one party's dominance over the other.

Colors can be specified in several ways in *R*, including by name, such as `cornflowerblue` and `lightgreen`. We can also use a triple of numbers for the amount of red, green and blue light to add together. These numbers are typically specified in hexadec-

imal and they range from 0 and 256, i.e. between 00 and FF. For example, the color called `cornflowerblue` is a combination of 100 red, 149 green and 237 blue, or in hexadecimal we express it as "#6495ED". We discuss the various ways to specify colors in greater detail in Chapter 10. Creating a sequential, diverging, or qualitative palette of colors that appropriately conveys the underlying values in a variable is a difficult task, and we recommend using those that have been developed by researchers, such as Cindy Brewer's palettes (see the functions in the `RColorBrewer` package [8] for examples).

Plots are meant to be examined for long periods of time so we want to use colors that we can stare at without impeding our ability to perceive the important information in the plot. For example, we want to avoid colors that create an after-image when we look from one part of the graph to another. Similarly, we should avoid using combinations of colors that color-blind people have trouble distinguishing between. Furthermore, people have trouble distinguishing between more than about 7 to 9 colors so we should limit the number of colors we use in a plot. Finally, colors can appear different when, e.g., printed on paper or projected on a screen, so they should be chosen with the medium in which they will be presented in mind.

3.3.6 Context

Depending on whether our plots are part of an informal exploratory analysis or a formal presentation, we include different amounts of contextual information. However, even with EDA we want to include some context so that when we return to an analysis we can easily determine what we have plotted. It's good practice to consistently use informative labels on axes (including the units of measurement), labels on tick marks, and titles. Plot captions should describe what has been plotted and point out the important features of the plot.

Ideally, we include additional context to help tell the data's story. For example, reference markers and lines provide benchmarks to compare against and other external information that's helpful in interpreting the results. Examples include the background map of the plot of sale locations in Figure 3.16 and the reference lines in the plot of Intel chip historical development in Figure 3.20. We also use color and plotting symbols to include additional variables in a plot, e.g., the points in the scatterplots in Figure 3.13 are colored according to the number of bedrooms in the house.

3.3.7 Over Arching Considerations

The previous guidelines are organized according to various features of a plot. When we examine a plot that we have made, in addition to considering this checklist, we can ask ourselves more holistic questions about the visualization. The following three questions are abstracted from Cleveland's *Elements of Graphing Data* and they help serve as a framework for developing a statistical graph. We connect these questions with the above topics and guidelines.

Do the Data Stand Out?

Essentially, this is a question of scale, transformations, and banking. That is, we want the data to fill the plotting region in order to best reveal its structure. By structure we mean the shape of a distribution or the relationship between variables. Other considerations in answering this question address whether the data are obscured because, e.g., plotting symbols are too small, there's too much over plotting, graphing elements cover the data, colors are too bright, and extraneous glyphs detract from the visual perception. We want to avoid hiding the data in our visualization.

Does the Plot Facilitate Comparisons?

When we create a plot, we need to keep in mind what is the important comparison and we want to be sure our plot has emphasized this comparison well. A comparison can be a simple benchmark or reference or a partition of the data into subgroups that we want to compare. We consider whether or not we can further reveal a relationship by providing more information or context by, e.g., using color to denote another variable. Additionally, we consider which kind of plot makes it easy for the reader to make accurate comparisons and whether we want to superpose graphing elements on the same plotting region, e.g., density curves, or juxtapose plots, e.g., a grid of scatter plots for subgroups. We have seen that aligning dots along the same axis is one of the easiest and most accurate approaches to compare proportions, much better than stacking bars or dividing pies into slices. The proper choice of a color palette can also facilitate comparisons. Lastly, identifying individual points can assist in making comparisons of one or a few cases against the rest.

Can We Add Information?

We want our visualizations to be information rich without distracting from or cluttering the main message. We nearly always want our axes labeled (an exception is with a map). Our plots should contain titles, and when needed, legends. We can add information with color and plotting symbols, reference markers, and point labels. This additional information provides a context for interpreting the findings. Importantly, we want to clearly describe these findings and what we have plotted in a comprehensive caption.

3.4 Iterative process

Making a good statistical graph is an iterative process of discovery. After we make a plot, we examine it for ideas on how to improve that plot and we consider whether we should make an entirely different plot. We consider the 3 questions in Section 3.3.7 as we continue to try to uncover the story in the data and find a way to present it clearly and effectively. For example, in making the plot about Intel chips, we went through several stages of development. Our first visualization (top left in Figure 3.21) was similar to the plot on the left in Figure 3.20, without all of the garish bells and whistles. When we examined this plot, we realized that we needed to transform the number of transistors in order to fill the data region and get a sense of the rate of growth. This led us to the second plot (top right) in Figure 3.21. We use base 10 logarithm rather than the natural logarithm to make it easier for the viewer to read the original units, e.g., 1,000 vs 10,000 transistors. This plot provides a reasonable visualization, but we asked ourselves whether we can include more information. The problem with including other measurements of the chip is that these variables have incompatible units. Since it's the pace of the change that interests us, we can convert the data into the same units by scaling each to their respective value in 1974. To create the next plot, which appears in the bottom left of Figure 3.21, we needed to perform several data manipulations (see the exercises). At this point, we think that the plot is basically complete, except for the addition of context to help in understanding the message. The final version (bottom right) has informative axis labels and legends, and we have added two reference lines to indicate major advances in chip technology. Note that we would normally add a title to our plot, but we use the figure title for the plot title. The caption describes what we have plotted and points out important features in the plot.

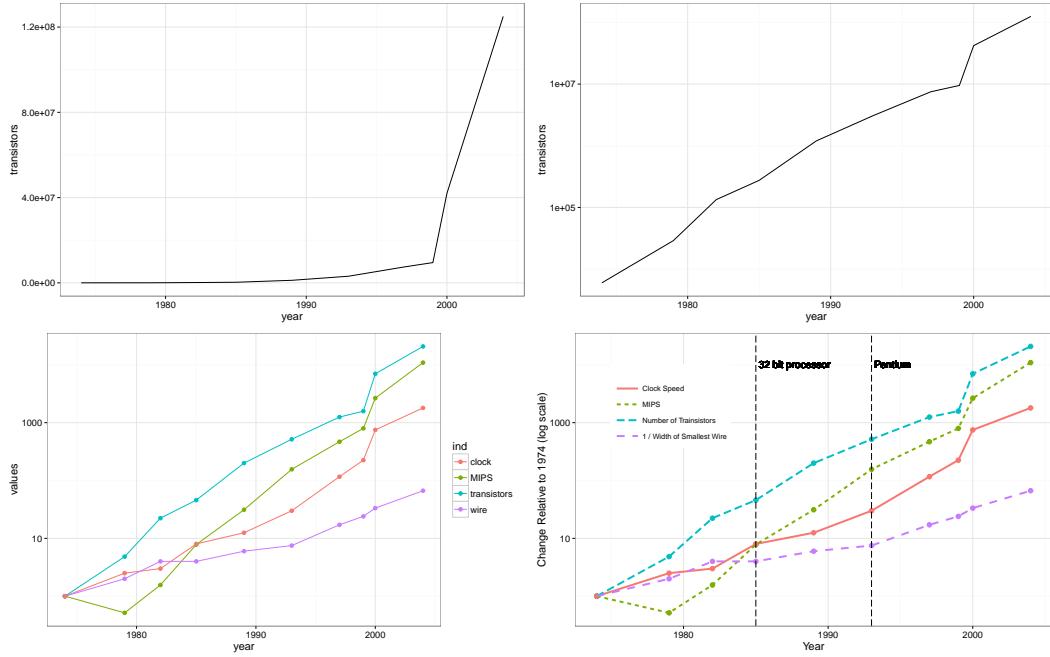


Figure 3.21: Intel Chip Benchmark Scores – An Iterative Visualization Process. These 4 graphs represent the 4 steps in the process of creating the visualization of the development of the Intel chip. They are (from top left to bottom right) the initial plot of the number of transistors, transforming the number of transistors to log-scale, transforming the measurements of all the variables so they are relative to the 1974 values, and adding context with more informative labels, legends, and references markers.

3.5 Rs Graphics Models

There are two models for creating statistical graphs in *R*: the painter's model, which the functions in base *R* provide, and the object-based model, which follows Grammar of Graphics [14] and has been implemented in the `ggplot2` package [?] using grid graphics approach of the `grid` package [7]. We have used both techniques to create the plots in this chapter and other chapters. However, since these two models take quite different approaches, we encourage the beginner to focus on one method at first. We briefly describe them both in this section.

3.5.1 Painter's Model in Base R

The graphics functionality in base *R* creates a statistical graph from a high-level function, such as `plot()`. There are a host of functions for making different kinds of plots, including `hist()`, `plot()`, `boxplot()`, `dotchart()`, `barchart()`, and `mosaicplot()` to make a histogram, scatter plot, box plot, dot chart, bar chart, and mosaic plot, respectively. A call to one of these functions initiates a new plot. We can think of this as a new plot on a new 'page', like a painter starting a new painting on a new canvas. This function call creates a complete plot.

In addition to these high-level plotting functions, there are several low-level functions

that can add more to the current plot. These functions include those to add a line, additional connected line segments, additional points, shapes, text, and legends, the corresponding functions are `abline()`, `lines()`, `points()`, `polygon()`, `text()`, and `legend()`, respectively.

The high-level plot functions have many common arguments to adjust the appearance of the plot. In most plots, we can modify the default labels for the axes (`xlab` and `ylab`), range of the axes (`xlim` and `ylim`), title of the plot (`main`), color of points and lines (`col`), plotting symbol and size (`pch` and `cex`), and type and thickness of line, (`lty` and `lwd`). There are many other parameters, some of which make sense for a particular type of plot, e.g., `vertical` to indicate whether the bars in a bar chart are to be vertical or horizontal; `freq` to indicate whether the area of the bars in a histogram should be counts or proportions; `breaks` to specify the number or location of the intervals in a histogram; `groups` to indicate whether the dots in a dot chart are to be grouped by a categorical variable; and `labels` to change the default labels for the dots.

Finally, the `par()` function can be used to globally control many plotting parameters. Some of the arguments to `par()` are exclusive to this function. Two that are quite useful are `mar` to control the size of the plot margins and `mfrow` to divide the canvas into subpanels for multiple plots. We highly recommend reading the documentation for `par()` to get a sense of the tremendous flexibility available for making plots with base R functionality.

We close this section with 2 examples of code that we used to create 2 of the plots in this chapter. Our first example creates the scatter plot in Figure 3.11. We begin by making a vector of muted green and purple to use for the color of the points and lines in the plot. We specify these with their RGB (red-green-blue hexadecimal values) as

```
smokeColors = c(Never = "#1b9e77", Current = "#7570b3")
```

Notice that we assigned names to these elements to match the levels of the `smoke` variable in `babies`.

We call `plot()` with

```
with(babies[babies$smoke == "Never" | babies$smoke == "Current", ],
     plot(x = jitter(ht, amount = 0.5),
           y = jitter(bwt, amount = 0.5),
           xlim = c(58, 71),
           pch = 19, cex = 0.4,
           col = smokeColors[as.character(smoke)],
           xlab = "Mother's Height (in)",
           ylab = "Birth Weight (oz)"))
```

We use `with()` to make the scatter plot because it makes it easier to specify the subset of current and never smokers and avoid using \$-notation when we specify the variables in `plot()`. Notice that we used `jitter()` to add a small amount of random noise to both height and birth weight to avoid overplotting. We also shrunk the plotting symbol to limit the amount of over plotting. We have specified the limits of the x-axis to zoom in on the main portion of the data; a few unusually small/tall mothers are not included in the plot. The argument `col` is set to the expression `smokeColors[as.character(smoke)]`. This expression uses indexing by name to create a vector of reds and blues for the current and never smokers.

Now that we have created the basic scatter plot with the high-level function `plot()`, we use `loess()` to fit two smooth curves to the data, for the current and never smokers. Later we add these 2 curves to this scatter plot using the lower-level function `lines()`. We fit the two curves with

```
bwtN.lo = loess(bwt ~ ht,
```

```
data = babies[babies$smoke == "Never", ], span = 0.5)
bwtS.lo = loess(bwt ~ ht,
                 data = babies[babies$smoke == "Current", ], span = 0.5)
```

The first argument to `loess()` is a formula that indicates we want to model birth weight as a function of mother's height. The fit is a smooth curve that essentially averages birth weight for mothers with similar heights.

Next, we use the information from the fits in `bwtN.lo` and `bwtS.lo` to make predictions of birth weight for a dense grid of heights from 59 to 69 inches. This way we can draw a 'curve' with line segments that connect the sequence of predictions. We create the set of heights with

```
gridHt = data.frame(ht = seq(59, 69, 0.2))
```

Then, we call `predict()` and pass it the fitted loess object (`bwtN.lo` or `bwtS.lo`) and the vector of heights in `gridHt` to find the predicted birth weights with

```
pred.bwtN = predict(bwtN.lo, gridHt, se = FALSE)
pred.bwtS = predict(bwtS.lo, gridHt, se = FALSE)
```

Lastly, we add these pairs (height, predicted birth weight) to the scatter plot and connect the dots with `lines()`. We do this with

```
lines(x = gridHt$ht, y = pred.bwtN,
      col = smokeColors["Never"], lwd = 3)
lines(x = gridHt$ht, y = pred.bwts,
      col = smokeColors["Current"], lwd = 3, lty = 3)
```

Notice that we matched the color of the curves to the associated group, used different line types to further help distinguish between the curves, and made the line thicker so that it stands out from the point cloud.

For a second example, we demonstrate how to make the bar chart in Figure 3.2. This bar chart is made from the following matrix of counts of counties,

	[,1]	[,2]	[,3]	[,4]	[,5]	[,6]	[,7]
Majority.of.Democrats	43	37	29	21	23	28	28
Majority.of.Republicans	15	21	29	37	35	30	30

Again, we begin by creating a vector of colors with

```
require(RColorBrewer)
partyColor = brewer.pal(3, "Set1") [1:2]
```

A call to `barplot()` make the basic bar chart for us with

```
barplot(countyCounts,
        beside = TRUE,
        col = partyColor,
        axes = FALSE,
        xlab = "Presidential Election Year",
        ylab = "Number of Counties",
        ylim = c(0, 47),
        main = "Majority Party in California Counties")
```

Note that we have asked that the bars corresponding to the 2 rows in the matrix (the number of Democratic and Republican majority counties) to be plotted side by side and that their colors match the traditional party colors, which we provided in `col`. We also provide labels for both axes and a title. This plot is unusual in that we have turned off the automatic drawing of the axes with `axes = FALSE`. We do this because we want to control the location and labels for the tick marks on the x-axis.

We complete the plot with several lower-level function calls to add axes, a box around the plot, text above each bar, and a legend. The axes are numbered 1 through 4 beginning at the bottom and moving clockwise around the perimeter of the plot. We begin by adding the y-axis on the left side of the plot with

```
axis(2)
```

This call, `axis(2)`, uses the default value for the location and labels of the tick marks. Then, we add the x-axis along the bottom of the plot with

```
axis(1, at = seq(2, 22, by = 3), labels = seq(1992, 2016, 4))
```

Here we place ticks between each pair of bars and label them with the election year. Note that the actual scale on the x-axis runs from 1 to 23, not from 1992 to 2016; the years are simply labels on the tick marks.

We use `text()` to add the county counts above their respective bars. To do this, we provide the (x, y) coordinates for the top of each bar and specify the `pos` argument as 3 so the text is placed above this point and doesn't interfere with the bars. We call `text()` with

```
text(x = rep(c(1.5, 2.5), 7) + rep(seq(0, 18, by = 3), each = 2),
     y = countyCounts,
     label = countyCounts, pos = 3, cex = 0.8)
```

Notice that we also shrink the text to 80% of its normal size so that it doesn't detract from the bars, and again the x-axis runs from 1 to 23. To make the axes more visually pleasing, we add an L-shaped box with

```
box(bty = "L")
```

Lastly, we add a legend to the plot with

```
legend(x = 18, y = 47, legend = c("Dem", "Rep"),
       fill = partyColor, cex = 0.8, bty = "n", title = "Party")
```

Notice that when we specified `ylim` in the original call to `barplot()`, we created enough vertical space in the plot for the numbers above the tallest bin and the legend.

3.5.2 Grammar of Graphics Model in `ggplot2`

The implementation of the grammar of graphics in `ggplot2` takes a different approach to constructing statistical graphs. We begin by defining an empty plot object with `ggplot()`. Then we add layers to the plot by specifying the graphics shapes, called geoms, with which to view the data, e.g., plotting symbols and lines. These are added to the plot object with the `+` operator. Each layer can have its own data (which must be in a data frame) and aesthetic mapping. This mapping connects variables in the data frame to a feature, such as x and y locations, color, and size. As an example the aesthetic `aes(x = Yr, y = CO2, color = Ctry)` maps year to the x-axis, CO2 emissions to the y-axis, and uses color to

denote country. This aesthetic mapping can be used to add points or lines to a plot. The geometric shape (e.g., point or line) is also specified in the layer.

We can modify the scales for these aesthetics to, e.g. use a log-scale for the y-axis, provide a special label on the x-axis, and specify a palette for the colors. We do this by adding `scale()` functions to the plot. That is, `scale_y_continuous()` has parameters `name` to specify the axis label, `breaks` to provide the location of the tick marks, `trans` to use a transformation such as log, `limits` to denote the range of the scale, etc. Scale functions are also available for discrete-valued axes and for color, shape, line type, size, etc.

Additionally, details related to, e.g., the appearance of axes, size of text, and background color for the plotting region are specified through themes.

When we create a plot, we call `ggplot()` and then add layers of data and, if desired, scale and theme specifications. We provide 2 examples by reviewing the code used to create the line plots in Figure 3.19 and the grid of scatter plots in Figure 3.13.

We begin by creating the plot on the right side of Figure 3.19. The first step is to create a plot object with `ggplot()`. We can specify the data and the aesthetic mapping in this function call, or we can do this in the layers, or both. When these are specified in `ggplot()`, they are available to all layers and they can be overridden in a specific layer. Since we have only 1 layer in this plot (the lines), it makes no difference where we provide the data and mapping so we specify them in the call to `ggplot()`. We provide this information in `ggplot()` with

```
co2Plot = ggplot(data = co2Top14,
                  mapping = aes(x = Yr, y = CO2, color = Ctry,
                                linetype = Ctry))
```

We have created a plot object, not a plot. After we specify the geometric shapes, we can print/see the plot. Notice that we map `Yr` to the x-axis, `CO2` to the y-axis, and `Ctry` is mapped to both color and line type. When we add a layer for lines to this plot, we see how this mapping is rendered. We add this layer with

```
co2Plot + layer(geom = "line", stat = "identity",
                 position = "identity",
                 params = list(size = 1))
```

Each layer has a geometric shape and a statistic. Depending on these, we can pass additional parameters to control the geom. In this example, we simply want to make the lines thicker so we set `params` to a list with a named `size` element. The `position` argument specifies how to handle over plotting. The value '`identity`' indicates that we are not jittering or performing any sort of repositioning to address over plotting.

Several short cut functions are available for layers. That is, each type of geom has a default statistic and position and the layer can be called with `geom_XXX()`, where XXX is the name of the `geom` and the default values for `stat` and `position` for this geom are provided. The addition of the above layer to `co2Plot` is equivalent to

```
co2Plot + geom_line(size = 1)
```

This one layer produces a plot very similar to the plot in Figure 3.19, except for the scale of the y-axis, the axis labels, locations of the tick marks for both axes, choice of color, line type for the lines, and the appearance and location of the legend. We modify the scales for the x and y axes and for the color and line type with scale components that we add to our plot. We change the x and y axes by adding the following calls to `co2Plot` with

```
co2Plot + geom_line(size = 1) +
  scale_x_continuous(name = "Year",
                     breaks = seq(1950, 2010, 10)) +
  scale_y_continuous(
    name = "CO2 Emissions (million tons, log scale)",
    breaks = c(1, 10, 100, 1000, 10000), trans = "log10") +
```

Note the trailing `+` indicates that we plan to add more component specifications to the visualization.

We want to use solid and dashed lines for each of 7 colors from a Brewer palette so that we can differentiate between the 14 countries. To do this, we add calls to `scale_linetype_manual()` and `scale_color_manual()` to the above plot object with

```
scale_linetype_manual(
  name = "",
  values = rep(c("solid", "dashed"), each = 7),
  guide = guide_legend(nrow = 7)) +
scale_color_manual(
  name = "",
  values = rep(brewer.pal(7, "Set1"), 2)) +
```

By setting the `name` of the line type and color scales to the same value, the legends for these two scales are combined into 1.

Lastly, we use the black and white theme for the background colors and grid lines and we modify the location and appearance of the legend with

```
theme_bw() +
theme(legend.position = c(0.75, 0.25),
      legend.text = element_text(size = 8),
      legend.key.width = unit(2, "line"),
      legend.key = element_blank())
```

In addition to placing the legend inside the plotting region, we shrank the labels in the legend, widened the key so that the line types are clear, and eliminated the box around the key.

For our second example, we create the plot in Figure 3.13. We need to create the variable price per square foot and to collapse the number of bedrooms to 1 through 5, where 5 represents 5 or more. We add these new variables to our original data frame because ggplot2 expects the variables to be collected in one data frame. We do this with

```
housing04S$ppsf = housing04S$price/housing04S$bsqft
housing04S$br5 = housing04S$br
housing04S$br5[housing04S$br5 > 5] = 5
```

Note that `housing04S` contains the sales for 2004 for 12 cities in the East Bay, as described in Section 3.2.6.

We also want to add a smooth curve to each of the panels in the figure. This curve is created by averaging over all cities. We can fit the curve to our data and then create a data frame with equi-spaced values for building square footage and the respective prediction of price per square foot. We do this with

```
housingSmooth = loess(ppsf ~ bsqft, data = housing04)
housingPreds = data.frame(bsqft = seq(320, 5000, by = 100),
                           housingPreds$pred = predict(housingSmooth, newdata = housingPreds)
```

The data frame `housingPreds` contains two variables `bsqft` and `pred`.

We have completed our data preparation and can begin to make our plot. We ‘add’ together the following function calls to build the plot, beginning with creating the plot object with

```
ggplot(data = housing04,
       mapping = aes(x = bsqft, y = ppsf)) +
```

Notice that we have specified the aesthetic for mapping the x and y axes but not the color mapping because the color aesthetic is needed only in the point layer. We add the point layer with

```
geom_point(aes(col = factor(br5)), size = 0.5, alpha = 0.5) +
```

In addition to specifying the mapping of the number of bedrooms to the color aesthetic, we also set the size of the points to be 1/2 their regular size and we set the transparency level to 1/2. These settings help with over plotting because smaller points do not overlap as much and, when they do, the transparency allows us to see the point density and colors.

To create the panels of scatter plots, one for each city, we use `facet()` as follows:

```
facet_wrap(~ city0) +
```

The argument to `facet()` is a formula that says that we want to see the dependency on cities. The variable `city0` is an ordered factor that we have created so the cities are ordered according to median price.

The curve in each panel is a second layer. This layer is created from line segments that connect price per square foot predictions (`pred`) for the grid of building square footage values (`bbsf`) in our `housingPreds` data frame. We add this line layer with

```
geom_line(data = housingPreds, mapping = aes(y = pred)) +
```

Notice that we need to specify the data frame in this layer because it is different from the default data frame provided in `ggplot()`. We also map the y-axis to `pred` because it too differs from the y-axis aesthetic supplied in `ggplot()`.

Now that we have added our 2 layers, we control the scales for the 2 axes and the color with `scale_xxx_xxx()` functions. We provide the title and the limits for the x and y scales with.

```
scale_y_continuous("Price per Square Foot",
                   limits = c(0, 1500)) +
scale_x_continuous("Building Square Foot",
                   limits = c(0, 6000)) +
```

In addition, we specify the colors to use and map them to labels in the color scale with

```
scale_color_manual("Bedrooms",
                   labels = c(1:4, "5+"),
                   values = brColors,
                   guide = guide_legend(
                     override.aes = list(size= 2, alpha = 1))) +
theme_bw()
```

Notice that we also override the point size and alpha transparency that was set in `geom_point()` for legend readability. Our last addition specifies that we want to use the black and white theme for the background, axes, etc.

3.6 Creating Unique Plots

After some practice making standard plots in base *R* or *ggplot2* and some experience in following the graphics guidelines, you will be ready to design your own unique plots. We saw an example of this in Figure 2.15. A more typical visual representation of these data would be a graph that arranges circles (nodes) for the individuals and represents the emails by lines or arrows from the sender to the recipients. However, with so many individuals and emails involved, it is difficult to arrange this network on a page so that it is readable. The BioFabric plot was created to address this problem.

A second example appears in Figure 4.7. This plot is unique in that it aims to create a visualization for missing data. We can think of it as 56 rug plots, one for each weather station. The yarns are placed at the days when we have a precipitation measurement at the station, including the days when 0 inches of rainfall was recorded. The vertical stripes in the plot reveal that there are time periods when we have no data recorded for all 56 weather stations (these are the winter months). Additionally, the horizontal bands of white indicate that some stations have not been in operation as long as others and that a few stations were out of operation for lengthy periods of time.

According to Wainer [11], we don't want to re-invent a visualization if it has been done well. Moreover, if we can use a standard plot, such as those that appear in this chapter, then it's typically a good idea to use that rather than invent our own because viewers are familiar with how to read and understand the existing format. However, there are times when we want to adapt one of these standard visualizations (as in Figure 4.7) or create an entirely new kind of visualization (as in Figure 2.15) to overcome a limitation or present information from a new perspective. And we may even develop a function to create a specialized visualization. For example, in Section 7.7, we write a function to visually examine the results of a simulation study (see Figure 7.16).

3.7 Summary

There are many resources on how to design informative graphics. Some of our favorites are [3, 4, 5, 10, 11]. There are also many resources on how to create visualizations in *R*. For *lattice* plots see [9]. The grammar of graphics by Wilkinson [14] has been implemented in *R* by Wickham in the *ggplot2* package; see [12] for more details. For an in-depth treatment of all graphics models in *R* see [6].

3.8 Exercises

Bibliography

- [1] Code to download and process SF housing sales data. <https://github.com/hadley/sfhousing>, 2009.

- [2] Bay Area Home Sales - Weekly Updates. <http://www.sfgate.com/webdb/homesales/>, 2016.
- [3] William S. Cleveland. *The Elements of Graphing Data*. Wadsworth Advanced Books and Software, Monterey, CA, 1985.
- [4] Andrew Gelman. Why tables are really much better than graphs. *Journal of Computational and Graphical Statistics*, 20:3–7, 2011.
- [5] Andrew Gelman and Antony Unwin. Infovis and statistical graphics: different goals, different looks. *Journal of Computational and Graphical Statistics*, 22:2–28, 2013.
- [6] Paul Murrell. *R Graphics*. Chapman & Hall/CRC, Boca Raton, FL, 2005.
- [7] Paul Murrell. grid: The grid graphics package. <http://cran.r-project.org/package=grid>, 2011. R package version 2.16.0.
- [8] Erich Neuwirth. RColorBrewer: ColorBrewer palettes. <http://cran.r-project.org/web/packages/RColorBrewer>, 2011. R package version 1.0-5.
- [9] Deepayan Sarkar. *Lattice: Multivariate Data Visualization with R*. Springer-Verlag, New York, 2008. <http://lmdvr.r-forge.r-project.org/figures/figures.html>.
- [10] Edward Tufte. *The Visual Display of Quantitative Information*. Graphics Press, Connecticut, 1983.
- [11] Howard Wainer. How to display data badly. *The American Statistician*, 38:137–147, 1984.
- [12] Hadley Wickham. *ggplot2: Elegant graphics for data analysis*. Springer, New York, 2009.
- [13] Hadley Wickham, Deborah Swayne, and David Poole. Bay Area blues: The Effect of the housing crisis. In Toby Segaran and Jeff Hammerbacher, editors, *Beautiful Data: The Stories Behind Elegant Data Solutions*, pages 303–322. O'Reilly Media, Inc., Sebastopol, CA, 2009.
- [14] Leland Wilkinson. *The Grammar of Graphics*. Springer Science & Business Media, 2006.

4

Reading and Exploring Complex Data

CONTENTS

4.1	Introduction	149
4.2	Lists	154
4.2.1	Subsetting Lists	157
4.2.1.1	Accessing An Element of a List	158
4.2.2	Applying Functions to Elements of a List	160
4.2.3	Exploring Rainfall on the Colorado Front Range	161
4.3	Reading Data into a Character Vector	166
4.3.1	A Study of Web Page Updates	166
4.4	Reading Data from a Web Page	171
4.4.1	World Records in the Men's 1500 meter	174
4.5	Reading <i>JSON</i> Formatted Data	175
4.6	Kiva	181
4.6.1	Loan Elements	182
4.6.2	The Payments	186
4.6.3	The Borrowers	186
4.6.4	Final Structure	186
4.7	Summary	187
4.8	Functions for Handling Complex Data Formats	187
4.9	Guided Practice	188
4.10	Exercises	189
	Bibliography	189

4.1 Introduction

Data do not always come in simple rectangular collections of numbers like in Chapter 2. At times the data have a nested structure where data values are at different levels, e.g., loans and payments on these loans, information about weather stations and daily rainfall recordings at the stations, and automobile accidents and the vehicles involved. When we have data at different levels, we typically have a few options for organization. The choice of how to organize the data can depend on how we plan to work with and analyze them. We may even decide to organize the data in two different formats because one format is more conducive to a particular kind of analysis than another. We saw this in Section 2.3.3 with the Clinton emails. The first table we examined had 1 row for each email. Then later in Section 2.6.1, we worked with 2 additional files. One table had 1 record for each person who sent or received one of these emails. The other table had 1 record for each recipient of an email so some emails corresponded to 1 record in this file, i.e., when the email had 1 recipient, and others corresponded to 2 or more records when the email was sent to multiple individuals.

In this chapter, we encounter data from several applications. What they all have in common is that the data do not immediately translate into a data frame or table. In these examples, we demonstrate how to create multiple tables, a collection of numeric vectors of different lengths, and a collection of vectors and data frames.

Reading Complex Data

When data can not be arranged in a simple rectangular form, we must decide how to organize them, i.e., what structure to use to contain the data. This organization depends on several considerations, i.e.,

- What are our plans for analyzing the data?
- What is a convenient structure for this analysis?

Often in these situations, the data are at different levels of granularity. For each level, we need to identify the observations and variables, i.e., we consider the following:

- What entity corresponds to a row? (the data may contain different entities so we can have more than one answer to this question)
- What are the variables?

The structure of the data may be a collection of varying-length vectors, multiple related data frames, or a combination of these. When this occurs, we need to know:

- What is the variable (or variables) that connects the observations across tables?

Example 4-1 Rainfall in the Colorado Front Range

Climate scientists and statisticians study historical patterns in rainfall in an effort to make predictions of extreme weather events. For example, statisticians at the National Center for Atmospheric Research (NCAR) in Boulder, Colorado have worked on the problem of estimating rainfall for a future storm that has a 1% chance of occurring in a year; this is called the 100-year event. To help in this effort, they study recordings of daily precipitation at dozens of weather stations in the Front Range of Colorado. Their work involves collaborations with researchers at other locations, and they share their data in a format that is ready for analysis in *R*. One example is the *FrontRangeWeather.rda* file. An *.rda* file is a binary file so it cannot be viewed in a plain text editor, but we can easily load it into our *R* work space and begin analysis. We load the contents of *FrontRangeWeather.rda* with

```
load("FrontRangeWeather.rda")
```

The functions introduced in Chapter 1 can help us examine the objects in the file. First, we list them with

```
ls()
```

```
[1] "FrontRangeWeather"
```

We see the *rda* file contains one object, *FrontRangeWeather*. We determine its class with

```
class(FrontRangeWeather)
```

```
[1] "list"
```

`FrontRangeWeather` is a list. This is a different kind of data structure from a data frame. We call `length()` and `names()` to learn more about `FrontRangeWeather`:

```
length(FrontRangeWeather)
[1] 3

names(FrontRangeWeather)
[1] "days"      "precip"    "stations"
```

We find that `FrontRangeWeather` has 3 elements named `days`, `precip`, and `stations`. Lists are more complex and richer in structure than the vectors, data frames, and arrays we worked with in Chapter 2. In order to examine these data, we need to understand more about the list structure in *R*. This is the topic of Section 4.2. ■

Example 4-2 World Record in the Men's 1500 meter

Wikipedia contains a vast amount of information, some of which is presented in table format. One example is the world records in the men's 1500 meter race (see Figure 4.1 and [4]). This table has 5 columns and a row for each time the world record was broken. Notice that this is not a typical data table because, for example, the third column contains three pieces of information: an image of the athlete's home country flag, the athlete's name, and a 3-letter country code in parentheses. Moreover, the table is embedded within a Web page (see Figure 4.2) and there are several other tables on this page. Clearly we need to carry out some special processing to access the records in this table.

Time	Auto	Athlete	Date	Place
3:55.8		Abel Kiviat (USA)	1912-06-08	Cambridge, Massachusetts, USA
3:54.7		John Zander (SWE)	1917-08-05	Stockholm, Sweden
3:52.6		Paavo Nurmi (FIN)	1924-06-19	Helsinki, Finland
3:51.0		Otto Peltzer (GER)	1926-09-11	Berlin, Germany
3:49.2		Jules Ladoumegue (FRA)	1930-10-05	Paris, France
3:49.2		Luigi Beccali (ITA)	1933-09-09	Turin, Italy
3:49.0		Luigi Beccali (ITA)	1933-09-17	Milan, Italy
3:48.8		Bill Bonthron (USA)	1934-06-30	Milwaukee, USA
3:47.8		Jack Lovelock (NZL)	1936-08-06	Berlin, Germany
3:47.6		Gunder Hägg (SWE)	1941-08-10	Stockholm, Sweden
3:45.8		Gunder Hägg (SWE)	1942-07-17	Stockholm, Sweden

Figure 4.1: Screen Shot of 1500 meter World Records Wikipedia Table. *This screen shot shows a portion of the Wikipedia page that contains the results for the world records in the men's 1500 meter race. See Figure 4.2 for a screen shot of the top of the page.*

A first impulse might be to copy and paste the table from our Web browser into a text editor and then ‘hand’ edit the text into a format that we can easily read into *R* with one of the functions we used to read rectangular data in Chapter 2. However, there are problems with this approach: it is error prone and not computationally reproducible. For example, we might accidentally miss the last 2 rows of the table when we copy it. Additionally, if in our editing we unknowingly delete or change a few values, then when we or someone else later

detects the problem, we are not able to retrace our steps to see where the error arose. This approach does not result in a record of the sequence of steps followed to produce the data frame. That is, we cannot check code that we used to create the table, fix that code, and repeat the process. Similarly, if we later decide to alter the processing of the table because, say, the format has changed slightly, then we cannot update our code to reflect the new processing and simply run it to recreate the data frame.

Time	Athlete	Date	Place
4:24 3/5	J. Borel (FRA)	1892	
4:21	Fernand Meiers (FRA)	1893-05-28	Paris, France
4:19 4/5	Felix Bourdier (FRA)	1894-07-22	Paris, France
4:18 2/5	Albin Lermusiaux (FRA)	1895-05-12	Paris, France

Figure 4.2: Screen Shot of 1500 meter Wikipedia page. *This screen shot shows the top of the Wikipedia Web page (https://en.wikipedia.org/wiki/1500_metres_world_record_progression) that contains a table of the world records in the men's 1500 meter race. See Figure 4.1 for a screen shot of the table. This page contains multiple tables, including world records for women and pre-IAAF (International Association of Athletics Federations) records.*

We briefly introduce the *HTML* format in Section 4.4, and in Section 4.4.1, we extract this table from the Web page using the `readHTMLTable()` function and create a data frame for analysis. *HTML* is covered in more detail in Chapter 12. ■

Example 4-3 Caching Web Page Updates

Internet search engines, such as Google, Bing, and Ask, keep copies of Web pages so that when you make a query, they can quickly search their stored pages and return their findings to you. These saved pages are called a Web cache. In order to keep the cache up to date, Web pages need to be visited regularly and the cache updated. To try to determine how often Web pages change and how often a site should be visited to keep the cache fresh, the updating behavior of 1,000 Web pages was studied. Each of these pages was visited every hour for 30 days. The page was compared to the previous visit, and if it had changed, the cache was updated and the time of the visit was recorded.

For example, below are records for the first 4 of these 1,000 Web pages. The 3 pieces of information provided in each row are: the type of domain of the page, the total number

of visits made to the page, and a comma-separated set of visits on which a change was observed. (These 3 pieces of information are separated by tabs.)

net	378	35, 134, 155, 157, 177, 204, 314, 315, 319, 350, 366, 369, 371
jp	707	552, 604, 672
com	418	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, ..., 417, 418
net	369	30, 36, 45, 65, 88, 154, 157, 166, 169, 197, 199, ..., 367, 369

We see that the domain of the 1st page is .net, which stands for a network technology; the next domain is .jp, which means the domain is registered in Japan; and the 3rd record is from a .com site.

According to the study design, each page was to be visited hourly for 30 days, which means that 24×30 , or 720 visits, were planned. Since the first visit has no previous version of the page to compare against, we have only 719 subsequent visits (or revisits) where we can observe a change. As noted above, the second value in a record provides the actual number of visits made to the page. These can differ from 719 if visits are unsuccessful. Our first 4 sites were successfully visited 378, 707, 418, and 369 times, respectively.

The last piece of information provides the ‘times’ when a change is observed. Notice that the site in the 2nd row had changed on 3 visits, the 552nd, 604th, and 672nd. In contrast, the 3rd site was visited 418 times, and from the partial display of times when a change was observed, it appears that the site had changed between every visit. If we think of a Web page as a record or observation, then our records do not form a typical rectangular shape with a fixed set of columns corresponding to specific variables. This situation often arises with, e.g., medical records where patients typically have a varying number of visits to a clinic. The notion of non-rectangular data impacts how the data are stored in the computer and how we write code to analyze the data. We address these issues in Section 4.3.1. ■

Example 4-4 Kiva Loans

Kiva [2] is a nonprofit organization whose mission is to “connect people through lending to alleviate poverty.” Essentially, Kiva allows people like you and me to volunteer small amounts of money to loan to people around the world who use it for such things as starting a small business. Like many Internet-based organizations, Kiva makes their data available via a Web service. This includes providing links for downloading the entire database of loans and lenders in both *JSON* and *XML* format. Open data such as Kiva’s create wonderful opportunities for statisticians, social scientists, and the interested public to explore and learn about the world of micro-financing. Moreover, the possibility of merging these data with other publicly available data sets, creates an even greater potential for discovery and understanding.

The API for Kiva’s Web services is described at <http://build.kiva.org/api>. (API stands for ‘a programming interface’, this interface describes how to access the data programmatically.) There we find that the information about the most recent loans is available at <http://api.kivaws.org/v1/loans/newest.json>. The ‘json’ at the end of the URL indicates that this is *JSON* content (another type of plain text file format). When we visit this page, we find the following, which we have formatted to make it easier to see the structure of the information,

```
{
  "paging": {
    "page": 1, "total": 6025, "page_size": 20, "pages": 302
  },
  "loans": [
    { "id": 984656,
```

```

    "name": "Khamheang Group",
    "description": {"languages": ["en"] },
    "status": "fundraising",
    "funded_amount": 0,
    "basket_amount": 0,
    "image": { "id": 2032579, "template_id": 1 },
    "activity": "Home Appliances",
    "sector": "Personal Use",
    "themes": ["Water and Sanitation"],
    "use": "to purchase TerraClear water filters
          so they can have access to safe drinking water.",
    "location": {"country_code": "LA", "country": "Lao PDR",
      "town": "Laos",
      "geo": {"level": "town", "pairs": "18 105", "type": "point" } },
    "partner_id": 393,
    "posted_date": "2015-11-28T22:30:08Z",
    "planned_expiration_date": "2015-12-28T22:30:08Z",
    "loan_amount": 325,
    "borrower_count": 7, "lender_count": 0,
    "bonus_credit_eligibility": false,
    "tags": []
  },
  .....
]
```

Notice the use of curly and square brackets, commas, and colons to delimit the various pieces of information. For example, this *JSON* “object” has two fields, `paging` and `loans`. The `paging` field is itself an object with four fields, which are all numeric values. The `loans` field is an array of objects. Each element in this array describes a loan with fields for such things as the loan identifier, the name and location of the lendee, the specific purpose of the loan, and the amount being sought.

We want to create a data structure that contains information about each loan, but it is not immediately obvious whether or not we can map the *JSON* content into a data frame. We do not know whether or not all the loans have the same fields of information. For example, do all loans have a `country_code` element associated with them? (It turns out they don’t.) Also, each loan contains a record of the payments made. If multiple payments have been made, then the information about each payment forms a ragged array where different loans have a different number of payment fields. More details about the *JSON* format would be helpful in determining how to organize the data in *R*. We cover this in Section 4.5, and we tackle the challenge of organizing these data into convenient data structures in Section 4.6.

Lastly, some data are either so large or so complex that they require a more programmatic approach to handling them. In Chapter 14 we consider situations that require programming to acquire and clean the data, in Chapter 9 we work with data in relational databases, and in Chapter 12 we examine data available through Web scraping, forms and APIs.

4.2 Lists

In Chapter 1 and Chapter 2 we examined three kinds of data structures, vectors, data frames, and arrays. We saw that vectors are ordered collections of values of the same type. Arrays are similar to vectors except that they also have shape information, e.g., an r -by- c matrix is a two-dimensional array which we can also treat as a vector of length $r \times c$. The data frame is an ordered collection of vectors of the same length and possibly different types. However, there are many situations where we want to group values together that are not necessarily all of the same length, or not necessarily all vectors. In Q.4-3 (page 152), we saw that each Web page has a record of the visits on which the page changed, and the number of these varies with the page. Also, we will soon see that the information provided for the `FrontRangeWeather` includes a data frame with one row for each station and an object with daily recordings of precipitation at each station. The weather station details and the daily precipitation are very different kinds of objects, and `FrontRangeWeather` is a container that allows us to group these different kinds of elements together. This container is a list data structure. The list structure is also useful as a container for the Web cache data..

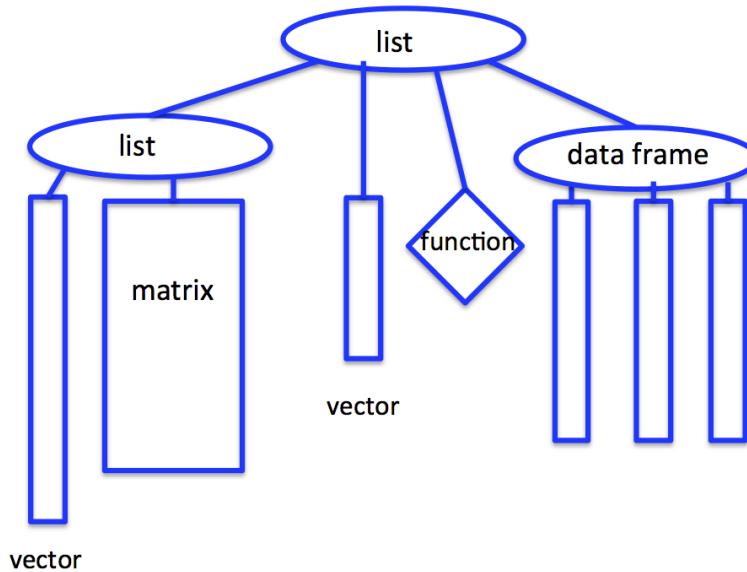


Figure 4.3: Conceptual Diagram of a List. *This diagram provides the basic structure of a simple list. This list has 4 elements: a list, a vector, a function, and a data frame. The list element contains a vector and a matrix. Notice that the data frame contains 3 vectors.*

Figure 4.3 provides a conceptual diagram for a simple list. This list consists of 4 elements: a list, a vector, a function, and a data frame. The 1st element (a list) contains two elements—a vector and a matrix. The 4th element (a data frame) contains 3 vectors which necessarily are all the same length.

We have created a list with the same structure shown in Figure 4.3 for us to explore. We load it into our session with

```
load("exampleList.rda")
```

Many of the functions that we used in Chapter 1 to glean information about vectors and data frames also accept a list as input. That is, we find the length of the list, the names of its elements (if they have names) and confirm its class with calls to `length()`, `names()` and `class()` as follows:

```
class(aList)
[1] "list"

length(aList)
[1] 4

names(aList)
[1] "listToo" "aVec"    "aFunc"   "aDF"
```

We see from these function calls that `aList` has 4 elements, which have been given names that suggest their types, i.e., the first element is another list, the second is a vector, the third is a function and the fourth is a data frame.

We print the contents of `aList` to the console to confirm that this list has the same basic structure as the list in Figure 4.3.

```
aList

$listToo
$listToo$aVec
[1] 1 3 5 7

$listToo$aMat
[,1] [,2]
[1,] 10 12
[2,] 14 16
[3,] 18 20

$aVec
[1] "a" "b" "c" "d"

$aFunc
function (x)
{
  sum(x^2)
}

$aDF
  id height sex
1   1      60   f
2   2      72   m
3   3      66   f
4   4      70   m
```

Notice that there is a vector within `listToo` and it is called `aVec`, which is the same name as the second element of `aList`. However, these two vectors are not the same objects. One is an element of `aList` and the other is an element of `listToo`, which in turn is an element of `aList`.

The \$-signs that are printed to the console preceding each element's name suggests that we can access the contents of a list in a similar fashion to vectors in a data frame. Accessing elements of a list and computing subsets on lists is the topic of the next section.

Lastly, we can also use the `str()` function, which is short for structure, to display the structure of `aList` in a compact fashion, e.g.,

```
str(aList)
```

```
str(aList)
List of 4
$ listToo:List of 2
..$ aVec: num [1:4] 1 3 5 7
..$ aMat: num [1:3, 1:2] 10 14 18 12 16 20
$ aVec   : chr [1:4] "a" "b" "c" "d"
$ aFunc  :function (x)
..- attr(*, "srcref")=Class 'srcref'
                  atomic [1:8] 1 14 1 37 14 37 1 1
... . . . - attr(*, "srcfile")=Classes 'srcfilecopy',
              'srcfile' <environment: 0x10e0cf4d8>
$ aDF     :'data.frame':        4 obs. of  3 variables:
..$ id     : int [1:4] 1 2 3 4
..$ height: num [1:4] 60 72 66 70
..$ sex    : Factor w/ 2 levels "f","m": 1 2 1 2
```

The return value contains a lot of information. We see `aList` is a list of 4 elements, and the names, types, and some of the contents of these elements.

4.2.1 Subsetting Lists

With lists, we can use the same style of subsetting as we do for vectors. In fact, a list is a special type of vector. The vectors we have worked with in Chapter 1 are actually atomic vectors, meaning all elements are of the same primitive type. A list is a non-atomic vector in that its elements can be any type of object.

We can subset by position, exclusion, logical, and name. For example, we compute a list of the 2nd and 4th elements of `aList` using the position of these elements in the list with:

```
aList[c(2, 4)]  
  
$aVec  
[1] "a" "b" "c" "d"  
  
$aDF  
  id height sex  
1   1      60   f  
2   2      72   m  
3   3      66   f  
4   4      70   m
```

Similarly, we arrive at the same subset when we use exclusion, logical, and names with the following computations, respectively,

```
aList[ -c(1, 3)]
aList[c(FALSE, TRUE, FALSE, TRUE) ]
aList[c("aVec", "aDF") ]
```

4.2.1.1 Accessing An Element of a List

When we subset a list, the return value is a list. Even when we ask for one element in the list, we obtain a list as the return value. For example, when we ask for the 2nd element of `aList`, we get a list of length 1 in return, e.g.,

```
aList[2]
$aVec
[1] "a" "b" "c" "d"
```

```
class(aList[2])
[1] "list"
```

This is consistent with subsetting vectors. That is, when we take a subset of a vector, the return value is a vector, even if it has only one element. However, at times, we want to drop the list container and work directly with the element.

If the elements have names, we can use \$-notation to work directly with the element, e.g.,

```
aList$aVec
[1] "a" "b" "c" "d"
```

Notice that the return value is printed to the console in a slightly different format, which indicates that it is a vector and not a list. We can operate on this vector, just as we do with other vectors. For example we can find its length and class with

```
length(aList$aVec)
[1] 4
```

```
class(aList$aVec)
[1] "character"
```

Also, we can use subsetting techniques to access elements in this vector, e.g., we can subset by position to change the 3rd element in `aList$aVec` to "z" with

```
aList$aVec[3] = "z"
```

Now `aList$aVec` has values "a" "b" "z" "d".

We can even invoke the function `aFunc()` in `aList` with the help of the \$-notation, e.g., `aList$aFunc(1:3)`. Notice the return value (14) is the sum of the squares of 1, 2, and 3.

Another way to extract an individual element from a list uses double-square brackets, e.g.,

```
aList[[2]]
[1] "a" "b" "z" "d"
```

Again, we can access subsets of this vector with, e.g.,

```
aList[[2]][3:4]
[1] "z" "d"
```

Notice that with `aList[[2]][1:2]` we obtain a vector of length 2, which consists of the 3rd and 4th elements of the 2nd element of `aList` (which is a vector). We can also use double square brackets to call `aFunc()`, e.g., `aList[["aFunc"]](1:3)` and `aList[[3]](1:3)` both return 14.

We provide a few more examples of subsetting lists with double-square brackets and \$-signs to help solidify our understanding of this notation. Recall that the first element of `aList` is also a list and its name is `listToo`. Compare the return values from the following commands to help check your understanding of subsetting lists.

```
aList[[1]][1] ①
$aVec
[1] 1 3 5 7

aList[[1]][[1]] ②
[1] 1 3 5 7

aList$listToo[[1]] ③
[1] 1 3 5 7

aList$listToo$aVec ④
[1] 1 3 5 7

aList[[1]][[2]] ⑤
[,1] [,2]
[1,] 10 12
[2,] 14 16
[3,] 18 20

aList[1][[2]] ⑥
Error in aList[1][[2]] : subscript out of bounds

aList[1][2] ⑦
$<NA>
NULL
```

- ① The double square bracket returns the first element in the list, which is `listToo`. Then, the second subset operation, i.e., the subset with the single square bracket, returns a list with one element, `aVec`.
- ② The first use of double square brackets in this expression returns the `listToo` list. Then the second set of double square brackets returns the first element in `listToo`, which is the vector `aVec`.

- ③ When the elements of the list have names, we can use the \$-notation to access them. This expression is equivalent to the previous expression that uses two sets of double-square brackets, and both are equivalent to `aList[[1]]$aVec`.
- ④ We also can chain \$-signs together. In this case, we again get the vector `1 3 5 7` as return value.
- ⑤ This sequence of two sets of double-square brackets returns the 2nd element of the 1st element of our list, which is the matrix `aMat`.
- ⑥ The first set of brackets in this expression are single so the return value is a list of length 1. The second set of brackets are double so the return value is the 2nd element of the list. However, this element does not exist because we have a list of length 1, so we get an error.
- ⑦ In contrast to the previous expression, these two sets of single square brackets returns a list of length 1. However, the element in this list is `NULL`.

A Data Frame is a List

It may have already occurred to you that data frames have many similarities to lists. In fact, a data frame is a special case of a list where all the elements are vectors of the same length. This means that we can use the double-square bracket form of subsetting with data frames, e.g., `aList$aDF[["sex"]]` retrieves the vector called `sex` in our example data frame. Additionally, we can compute a subset of the columns of the data frame with, e.g., `aList$aDF[2:3]`. We did not include a comma within our square bracket so *R* interprets this subset command as applying to the columns, i.e., to the elements of the list/data frame. The return value is a 2 column data frame that contains all rows from the original data frame.

The List

The list is an ordered container of heterogeneous objects. That is, it is a vector of heterogenous objects. A list can include as elements a combination of vectors, multi-dimensional arrays, data frames, lists, functions, etc.

A subset can be computed on a list by position, exclusion, logical, name, and all.

A single element can be extracted from a list with double-square brackets, i.e., `[[]]`. This is different from a subset of 1 element of a list, which is a list of length 1. If the elements in the list are named, then \$-notation can be used to extract one element from a list, e.g., `aList$aVec`.

The subset operators can be applied in sequence to compute a subset of elements from an element of a list, e.g., `aList[[1]]$aMat[2, 1]` returns the value in the 2nd row and 1st column of `aMat`, which is an element of the 1st element of `aList`.

The `lapply()` and `sapply()` functions apply a function, supplied as an argument, to each element of a list. The return value is a list, or, with `sapply()` a vector is returned if the return value can be reduced to a vector.

4.2.2 Applying Functions to Elements of a List

As with data frames, we can apply a function to each element of a list. We can use the function `lapply`, which stands for “list apply”. We can also use `sapply()`, which performs the same operations but simplifies the return value to a vector when possible. For example, we can find the mean of each element in the list `listToo` with

```
lapply(aList$listToo, mean)
```

```
$aVec
[1] 4
```

```
$aMat
[1] 15
```

Notice that the mean of the matrix is the average of all elements in the 3 by 2 matrix. Since the return values are single numerics, we can use `sapply()` to “simplify” the return value with

```
sapply(aList$listToo, mean)
```

```
aVec aMat
     4    15
```

Additionally, we can find the class of each element in `aList` with

```
sapply(aList, class)

listToo      aVec      aFunc      aDF
"list" "character" "function" "data.frame"
```

We obtain the length of each element with

```
sapply(aList, length)

listToo      aVec      aFunc      aDF
2           4         1         3
```

We now have the tools and understanding of the list structure to examine and explore the `Rda` file introduced in Q.4-1 (page 150).

4.2.3 Exploring Rainfall on the Colorado Front Range

Let’s begin our exploration of the Front Range weather data by getting a better understanding of its structure. After we load the data into `R`, we can call `class()` to find that `FrontRangeWeather` is a list, as mentioned in Q.4-1 (page 150). That is,

```
load("FrontRangeWeather.rda")
class(FrontRangeWeather)

[1] "list"
```

Earlier, we found the length and names of the elements in `FrontRangeWeather` with calls to `length()` and `names()`, e.g., `length(FrontRangeWeather)` returns 3 and `names(FrontRangeWeather)` returns:

```
[1] "days"      "precip"    "stations"
```

We continue our investigation of the contents of `FrontRangeWeather` and apply functions to its elements. For example, we can find the class of each of the 3 elements in `FrontRangeWeather` with

```
sapply(FrontRangeWeather, class)

days      precip      stations
"list"    "list"    "data.frame"
```

Both `days` and `precip` are lists and `stations` is a data frame. We can find the length of each of these objects by applying `length()` to each element of `FrontRangeWeather` with

```
sapply(FrontRangeWeather, length)
      days    precip stations
      56        56         4
```

We find that both `days` and `precip` have length 56 and `stations` has length 4. Recall that the length of a data frame is the number of columns, or vectors in the data frame. The `str()` function provides this information too, but it is not as succinct as we might like because information is given for all 56 vectors in `days` and `precip`.

Let's dig a little deeper to learn more about the `stations` data frame. Given it has only 4 variables, we can use `head()` to view the first few rows of the data frame with

```
head(FrontRangeWeather$stations)

  station     lon     lat elev
1 st050183 -105.53 40.22 8450
2 st050263 -105.88 39.00 8920
3 st050712 -104.32 38.87 6040
4 st050843 -105.27 40.03 5420
5 st050945 -104.33 40.65 4880
6 st051179 -104.13 39.75 5150
```

This data frame has information about the weather stations, including location (latitude and longitude) and elevation, as well as an identifier for the stations.

We can explore these data graphically by plotting latitude against longitude. We provide some context to such a plot by placing the stations on a map that includes the Colorado state boundary, and we color the points according to elevation. Our map appears in Figure 4.4.

We made this map with the `map()` function in the `maps` package as follows:

```
library(maps)
map('state', fill=TRUE, col="gray95",
    xlim = c(-110, -101), ylim = c(36, 42))

elevCut = cut(FrontRangeWeather$stations$elev, breaks = 7)
rcolors = rainbow(n = 7, alpha = 1)

with(FrontRangeWeather$stations,
     points(x = lon, y = lat, pch = 19, col = rcolors[elevCut]))
```

Note that when we call `map()`, we specify a range for the latitude and longitude values that are a bit larger than Colorado so the map shows the boundaries with the neighboring states. Then we add points to this map with `points()`. These are placed at the latitude and longitude of the 56 weather stations. We discretize `elev` (with the `cut()` function) and use `elevCut` to select a color from the rainbow palette (in `rcolors`). Red represents the lowest and blue the highest elevation with orange and yellow falling in between.

From the map we see where the Front Range is within the state of Colorado. Also, the colors indicate that lower elevations are in the eastern region of the Front Range. This makes sense because the Rocky mountains are in the western part of the state and the eastern part corresponds to the high plains.

Now that we have a much better picture of where the weather stations are located, let's further explore the other two elements of `FrontRangeWeather`: `days` and `precip`. We

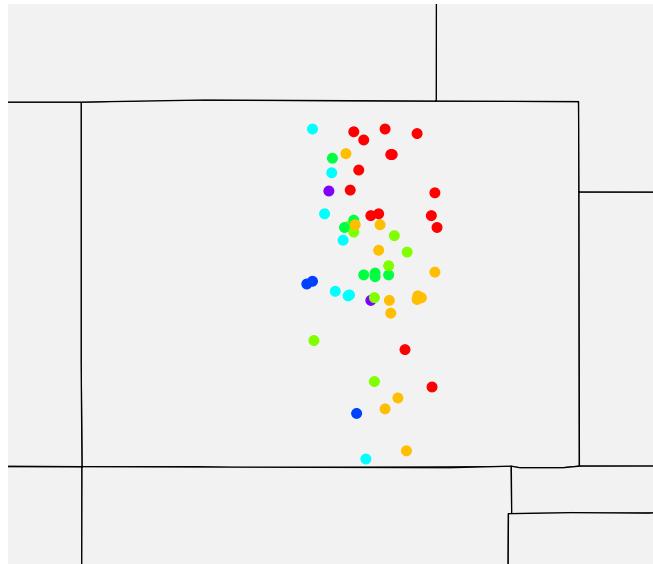


Figure 4.4: Location of Weather Stations in the Colorado Front Range. *The points on this map of Colorado show the locations of the weather stations in `FrontRangeWeather`. The color of the points indicates the elevation of the station. Red represents the lowest elevations, blue the highest, and orange and yellow in between. It is evident from the locations of these points that the Front Range corresponds to a north-south swath of the state, and the higher elevations are in the western portion nearest the Rocky mountains.*

now know that both `days` and `precip` are lists of 56 elements. This count (56) corresponds to the number of weather stations in `station`. You can check this with, e.g., `dim(FrontRangeWeather$stations)`. To find the class of these elements we use `sapply()`, e.g.,

```
sapply(FrontRangeWeather$days, class)
st050183  st050263  st050712  st050843  st050945  st051179
"numeric" "numeric" "numeric" "numeric" "numeric" "numeric" ...
```

These are all numeric vectors, and these vectors have names that match the values in the `station` variable in the `stations` data frame. We check the lengths of these vectors with `sapply(FrontRangeWeather$days, length)` and find that they have different lengths, e.g. the first station has 9878 measurements, the 2nd has 6751, and the 3rd has only 3959. When we apply `class()` and `length()` to the elements of `precip`, we find that its 56 elements also are numeric vectors, and their lengths match the the lengths of the vectors in `days`. That is,

```
all(sapply(FrontRangeWeather$days, length) ==
  sapply(FrontRangeWeather$precip, length))
```

returns TRUE.

We have enough information to sketch a conceptual diagram of `FrontRangeWeather` (see Figure 4.5). From this diagram we see that `FrontRangeWeather` has three elements, two lists (`days` and `precip`) and a data frame (`stations`). The data frame has 56 rows, corresponding to the 56 weather stations, where precipitation measurements are recorded, and 4 variables providing the latitude, longitude, elevation, and station identifier. The two lists (`days` and `precip`) each contain 56 numeric vectors, one for each weather station. Their lengths differ according to the number of days on which a station recorded precipitation.

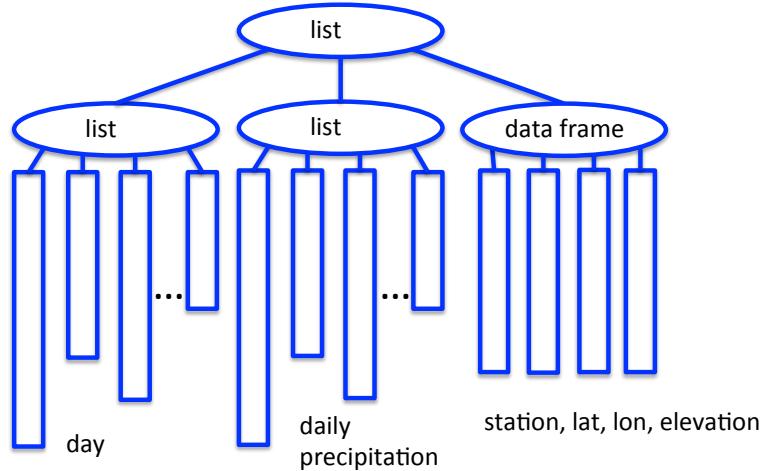


Figure 4.5: Conceptual Diagram of Colorado Front Range Data. This diagram provides a sketch of the structure of `FrontRangeWeather`. This object is a list with 3 elements, which are two lists and a data frame. The data frame contains information about each of the 56 weather stations (rows), including the station identifier, latitude, longitude, and elevation. For each of the stations, there is a numeric vector in the two lists, i.e., each list contains 56 numeric vectors. These vectors contain the date (in `day`) and the amount of precipitation recorded on that day (in `precip`).

Let's examine the first few values for the first station with

```
head(FrontRangeWeather$days[[1]])
[1] 1948.585 1948.587 1948.590 1948.593 1948.596 1948.598
```

These numeric values have the format *year + days/365* so, e.g., January 1, 1950 is 1950.003 and February 20, 1950 is $1950 + (31 + 20)/365$ or 1950.14.

The precipitation recorded on the first 6 days at this weather station are

```
head(FrontRangeWeather$precip[[1]])
[1] 0 10 11 1 0 0
```

These measurements are recorded in 100ths of an inch so a value of 11 is 0.11 inches of rain. Let's examine with a line plot the relationship between precipitation and date for the 1st station. We make this plot with

```
with(FrontRangeWeather,
  plot(precip[[1]] ~ days[[1]], type = "l",
    xlab = "Date", ylab = "Precip (100s inch)"))
```

The `with()` function is a convenience function to save us some typing when we specify the elements in a list of data frame. For example, the following expression: `plot~(FrontRangeWeather$precip[[1]] ~ FrontRangeWeather$days[[1]], ...)` is equivalent to the above.

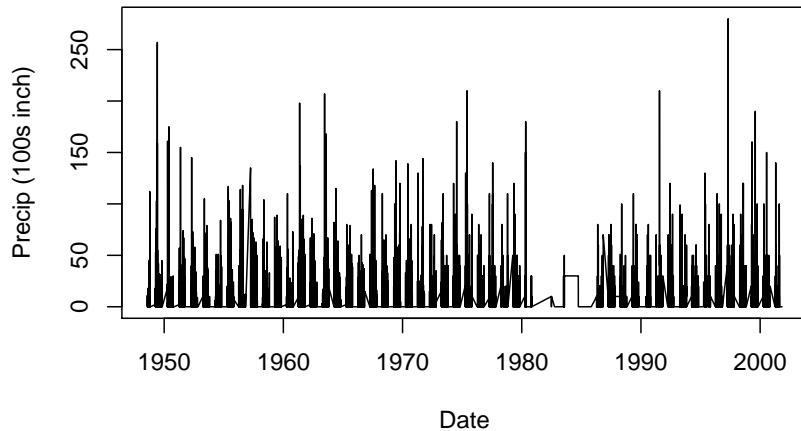


Figure 4.6: Precipitation Recorded at One Weather Station. *This line plot of the daily precipitation (rainfall) measured at the 1st weather station in `FrontRangeWeather` has a gap where only a few measurements were recorded between 1980 and 1985. Additionally, there are regular annual gaps over the entire range of observations. These are due to the winter months when rainfall was not measured.*

The plot appears in Figure 4.6. We make a few observations: a) at the time these data were published, this weather station had been in operation for over 50 years; b) there are very few measurements between 1980 and 1985, which indicates either the equipment/station was not in operation or those measurements were lost; c) there appear to be regular annual gaps in precipitation; d) there are a few spikes in precipitation indicating a few large storms with over 2.5 inches of rainfall.

Let's examine these gaps in operation for all of the stations in more detail. To do this, we make a specialized dot chart. In this plot, we construct a row of dots for each station. The dots are placed at the dates that a recording was made, even if the measurement was 0. We use the `stripchart()` function to make the plot as follows:

```
plot(c(1948, 2002), c(1, 56),
      xlab = "", ylab = "Station", type = "n", axes = FALSE)
stripchart(FrontRangeWeather$days, add = TRUE,
           col = "blue", pch = ".")
axis(1)
```

We see in Figure 4.7 narrow vertical white stripes. These stripes correspond to a consistent lack of measurements; in other words, all stations are missing data for these days. This occurs because we do not have precipitation measurements for the winter months when the precipitation is in the form of snow and there is not a risk of flooding. Longer horizontal

white spaces indicate that a station is not in operation. We confirm that the first row has a white region in the early to mid 1980s. We also see that many stations have not been in operation for 50 years.

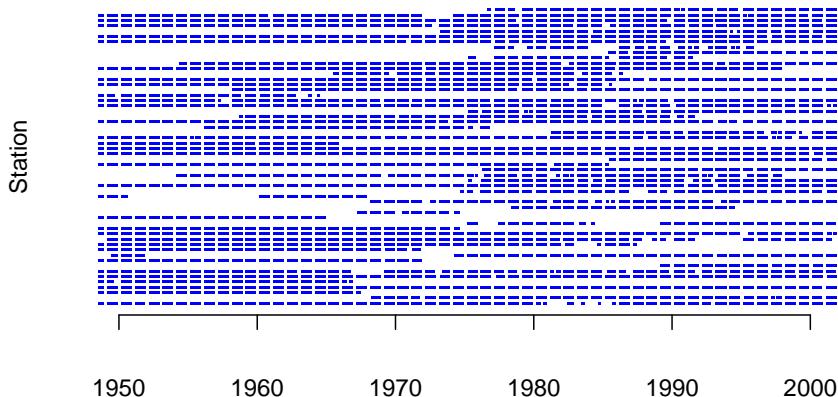


Figure 4.7: Dates of Operation of the Weather Stations. *This strip chart has a blue dot for each measurement taken at each weather station. All dots for one weather station appear on the same horizontal line. The dots for a station are placed at the date when the recording was made. The bottom row of dots corresponds to the 1st weather station. The gap from 1980 through 1985 is apparent from the horizontal gap in the 1st row. The plot has white vertical stripes which make clear the regularity of the lack of measurements in the winter months for all stations.*

We have successfully determined the structure of `FrontRangeWeather` and made a cursory examination of the values to learn about the units of measurement, range of values, and missing-ness of the data. We are ready to more carefully analyze the data, but that is not our goal here.

4.3 Reading Data into a Character Vector

We have the ability to read text data into *R* as a character vector, where each line in the input file becomes one string (element) of the character vector. Then, the length of the vector equals the number of lines read from the file. This approach can be useful when the text needs special processing that is not readily accomplished through the use of functions such as those described in Chapter 2. That is, we can write specialized code to process the strings and create the desired data structure. Of course, we do not want to take this approach unless the more standard approaches are inadequate. The functions such as `read_delim()`, `read.table()`, etc. tend to be more robust than our code and we should use them.

The `readLines()` function reads text into *R* and returns a character vector. In addition to supplying the ‘connection’ (such as a file name or *URL*), we can also specify the number of lines to read with the `n` argument. The following example provides a demonstration of how to use `readLines()` and why we may need to read the contents of a file into a character vector. The processing of the strings in this example are simple. More complex processing of strings is described in Chapter 8 on regular expressions and string manipulation.

4.3.1 A Study of Web Page Updates

Recall from Q.4-3 (page 152) that the Web cache study consists of 3 pieces of information for each Web page: the domain of the page, the total number of visits made to the page, and a comma-separated set of visits when a change was observed. We re-display the first 4 of the 1000 records below for convenience.

```
net      378    35,134,155,157,177,204,314,315,319,350,366,369,371
jp       707    552,604,672
com      418    1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,...,417,418
net      369    30,36,45,65,88,154,157,166,169,197,199,...,367,369
```

As noted earlier, these 3 pieces of information are separated by tab characters in the file. We can use `read_delim()` and provide the tab character as the separator to read these data into *R* as a data frame, e.g.,

```
tryDF = read_delim("webCache.txt", delim = "\t")
dim(tryDF)
```

```
[1] 1000     3
```

The first two rows of this data frame are:

```
tryDF[1:2, ]
  V1   V2                               V3
1 net 378 35,134,155,157,177,204,314,315,319,350,366,369,371
2 jp   707                           552,604,672
```

Clearly the data are not in a convenient format for analysis because all of the Web page's changes are in a character string. This data frame is not a workable structure. A better structure is one where the changes are numeric vectors and since these vectors are of different lengths, we want a list of vectors. The domain and the number of visits can remain as a 1000 by 2 data frame, or it might be simpler to keep them as 2 separate vectors. If we want one container for all of 3 pieces of information, then it is simpler to access the domains and the number of visits as vectors in the list rather than as vectors within a data frame within the list. In this case the structure that we want is a list with three elements containing a character vector of domains, a numeric vector of the total number of visits to a page, and a list of 1000 numeric vectors, each containing the visits on which a change was observed. See Figure 4.8 for a diagram of this data structure.

We can continue to work with the `tryDF` data frame to create our desired list but we leave this approach as an exercise. Instead, we use the `readLines()` function to read the file into *R* as a character vector of length 1000. Each line in the file corresponds to a string in the return vector. We do this with

```
txt = readLines("webCache.txt")
```

We confirm that indeed we have a character vector of length 1000 with

```
class(txt)
[1] "character"
```

```
length(txt)
```

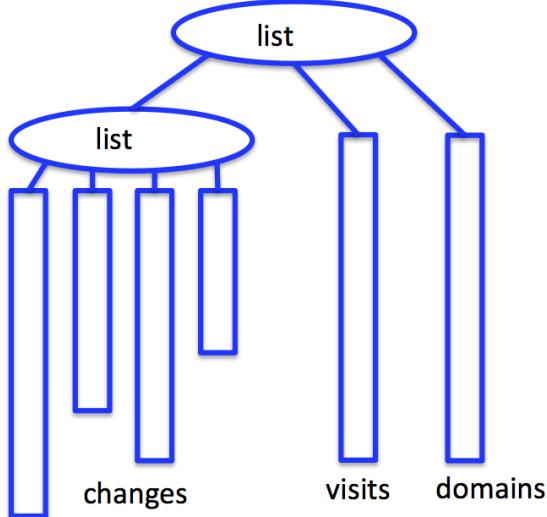


Figure 4.8: Conceptual Diagram of Web Cache Data. *This diagram provides a sketch of the desired structure for the Web cache data. The structure is a list with 3 elements: a list and 2 vectors. The `domains` vector is a character vector with 1,000 strings, each denoting the domain of the corresponding Web page. The `visits` vector is a numeric vector with the total number of successful visits to the page. The `changes` list is a collection of 1,000 numeric vectors. Each vector corresponds to a Web page, and the vector contains the times when a change was observed on the page.*

```
[1] 1000
```

We can split these strings up into the desired three pieces with a call to `strsplit()` as follows

```
els = strsplit(txt, "\t")
```

This function returns a list, where each element is a character vector containing the substrings that result from the splitting action. For example, the first two strings in `txt` are

```
txt[ 1:2 ]
```

```
[1] "net\t378\t35,134,155,157,177,204,314,315,319,350,366,369,371"
[2] "jp\t707\t552,604,672"
```

(These match the first two records in the file.) And, the first two elements of `els` are

```
els[1:2]
```

```
[[1]]
[1] "net"
[2] "378"
[3] "35,134,155,157,177,204,314,315,319,350,366,369,371"

[[2]]
[1] "jp"           "707"          "552,604,672"
```

We have successfully split each of these strings into 3 substrings with the 1st substring containing the domain, the 2nd the total visits, and the third the comma-separated times. If we can extract the 1st element from each vector then we have all of the domains. We can extract the 1st substring of the first record with

```
els[[1]][1]
[1] "net"
```

Similarly, we extract the domain of the 2nd page with `els[[2]][1]`. Essentially, we want to apply `[1]` to each element of `els`. In fact, the square bracket operator is a function and we can use it in a call to `sapply()`. Before we do, let's try to call the square bracket function using standard function syntax, e.g., `f(x)`. There are two inputs to this function—the vector to be subsetted and the position(s) that we want. We can construct the function call with:

```
"["(els[[2]], 1)
[1] "jp"
```

This is a bit strange looking, but we see that we have provided the `[-operator with two inputs ([[2]] and 1) and the [-operator has returned the desired domain. We needed to put the [-operator in quotes to avoid a syntax error. Now we are ready to apply the [-operator to each element of the els vector. We do this with`

```
domains = sapply(els, "[", 1)
```

Notice that we supplied the square bracket function with the additional argument value of `1`. We also use `sapply()` rather than `lapply()` because we know that the return value from each call is a character vector of length 1 and we want these simplified into a vector. Let's check the first few elements of `domains` with

```
head(domains)
[1] "net" "jp"  "com" "net" "ca"   "de"
```

It appears that our extraction of the domains into a character vector has worked as expected.

We can similarly extract the total number of visits to a page with

```
visits = sapply(els, "[", 2)
head(visits)
[1] "378" "707" "418" "369" "719" "612"
```

We convert these character strings to numeric with

```
visits = as.numeric(visits)
```

Now we have two vectors `domains` and `visits`. Our remaining task is to create the list of vectors of changes.

We begin by extracting the strings of changes from `els` with

```
changes = sapply(els, "[", 3)
changes[1:2]
```

```
[1] "35,134,155,157,177,204,314,315,319,350,366,369,371"
[2] "552,604,672"
```

We can use `strsplit()` to split the string of changes. This time we want the split to be on commas, i.e.,

```
changes = strsplit(changes, ", ")
changes[1:2]

[[1]]
[1] "35"   "134"  "155"  "157"  "177"  "204"  "314"  "315"  "319"
[10] "350"  "366"  "369"  "371"

[[2]]
[1] "552"  "604"  "672"
```

As with `visits`, we want to convert these substrings into numeric values. We do this with

```
changes = lapply(changes, as.numeric)
changes[1:2]

[[1]]
[1] 35 134 155 157 177 204 314 315 319 350 366 369 371

[[2]]
[1] 552 604 672
```

We see that we have created `changes` as a list of numeric vectors.

Let's explore a bit further to check our work. We can examine the distribution of the total number of visits with

```
hist(visits, breaks = 50, main = "")
```

We see in this histogram (Figure 4.9) that the vast number of sites were visited the planned number of times or nearly so. Yet, all did not go as planned and some sites were visited fewer than 100 or 200 times.

Let's also examine the distribution of the number of changes observed for the sites. This information can be obtained from the lengths of the vectors in `changes`. We can find these values with

```
numChanges = sapply(changes, length)
head(numChanges)

[1] 13    3 385  27 185  17
```

Before we make a histogram of these values, let's restrict the sites that we examine to those that have been visited close to the planned number of times (719). We use `visits` to take a subset of `numChanges` and then make the histogram with

```
hist(numChanges[ visits > 700 ], breaks = 25, main = "",
     xlab = "Number of Changes Observed")
```

We can see in this histogram (Figure 4.10) that the distribution of the number of changes is unimodal and skewed right, i.e., many pages change a few times and some pages change a great number of times.

Our final task is to collect these three pieces of data into a list. We do this with

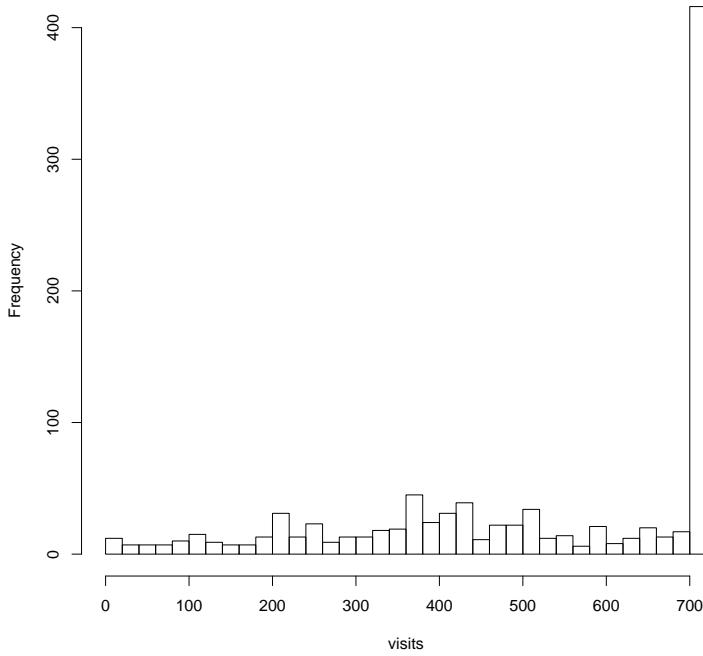


Figure 4.9: Histogram of the Total Number of Visits to Web Pages. *This histogram shows the distribution of the number of visits to a Web page for the 1,000 pages. The plan was for each page to be visited 719 times, and the bar above 700-720 shows that at least 400 pages were visited the planned number of times. However, many sites were visited fewer than 700 times, including a few sites that have less than 100 visits.*

```

cache = list(domains, visits, changes)
class(cache)

[1] "list"

length(cache)

[1] 3

names(cache)

[1] "domains" "visits"  "changes"

```

We can continue to analyze these data and we can collaborate with other researchers by sharing `cache`. To do this, we save `cache` to an `.rda` file with, e.g., `save(cache, file = "webCache.rda")`. An `rda` file can contain multiple objects so, e.g., if we find it easier to not group our 3 objects into 1 list, then we can save them in an `rda` file as follows: `save(domains, visits, changes, file = "webCache.rda")`.

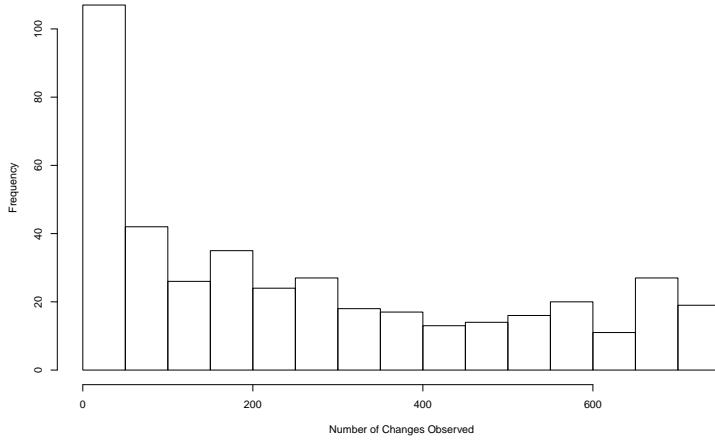


Figure 4.10: Histogram of the Total Number of Changes to Web Pages. *The distribution of the number of changes observed is skewed right with a peak in the 0-25 range and a long right tail with some pages changing on every or nearly every visit. This histogram includes only those pages with at least 700 visits.*

4.4 Reading Data from a Web Page

Web pages often contain tables of data that we want to extract and analyze. Although the data might appear to be a simple plain text table, a table in a Web page is typically marked up with tags that describe the cell and how the information should be rendered in the Web page, e.g., left or right justified, padding around the value, and format of the table header. If we want to extract just the values from the table, we need to separate the content from the mark up. For example, we have created a very simple *HTML* page containing a table of the family data for the family in Q.1-7 (page 21). Figure 4.11 displays a screen shot of this page as it is displayed in a browser. In the following example, we show how to easily extract these values from the table and create a data frame. We use the `readHTMLTable()` function in the *XML* package to do this.

Example 4-5 Extracting Data From an HTML Table

The actual content of this *HTML* file appears below. The terms such as `<title>` and `<table>` are *HTML* tags, and we describe them in greater detail in Chapter 12. For now, we point out that the value in a cell is between `<td>` and `</td>` tags and each row of the table begins with `<tr>` and ends with `</tr>`.

```

<html>
<head>
<title>Example table</title>
</head>
<body>
<h2>A Family</h2>

<p>

```

A Family

This is a simple Web page with a table of data.

name	sex	age	height	weight	bmi	overWt
Tom	m	77	70	175	25.16	T
May	f	33	64	125	21.50	F
Joe	m	79	73	185	24.46	F
Bob	m	47	67	156	24.48	F
Sue	f	27	64	105	18.06	F
Liz	f	33	68	190	28.95	T
Jon	m	67	68	185	28.19	T
Sal	f	52	65	124	20.68	F
Tim	m	59	68	175	26.66	T
Tom	m	27	71	215	30.05	T
Ann	f	55	67	166	26.05	T
Dan	m	24	66	140	22.64	F
Art	m	46	66	150	24.26	F
Zoe	f	48	62	125	22.91	F

Figure 4.11: Screen Shot of a Simple *HTML* Web Page. *This screen shot shows a very plain Web page with little content except a table of data for the 14-member example family from Chapter 1. Note that the column labeled ‘bmi’ provides values to 2 decimal places, and the ‘overWt’ column has capital T or F for whether or not the person’s BMI is over 25.*

This is a simple Web page with a table of data.

```
</p>
<table>
<tr><th>name</th><th>sex</th><th>age</th><th>height</th>
    <th>weight</th><th>bmi</th><th>overWt</th></tr>
<tr><td>Tom</td><td>m</td><td>77</td><td>70</td><td>175</td>
    <td>25.16</td><td>T</td></tr>
<tr><td>May</td><td>f</td><td>33</td><td>64</td><td>125</td>
    <td>21.50</td><td>F</td></tr>
<tr><td>Joe</td><td>m</td><td>79</td><td>73</td><td>185</td>
    <td>24.46</td><td>F</td></tr>
...
</table>
</body>
</html>
```

If you understand a little about *HTML* then it seems possible to extract the information from the cells of a table and convert them into a data frame in *R*. *HTML* files (and the tables they contain) have a lot of structure, which we discuss in greater detail in Chapter 12. For now, we simply use the `readHTMLTable()` function in the *XML* package, which exploits this structure to find and extract cell values from an *HTML* table. We load the *XML* package and call `readHTMLTable()` as follows:

```
library(XML)
familyH = readHTMLTable("family.html", header = TRUE, which = 1,
                       colClasses = c("character", "factor", "numeric",
                                     "numeric", "numeric", "numeric", "logical"))
```

Notice that we specified that the table has a header so the first row of the table is used for

variable names. The `which` argument indicates that we want the first (and in this case only) table in the page. If we do not use this argument then all tables in the page are returned as a list of data frames. The `colClasses` argument specifies the class for each of the variables (i.e., columns); we have indicated that the first column is a character vector, the second is a factor, the next four are numeric, and that last is logical. Without these specifications, all the columns are treated as factor vectors.

We confirm that the data frame contains the data from the Web page,

```
head(familyH)
```

	names	sex	age	height	weight	bmi	overWt
1	Tom	m	77	70	175	25.16	TRUE
2	May	f	33	64	125	21.50	FALSE
3	Joe	m	79	73	185	24.46	FALSE
4	Bob	m	47	67	156	24.48	FALSE
5	Sue	f	27	64	105	18.06	FALSE
6	Liz	f	33	68	190	28.95	TRUE

And, we can check the class of the variables with a call to `sapply()`. ■

Web pages often have more complex content than the page shown in Figure 4.11. In the next section, we return to the problem of extracting data from a table on a Wikipedia page. This page has several tables and figures, but the process of extracting the table of interest is nearly as simple as this example.

4.4.1 World Records in the Men's 1500 meter

We saw in Figure 4.2 that the Wikipedia Web page `1500_metres_world_record_progression` contains tables of world records for the 1500 meter race. The particular table that we want is shown in Figure 4.1. It is one of several tables in the page. We access this page from within R with the `getURL()` function in the `RCurl` package. We do this with

```
library(RCurl)
wikipedia = "https://en.wikipedia.org/wiki/"
wikipediaPage =
  paste(wikipedia,
        "1500_metres_world_record_progression", sep = "")
htmlContent = getURL(wikipediaPage)
```

The page is now in the character vector `htmlContent`. We use the `readHTMLTable()` function to extract all the tables as a list of data frames with

```
library(XML)
result = readHTMLTable(htmlContent)
```

We examine the first few rows of each data frame with `sapply(result, head, 2)` or with `str(result)` to determine that the desired data frame is the second element of `result`. We can work directly with `result[[2]]` or process `htmlContent` again and this time specify that we want only the 2nd table. We do this with

```
tableWR = readHTMLTable(htmlContent, which = 2,
                        stringsAsFactors = FALSE)
```

We kept the variables as strings because we want to specially process the race time and date. We combine the `min` and `sec` vectors to create `runTime` in seconds. We also convert the `Date` to a special format. The plotting (and other) functions recognize this format and make sensible axes with the dates. A step plot (Figure 4.12) shows the setting/breaking of the world record from 1912 to 2015. We make this plot with

```
tempTime = as.POSIXlt(tableWR$Time, format = "%M:%OS")
finalTable =
  data.frame(runTime = (tempTime$min * 60) + tempTime$sec,
             recordSet = as.Date(tableWR$Date,
                                  format = "%Y-%m-%d"))
plot(runTime ~ recordSet, data = finalTable,
     type = "s", xlab = "Year", ylab = "Time (sec)",
     main = "World Records in Men's 1500 meter",
     xlim = c(1912, 2015))
```

The current record was set in 1998, i.e., it has held for 18 years.

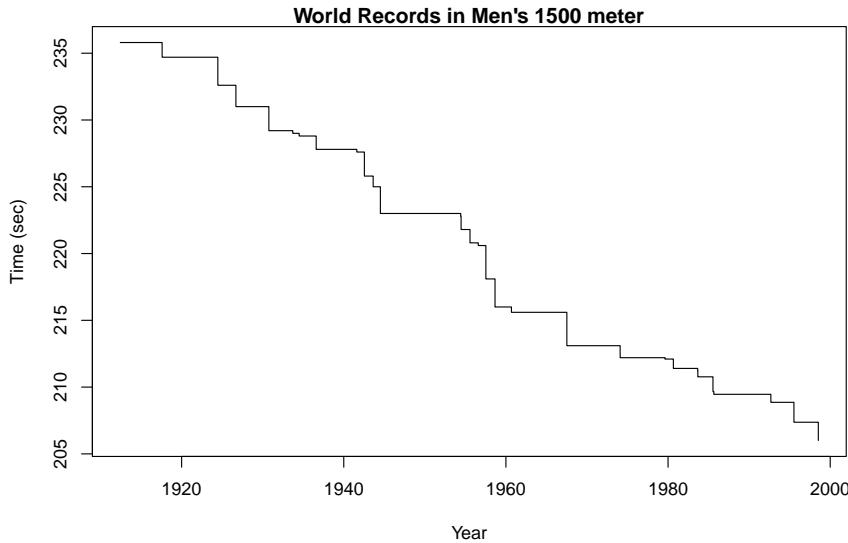


Figure 4.12: World Record in Men's 1500 m. *This step plot has a step for each time the world record for the men's 1500 meter was broken. The horizontal segments indicate the length of time the record stood and the vertical drops show the seconds by which the record was broken.*

4.5 Reading *JSON* Formatted Data

JSON (JavaScript Object Notation) is a simple, lightweight format that originates from the *JavaScript* [1] language syntax for creating objects. Data given in *JSON* format can be used directly in *JavaScript* code such as in Web pages. These characteristics make *JSON* a

valuable tool for working with data, especially data coming from Web services or displayed in browsers.

The *JSON* format is very simple. As with most data-oriented computer languages, *JSON* has common primitive data types: boolean/logical (*true* and *false*), number, and string. Unlike in *R*, these are scalars in *JSON*. Also, with *JSON* there is only one type of number—a real or floating-point value, which is a *numeric* in *R*. *JSON* does not have the notion of a missing value, infinity, or “not a number”, i.e., *R*’s `NA`, `Inf`, `Nan`, respectively. It does have the notion of *null*, the empty “object.”

These four types (boolean, number, string, and *null*) make up the entire collection of scalar values so there is a reasonably obvious mapping between *JSON* and *R* primitive data types. Specifically, *JSON* scalars map to *R* vectors with length 1. The *true* and *false* map to `TRUE` and `FALSE`, respectively; *null* maps to `NULL`; numbers map to *numeric* vectors; and a *JSON* string maps to a character vector of length 1.

In addition to the scalar types, *JSON* has two container data types for collections of zero or more values. In *R*, these correspond to unnamed and named vectors or lists. In other languages, these are often referred to as simple ordered arrays and associative arrays. In *JSON*, these are termed arrays and objects, respectively. The array is an ordered, unnamed collection that is identified by the notation:

```
[ value, value, value, ... ]
```

That is, an opening [and a closing] denote the beginning and end of a comma-separated array of values. Named arrays or “objects” use { and } and have the form

```
{ "name" : value, "name" : value, ... }
```

The quotes are required for the names or “keys,” and again, the elements are comma-separated.

These containers can be nested, i.e., an element of a container can itself be a container (or a scalar). This gives *JSON* the flexibility to represent arbitrary data structures. Unlike *R*, there is no distinction between a collection of homogeneous values and nonhomogeneous values, i.e., the distinction between a *vector* and *list* in *R*. In other words, *JSON* ignores the fact that a collection contains values of the same type and just has the notion of a container, either named (the object) or ordered (the array).

An important caveat is that top-level *JSON* content cannot be a simple primitive value. Instead, it must be a container, either named or not. This means that we would never have *JSON* content of the form *1*, *true*, or `"xy" : [1, 2]`. Instead, we need to have, respectively, `[1]` or `[true]` or `{"xy" : [1, 2] }`. Note the white-space between values in *JSON* content is ignored, unless of course it is within a string.

JSON (JavaScript Object Notation)

JSON formatted data have the following properties:

- The primitive data types are boolean (*true* and *false*), number, and string. These are scalars.
- *null* stands for the empty “object”.
- The array is a simple, ordered, unnamed collection of values. These are denoted by an opening [and a closing] and the values are comma-separated, i.e.,

```
[ value, value, value, ... ]
```

- The object is a named, unordered collection values. The { and } delimit the object and the values appear as "name": value pairs. That is, an object has the form

```
{ "name" : value, "name" : value, ... }
```

The quotes are necessary for the names (keys).

- Arrays and objects can be nested, i.e., an element of a container can itself be a container (or a scalar). This allows nesting of values and gives *JSON* the flexibility to represent arbitrary data structures.

A *JSON* object can be imported into *R* as a list, and depending on its structure, it can possibly be reduced to a vector or data frame or collection of data frames.

We provide an example of *JSON* content that is available from the *New York Times*.

Example 4-6 JSON Content Available from the New York Times

The following shows data in *JSON* format, taken from *New York Times* Web service documentation for its campaign finance API [3]. These data illustrate the full set of data types and how they are represented in *JSON*.

```
{
  [1]
  "results": [ [2]
    {
      "city": "NEW YORK", [3]
      "address": "34 WEST 38TH ST - FLR 5",
      "name": "AMERICANLP",
      "zip": 10018, [4]
      "treasurer": "TJ WALKER",
      "super_pac": true,
      "relative_uri": "/committees/C00507244.json",
      "candidate": null, [5]
      "id": "C00507244",
      "leadership": false, [6]
      "sponsor_name": null,
      "party": "",
      "fec_uri": "http://query.nictusa.com/.../C00507244/",
      "state": "NY" [7]
    },
    ...
  ], [8]
  "base_uri": "http://api.nytimes.com/.../finances/2012/",
  "copyright": "Copyright (c) 2011 The New York Times Company...",
  "cycle": 2012,
  "status": "OK"
}
```

- [1] The { delimits the *JSON* object. We can think of the content of this object, in *R* terms, as a list with five named elements: *results*, *base_uri*, *copyright*, *cycle*, and *status*.

- ② The first element, `results`, is an ordered collection of individual result objects. That is, the `[` indicates that `results` is a simple ordered array.
- ③ Each element of `results` is an object/associative-array with many fields such as `city`, `address`, `name`, `zip`, etc.
- ④ The `zip` field is a number. In the original source, these were represented as strings (we modified them to display the full spectrum of primitive data types).
- ⑤ The `candidate` and `sponsor_name` fields have the special value `null`.
- ⑥ The value of each of the `leadership` and `super_pac` fields is a logical value: `true` or `false`.
- ⑦ Many of the values are strings.
- ⑧ The last four elements of this object are simple strings named `base_uri`, `copyright`, `cycle`, and `status`.

The two main operations when dealing with *JSON* in *R*—converting *JSON* content to *R* objects, and converting *R* objects to *JSON*—are handled by functions named `fromJSON()` and `toJSON()`, respectively. The `fromJSON()` function can read content from a file, *URL*, or directly from a string, i.e., in memory. It processes the bytes from this source and converts the values into *R* objects and returns a single *R* object containing the subelements. The `toJSON()` function takes an *R* data object and generates the *JSON* representation as a single string. This can then be, for example, added to a file, sent as part of an *HTTP* request, or sent to another application. These functions are available in 3 packages: `RJSONIO`, `rjsonlite` and `rjson`. In the next example, we show how to read a *JSON* file into *R* and create a data frame.

Example 4-7 A Family's Data in JSON

The data for this example family was first introduced in Chapter 1 (see Q.1-1 (page 12)). Here these data have been recast into a *JSON* formatted file. The file appears below.

```
{
  "family": [
    {"firstName": "Tom", "sex": "m", "age": 77, "height": 70,
     "weight": 175, "bmi": 25.16, "overWt": true},
    {"firstName": "May", "sex": "f", "age": 33, "height": 64,
     "weight": 125, "bmi": 21.50, "overWt": false},
    {"firstName": "Joe", "sex": "m", "age": 79, "height": 73,
     "weight": 185, "bmi": 24.46, "overWt": false},
    {"firstName": "Bob", "sex": "m", "age": 47, "height": 67,
     "weight": 156, "bmi": 24.48, "overWt": false},
    ...
  ]
}
```

Notice that this *JSON* content consists of an object with one field named `family`. The `family` field is an array of 14 objects, each object corresponds to a family member. The keys in each of these objects are the same: `firstName`, `sex`, `age`, `height`, `weight`, `bmi`, and `overWt`. The value for each key is a scalar and these scalars correspond to the measurement of that variable for the respective family member. We read this file into *R* with

```
familyList = fromJSON("family.json")
```

We confirm that we have a list with one element which itself is a list named `family`.

```
class(familyList)
```

```
[1] "list"

names(familyList)
[1] "family"

class(familyList$family)
[1] "list"
```

The `family` has 14 unnamed elements, i.e.,

```
names(familyList$family)
```

```
NULL
```

```
length(familyList$family)
[1] 14
```

Each of the 14 elements in `family` is a list. We assign the first of these to the variable `tom` and further explore this object with

```
tom = familyList$family[[1]]
class(tom)

[1] "list"

names(tom)
[1] "firstName"   "sex"        "age"         "height"
[5] "weight"      "bmi"        "overWt"
```

Furthermore, each of these 7 named objects in `tom` is a vector of length 1. For example, `tom$age` is the numeric containing the single value 77. Figure 4.13 provides a sketch of the structure of `familyList`.

We want to create a data frame from `familyList`. One approach is to extract the values of each of the vectors in the 14 lists. For example, we can extract the age of each family member by applying the `[[`-operator to each element of `family`. We do this with

```
sapply(familyList$family, "[[", "age")
[1] 77 33 79 47 27 33 67 52 59 27 55 24 46 49
```

Note that we supplied the argument `"age"` to `[[` in our call to `sapply()`. We can create a vector for each of the variables with

```
name = sapply(familyList$family, "[[", "firstName")
sex = sapply(familyList$family, "[[", "sex")
age = sapply(familyList$family, "[[", "age")
ht = sapply(familyList$family, "[[", "height")
wt = sapply(familyList$family, "[[", "weight")
bmi = sapply(familyList$family, "[[", "bmi")
owt = sapply(familyList$family, "[[", "overWt")
```

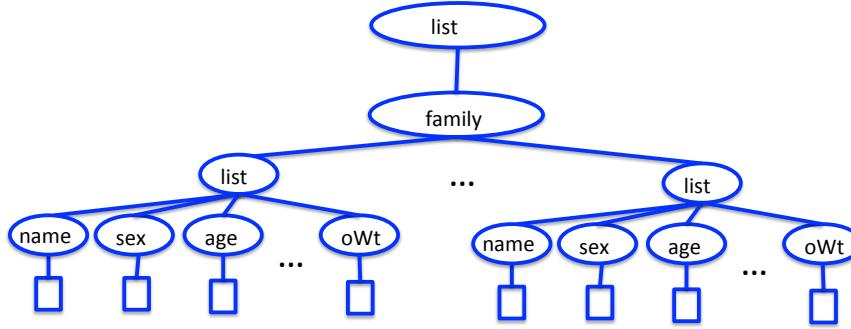


Figure 4.13: Diagram of a JSON File Read into R.

This diagram shows the structure of the list containing the information in the JSON-formatted file after being read into R. This list consists of an object called `family`. The `family` is an array of 14 fields, each of which is an object with 14 scalar fields. These scalars are `firstName`, `sex`, `age`, `height`, `weight`, `bmi`, and `overWt`. Each object and array is converted into a list in R so the data structure is a list of 14 lists, one for each family member. Each of these 14 lists is a list of 7 lists, one for each variable. Finally, each of these vectors has length 1 and contains the value for that variable for that member of the family.

We can determine the class of each of these variables by checking the class of the elements of `tom`, e.g.,

```
sapply(tom, class)
firstName      gender       age      height
"character" "character" "numeric" "numeric"
weight        bmi         overWt
"numeric"     "numeric"   "logical"
```

Note that the `true` and `false` values were converted to a logical vector. For our final data frame, we want `firstName` to remain a character vector and to convert `sex` to a factor. We combine all of these vectors into a data frame with

```
family = data.frame(name, sex = as.factor(sex), age, height = ht,
                     weight = wt, bmi, overWt = owt,
                     stringsAsFactors = FALSE)
```

Notice that we supplied the value `FALSE` for the `stringsAsFactors` argument to prevent the default conversion of character vectors into factors. This way, `name` remains a character vector. We check the data frame with

```
head(family)
  name sex age height weight   bmi overWt
1  Tom   m   77     70    175 25.16  TRUE
2  May   f   33     64    125 21.50 FALSE
3  Joe   m   79     73    185 24.46 FALSE
4  Bob   m   47     67    156 24.48 FALSE
5  Sue   f   27     64    105 18.06 FALSE
6  Liz   f   33     68    190 28.95  TRUE
```

One last comment: when we created the 7 vectors `name`, `sex`, etc., the code was repetitive. When this happens, we typically ask ourselves if there is a more concise way to carry out these computations. Indeed there is and we cover this approach in Chapter 5. However, for those who already know a bit about writing functions, we supply the alternative code as a preview.

```
varNames = c("firstName", "sex", "age", "height",
            "weight", "bmi", "overWt")
family = lapply(varNames, function(x) {
  sapply(familyList$family, "[[", x)
})
family = as.data.frame(family)
names(family) = varNames
```

Notice the nested apply functions. The outer `lapply()` applies the function to each string in `varNames`. This function is one that we have written ourselves in order to call `sapply()` with the variable name as an argument. ■

4.6 Kiva

Recall that the information about the Kiva loans is formatted in *JSON* as an array of objects (see Q.4-4 (page 153)). Each element in this array describes a loan with fields for such things as the loan identifier, the name and location of the person looking for the loan, the more specific purpose of the loan, and the amount being sought. We want to create a data structure that contains information about each loan, but it is not immediately obvious from the display of a loan field in Q.4-4 (page 153) whether or not we can map the *JSON* content into a data frame. To help us figure this out, we can read the *JSON* content into *R* and explore the structure.

We begin by downloading the *JSON* file and extracting all of the files with

```
download.file("http://s3.kiva.org/snapshots/kiva_ds_json.zip",
             destfile = "kiva_json.zip")
unzip("kiva_json.zip")
```

We can also download the file interactively in our browser or use shell commands to download it and then unzip it (see [?]). When we unzip the file we find a `loans/` directory and within it there are files `n.json` for `n` ranging from 1 to 1975.

When we open one of the files, e.g. `1.json` in our text editor to see the structure of the file, we find there is only one line in this file and it contains all the data for 500 loans. The *JSON* format has a very simple, well-defined structure, and most languages have parsers for reading *JSON*. That is, the 1st element is named header and contains fields for the total, page, date and page_size. The 2nd element contains information about the loans. Each loan has an identifier (`id`), description, a status, funded amount, and so on. The format of each loan is similar, but we do not know if all loans have all fields. Also, some fields are simple numbers or text strings, but others are collections of sub-values. We need to investigate these formats to determine how we want to organize the data in *R*.

Our first task is to read the data into *R*. As noted in Section 4.5, *R* has packages and functions to convert *JSON* content directly into *R* objects. We can read `1.json` with

```
library(RJSONIO)
loanDoc1 = fromJSON("loans/1.json")
```

Note that to read this file, we can also use either the `rjsonlite` or `rjson` packages and their `fromJSON()` functions.

The first element of `loanDoc1` is the header information, i.e.,

```
names(loanDoc1)
[1] "header" "loans"

loanDoc1$header
$total
[1] 987161

$page
[1] 1

$date
[1] "2015-12-02T18:49:18Z"

$page_size
[1] 500
```

We can access the first loan with

```
loan1 = loanDoc1$loans[[1]]
```

At this point, we have read the first file, `1.json`, into `R` and technically we have the data from this file available to analyze. However, the data are not in a convenient format to work with. If we want to compute the median loan amount, we must loop over each loan and extract the `funded_amount` and then compute the median, e.g.,

```
median(sapply(loanDoc1$loans, '[[, 'funded_amount'])
```

If we are interested in the `paid_amount`, things become slightly more complex as some loans do not have this field. For example, the 1st record does not have this field but the 2nd one does, e.g.,

```
loanDoc1$loans[[1]]$paid_amount
```

```
NULL
```

```
loanDoc1$loans[[2]]$paid_amount
```

```
[1] 500
```

As a result,

```
sapply(loanDoc1$loans, '[[, 'paid_amount')
```

returns a list rather than a vector of numbers. We have to `unlist()` this and then compute the median, e.g., `median(unlist(sapply(loanDoc1$loans, '[[, 'paid_amount'))))`.

4.6.1 Loan Elements

Let's think about how we want to structure the data. A data frame is a convenient form where each row corresponds to a loan. The variables in this data frame are `id`, `status`, `funded_amount`, etc. In order to determine the variables in this data frame, let's see what elements are in all loans and which are not. We can do this with

```
table(unlist(lapply(loanDoc1$loans, names)))
```

activity 500 bonus_credit_eligibility 500 currency_exchange_loss_amount 500 ...	basket_amount 500 borrowers 500 delinquent 500
---	---

This shows that all loans have all fields in the first file. (We need to verify this for the other files.)

Some elements are simple single values and others have list values. Let's examine the elements of the first loan and determine which elements are simple single values. We apply the `is.atomic()` function to each element in `loan1` with

```
atomic1 = sapply(loan1, is.atomic)
```

We also find the length of each element, i.e.,

```
length1 = sapply(loan1, length)
```

Scalar values are atomic and length 1. These are:

```
indexAtomic = atomic1 & (length1 == 1)
names(loan1)[indexAtomic]
```

```
[1] "id"                      "name"
[3] "status"                  "funded_amount"
[5] "activity"                "sector"
[7] "use"                     "partner_id"
[9] "loan_amount"              "lender_count"
[11] "bonus_credit_eligibility"
```

We can determine which elements have more complex values from the negation of `indexAtomic`, i.e.,

```
names(loan1)[!indexAtomic]
```

```
[1] "description"              "basket_amount"
[3] "paid_amount"               "image"
[5] "video"                     "themes"
[7] "delinquent"                "location"
[9] "posted_date"                "planned_expiration_date"
[11] "currency_exchange_loss_amount" "tags"
[13] "borrowers"                 "terms"
[15] "payments"                   "funded_date"
[17] "paid_date"                   "journal_totals"
[19] "translator"
```

The classes of these non-atomic elements are:

```
sapply(loan1[!indexAtomic], class)
```

description	basket_amount
"list"	"NULL"
paid_amount	image
"NULL"	"numeric"
video	themes
"NULL"	"NULL"
delinquent	location
"NULL"	"list"
posted_date	planned_expiration_date
"NULL"	"NULL"
currency_exchange_loss_amount	tags
"NULL"	"AsIs"
borrowers	terms
"list"	"list"
payments	funded_date
"AsIs"	"NULL"
paid_date	journal_totals
"NULL"	"numeric"
translator	
"NULL"	

Some of these elements are of no interest to us for data analysis, e.g., image and video. Several are NULL so can we omit these? All of the information here is for a single loan – the 1st. We have to see if these elements are NULL for all loans in the file, and also for all different files.

Delinquent Loans

Let's check to see if all the `delinquent` elements are NULL. We can extract the `delinquent` elements and test if they are null with

```
dels = lapply(loanDoc1$loans, '[[', 'delinquent')
nullDells = sapply(dels, is.null)
```

We make a table of the logical values in `nullDells` with

```
table(nullDells)
```

FALSE	TRUE
14	486

There are a few loans that have information in the delinquent field. Let's examine the contents of these non-NULL elements. We do this with

```
unlist(dels[!nullDells])
```

```
[1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
[11] TRUE TRUE TRUE TRUE
```

The delinquent field, when not NULL, has the value TRUE. This suggests that we store the delinquent information as a *logical* vector and map the NULL values to FALSE. We leave this as an exercise.

Note that there are more efficient way to use `sapply()` to extract the `delinquent` elements from `loanDoc1` and apply the `is.null()` all in one function call without having to save the intermediate delinquent elements (in `dels`). This approach requires us to provide an anonymous function to `sapply`. We cover this topic in Chapter 5.

Journal Totals

We can continue our investigation of these non-atomic elements to determine how to store them in a data frame. For example, the `journal_totals` field is numeric, but it consists of two values named `entries` and `bulkEntries`. One way to incorporate these into our data frame is to flatten the vector by adding each element as its own top-level variable in the data frame. That is, create two variables, `entries` and `bulkEntries` from `journal_totals`. We also leave this taks as an exercise.

Loan Themes

As another example, the `themes` field is mostly NULL but there are a few character vectors. We confirm this with

```
themes = lapply(loanDoc1$loans, '[[]', "themes")
table(sapply(themes, class))

character      NULL
    28        472
```

We can look at the length of these character vectors with

```
table(sapply(themes, length))

 0   1   2   3
472 20   7   1
```

We see that there are a few loans that have multiple themes. We can combine these multiple themes into a single string to put the themes into a single column. Alternatively, we can create a separate data frame and use the loan identifier to map between the loan and the theme, e.g.,

```
loan_id theme
 1      Underfunded Areas
 1      Rural Exclusion
 97     Conflict Zones
124     Rural Exclusion
```

How we decide to represent these data (if at all) depends on how we plan to use them and which representation makes the computations more convenient overall.

Translator

The `translator` field is sometimes NULL but is typically a list, and in some cases it is a character string:

```
translators = lapply(loanDoc1$loans, '[[]', 'translator')
table(sapply(translators, class))

character      list      NULL
      5        365       130
```

The list elements have a `byline` and an `image` identifier field. The `byline` is the name of the person. The character elements in `translator` give the name of the translator, i.e., the `byline` field. We can collapse `translator` to the `byline` field as a character string and drop the `image`. We can use NA for those that are NULL. However, do we really need the translator information for a loan? We may want to drop this information all together.

4.6.2 The Payments

The payments field is complicated because it contains information about each installment. We can combine these records into their own data frame for each loan. However, where do we put this data frame in the loans data frame? We can have a payments column each of whose elements is itself a 7 column data frame. This can work, but it is slightly awkward to work with. If we want to find the number of payments made for each loan, we have to loop over each loan and compute the number of rows. To work with the payment amounts or dates, we have to do slightly more. An alternative approach puts all of the payment information for all of the loans into a single data frame that is separate from the loan data frame. That is, each row in this data frame corresponds to a payment. In this case, some loans will have multiple rows and others may not have any. If we take this approach, then we must add the loan identifier (`id`) as an additional column so that we can connect the payment records with the loans records.

Another approach is to use a payment as the entity corresponding to each row in our loans data frame. We then have to repeat all of the fixed information about the loan for each row (e.g. `id`, name, description, status, ...). This leads to quite a lot of data repetition. Importantly, it makes the typical operations we do on loans more complicated. For example, to compute the distribution of loan amounts with this structure, we have to discard all the duplicate loan amounts for the same loan. This is a cumbersome approach. The solution of a separate payment data frame is preferable.

4.6.3 The Borrowers

The `borrowers` field is another `list` that appears in each loan element. This field identifies each of the borrowers for the loan. There are a different number of borrowers for each loan so this is a variable length list. We confirm this with

```
borrowers = lapply(loanDoc1$loans, '[[, "borrowers"]')
table(sapply(borrowers, length))
```

1	2	3	4	5	6	7	8	9	10	11	12	13	14
419	3	13	10	15	1	4	2	5	5	3	3	5	4
15	16	19	20	26									
4	1	1	1	1									

Most loans (419 of the 500) are made to one person. However 81 loans have multiple lendees, including one loan with as many as 26 borrowers.

In our analysis, we may want to see if there are any people who received multiple loans, and whether they paid them back. We also may want to see if the number of lendees indicates whether the loan is paid back and whether different loan uses are associated with more lendees. Therefore, it makes sense to map the lendees to their own table, and use the loan identifier to link between the loan data frame and the lendees table. Additionally, we may want to create a new variable—the number of borrowers for a loan—and add it to our data frame.

4.6.4 Final Structure

Given our explorations and ideas for analysis so far, we want to create three data frames – one for loans, one for payment installments, and one for borrowers. The loans data frame is the primary one. There is one observation for each loan, and all of the atomic elements are included as variables in the data frame. Additionally, we may want to flatten some complex elements, such as `journal_totals` and `translator` to create additional variables in the data frame and derive new variables to include, such as the number of borrowers on a loan. We also need to systematically examine the non-atomic length-one entries. For example, we have seen that with the `delinquent` field we want to create a logical vector where the NULL entries have values of FALSE.

The other two data frames are at different levels of granularity than the `loans` data frame. The `payments` data frame has one row for each loan payment so some loans may not appear in the data frame because no payments have been made, and other loans may have many rows, one for each payment made on the loan. Similarly, the `borrowers` data frame has a row for each borrower of a loan. Most loans have one borrower and consequently one row in the data frame. Those loans with multiple borrowers have a row for each person. In both the `payments` and `borrowers` data frames, we want to include the `id` for the corresponding loan so this information can be linked to the loan information. We create these three data frames in the exercises.

4.7 Summary

4.8 Functions for Handling Complex Data Formats

This chapter introduced many of the functions available in *R* for working with lists and data that do not have simple table-like formats. These are summarized below.

`readHTMLTable()` (in `XML` package) Return all the tables in the *HTML* document as data frames. The `which` parameter allows us to specify those tables we want to extract from the document. The `colClasses` parameter allows us to specify the classes/types for the columns or a function to convert the content to numbers, percentages, factors, etc.

`str()` Compactly display the structure of an arbitrary *R* object. The display includes the class of the object, at minimum, and may include information such as the object's dimensions, initial elements, or arguments.

`getURL()` (in `RCurl` package) Retrieve the content of a URL. Note the `getURLContent()` function is a more general function that handles binary or text results.

`fromJSON()` (in `RJSONIO` package) Parse *JSON* content into a list, and convert the content into the corresponding *R* type.

`toJSON()` (in `RJSONIO` package) Convert a vector or list to a *JSON* object.

`unlist()` Reduce the supplied list to a vector by concatenating elements in that list. If `recursive` is TRUE (default), the reduction is recursively applied.

`is.atomic()` Return TRUE if the supplied argument is a vector, specifically an atomic vector type (logical, integer, numeric, complex, character, or raw).

list.files() Return a character vector with all directories and files in the supplied directory.

do.call() Construct and execute a function call by passing a list of arguments (`args`) to a function (`what`). The function can be specified as a string.

rbind() Stack a sequence of vectors, matrices, or data frames by rows (on top of each other).

The return value is a matrix or data frame with row length equal to the length of the sequence (vectors) or sum of the row lengths of each element in the sequence (matrices and data frames). The inputs must have the same lengths (when stacking vectors) or columns (matrices and data frames).

readLines() Read in a text file from a connection (e.g., a file name or *URL*) and return a character vector with the same length as the number of lines in the text file. To limit the number of lines to be read, use the `n` argument.

The Family of Apply Functions

These functions are variants on the concept of applying a specified function to each element of a list or data frame. The first argument in each of these apply functions is the object on which to operate.

lapply() Apply supplied function to each element in the data frame or list and return the result as a list with the same length as the original data frame or list. The input can also be a vector, in which case the function is applied to each element in the vector.

sapply() Similar to `lapply()` but the result is simplified to a vector or matrix, when possible.

mapply() Multivariate version of `sapply()`. Apply the supplied function and vectorize over multiple arguments. In this case, the function is provided as the first argument to `map-`
`ply()` and the multiple inputs follow. Additional arguments can be specified as a list in `MoreArgs`.

apply() Apply supplied function to the specified margins of the data frame, matrix, or array.

For example, for matrices, `MARGIN == 1` indicates rows (apply the function across columns for each row) and `MARGIN == 2` indicates columns (apply down columns). As another example, for a three-dimensional array, `MARGIN == c(1, 2)` indicates rows and columns (across pages).

tapply() Apply function to each sub-group of the data vector (first argument). The sub-groups are formed from the unique values of the vector in `INDEX`. A list of vectors can be supplied in `INDEX`; these vectors should have the same length as the data vector.

4.9 Guided Practice

Subsetting lists

This section provides practice on subsetting lists. There are many ways to subset (see Section 4.2.1 for a review). Double square brackets are used to go one level deeper into a list structure and access individual elements of the list. A named element of the list can also be accessed with the \$-notation. Since data frames are special kinds of lists, both double square brackets and the \$-notation work for data frames.

The following set of questions concern the list `myList`. The list is loaded by `load("exampleList.rda")`.

1. Examine `myList`. How many elements are in the list? What are the names and classes of the variables in the list?
2. Create a new vector `check` that contains the elements in the even positions of `chars`. Write the code generally by not using the fact that you know how many elements are in `chars`.
3. The list `myList` contains a function and a data frame. Call this function, passing in the variable `x` from the data frame.

Apply functions

This section provides practice with the family of `apply()` functions, which include `apply()`, `lapply()`, `sapply()`, `tapply()`, and `mapply()`.

In answering the following questions, you will work with a subset of the rainfall data from five weather stations in the Colorado Front Range (see Q.4-1 (page 150) and Section 4.2.3). The data are loaded with `load("rainfallCO.rda")`. The data are in two lists, `rain` and `day`. Each list has five elements; one for each weather station. The goal is to summarize the data and create *R* objects useful for analysis, e.g., determining the average daily rainfall.

1. Examine the two lists, `rain` and `day`. Verify that they each have five elements. What are the names, classes, and dimensions of the elements in the lists?
2. Since `rain` provides rainfall data on the corresponding dates in `day` for each weather station, make sure the station names and dimensions match between the two lists. Use the `sapply()` function in your answer.
3. To help calculate average daily rainfall per *year*, create a new list called `year` that extracts the year from the data in the `day` list. Then, use `year` to find the total number of years that each station was in operation. *Hint:* Use the `floor()` and `unique()` functions in your answer. Why is the function `floor()` necessary?
4. Calculate the average rainfall per year for the third station in the dataset. Use the `tapply()` function in your answer. *Hint:* With the `tapply()` function, the function provided in the `FUN` argument is applied to subsets of the data (`X` argument) where the subsets are determined by the vector supplied in `INDEX` argument.
5. Create a new list `avgRainList` that contains the average daily rainfall per year for each station. Each element in the new list should correspond to one weather station. Use the `mapply()` (multivariate apply) function in your answer. *Hint:* The technique from the last question is useful here.

4.10 Exercises

Bibliography

- [1] David Flanagan. *JavaScript: The Definitive Guide*. O'Reilly Media, Inc., Sebastopol, CA, 2006.
- [2] Kiva Organization. Kiva: Loans that change lives. <http://www.kiva.org/>, 2011.

- [3] The *New York Times* Company. The Times Developer Network: An API clearinghouse and community. http://developer.nytimes.com/docs/campaign_finance_api/campaign_finance_api_examples, 2012.
- [4] Wikipedia. 1500 metres world record progression. https://en.wikipedia.org/wiki/1500_metres_world_record_progression, 2015.

Part II

Programming, Monte Carlo, and Resampling

5

Programming Concepts

CONTENTS

5.1	Introduction	193
5.2	Steps in Writing Functions	196
5.2.1	Calculating BMI from Height and Weight	196
5.2.2	Generalizing Code	198
5.2.3	Commenting Code	198
5.2.4	Reporting Run Times in Seconds	199
5.2.5	Taking the Logarithm of ‘0’	201
5.3	Defining a Function	202
5.3.1	Anonymous Functions	204
5.4	Calling a Function	205
5.4.1	Argument Matching by Name and Position	205
5.4.2	The ... parameter	207
5.4.3	Lazy Evaluation	207
5.4.4	The Search Path	209
5.4.5	Partial Matching	211
5.5	Exiting a Function	211
5.6	Conditionally Invoking Code	212
5.6.1	Conditionally Modifying Inputs to <code>log(0)</code>	214
5.6.2	Converting Liquid Measures into Tbs. Using <code>switch()</code> ..	219
5.7	Iterative Evaluation of Code	221
5.7.1	Reading Thousands of Files of Data With a <code>for</code> Loop ..	221
5.7.2	Generating Fibonacci Numbers	223
5.7.3	Playing Penney’s Game	228
5.8	Style Guidelines	233
5.9	Beginning Notions of Debugging	236
5.10	Summary	240
5.11	Control Flow and Debugging Functions	240
5.12	Guided Practice	241
5.13	Exercises	243
	Bibliography	244

5.1 Introduction

The code we have written so far has exclusively used functions provided in base *R* and in packages contributed by others. However, there are often situations where it can be advantageous to write our own functions. For example, we can imagine wanting to reuse some code, and rather than making copies of the code, we can create a function that performs these computations. We simply call our function when we need to carry out the

computations. This approach is less error prone because we typically have less to type when we call the function than when we repeat several lines of code. It is also less error prone than copy-and-pasting lines of code. Additionally, we might want our function to handle variations in the code. To do this, we can design our function to accept arguments that control how the code is evaluated. Also, corrections and updates to our code can be made in one place (in the function), and these changes are available to all function calls. Moreover, a function typically encapsulates a task and if we give it a name that reflects this task, then our function call helps makes it clear to us and others what our code is doing. Furthermore, when we have a large task to perform, we can more easily understand our work, test our code, and track progress on the project if we divide the large task into smaller tasks that we encapsulate in separate functions.

In this chapter, we introduce the basics of writing functions. The functions that we create here perform simple tasks, such as those described in the following examples. Later, we demonstrate how to take on larger tasks and how to organize our work into multiple functions, each carrying out a separate subtask. For example, in Chapter 6, we develop a resampling technique to select a model for prediction, in Chapter 7, we design and carry out a simulation study of a complex random process, and in Chapter 14, we develop a programmatic approach to read and merge data from different sources.

Example 5-1 Calculating BMI

In Q.1-1 (page 12), we worked with artificial data from a 14-member family. These data are similar to those collected by the Centers for Disease Control (CDC) in the Behavioral Risk Factor Surveillance Survey (BRFSS). The survey respondents provide their height (inches), weight (pounds) and desired weight (pounds). Their Body Mass Index (BMI) is calculated and provided as part of the BRFSS data set. Suppose in our analysis, we are interested in the relationship between a respondent's BMI and 'desired' BMI (calculated from desired weight). To study this relationship, we can create a function for computing BMI from height and weight (or desired weight in this case). We take the trouble to write a function to do this simple calculation because we anticipate needing to compute BMI for other data and possibly for units other than pounds and inches.

Example 5-2 Tally Run Times in Seconds

In Q.4-2 (page 151) we extracted run times for the men's 1500 meter race from a table on a Wikipedia page. In that process, we made a simple calculation to convert the times into seconds to assist us in making plots. There are several other tables of world records on that page, including those for women and pre-IAAF (International Association of Athletics Federations) records, and if we have a function to convert times reported in minutes and seconds into seconds, then we can use our function with the times in these tables. Even though this calculation is simple, there are several good reasons to create a function for this task. We can hide the code that converts numeric times to the POSIX format in the function. If we find that we made a mistake in our code, then we can correct it in one place, i.e., in the function, and rerun our code to process the data again. Otherwise, we need to correct the mistake in multiple places, and we run the danger of making more mistakes. Additionally, the pre-IAAF times are formatted differently so we can design our function to handle these variations in format. Finally, we can later generalize our function to convert times for other races, e.g., marathons with times that include hours as well as minutes and seconds.

Example 5-3 Taking the Logarithm of 0

The log transformation is very useful in data analysis. For example, we often log-transform

data for variance stabilization so that the data distribution is less skewed and more symmetric. Also when plotting data, we often use a log transformation to help us better understand relationships between variables (see Section 3.3.1). In many situations, the data include a few 0 values and produce errors when we take a logarithm. A common solution is to add a small positive value to all of the data. That is, we essentially compute $\log(x + c)$, where c is some positive constant. Popular values for c are 1, a small value such as 0.001, or a value that depends on the minimum data value, e.g., $\min(x)/100$ (where the minimum is over the positive x values). We can imagine having our own ‘log’ function can be useful. It can give us versatility in specifying the constant added to the data.

Example 5-4 Converting Liquid Measures into Tablespoons

While we primarily use *R* for data analysis and simulation studies, we sometimes use it as a calculator. For example, we can use *R* to convert liquid measures in metric units into, say, tablespoons. If we find that we often need to make these conversions, then we can write a function for this purpose where we specify the amount and units of the liquid and the function provides the equivalent number of tablespoons.

Example 5-5 Reading Thousands of Data Files

We often begin a data analysis by reading data stored in a file into *R* with one of the functions described in Chapter 2 and Chapter 4, such as `read_delim()`, `readLines()`, and `fromJSON()`. At times the data are arranged in many files, e.g., the Kiva data in Q.4-4 (page 153) are provided in 1975 files. When this happens, we need to read from all of these files and organize the results into a structure for analysis. To determine the file names, we can use a file finder, such as Spotlight, and add these to our code. However, when we have thousands of files and so thousands of file names, then using a file finder is not feasible. We want to automate this process and avoid typing or copy-and-pasting thousands file names. We can write a function for this purpose.

Example 5-6 Generating Fibonacci Numbers

The Fibonacci numbers are generated by setting $F_0 = 0$, $F_1 = 1$, and then using the recursive formula below for finding subsequent numbers:

$$F_n = F_{n-1} + F_{n-2}$$

The first 10 Fibonacci numbers are: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34. For fun, we can write a function to compute F_n , for any n . Given the recursive nature of the definition, i.e., the n th number is defined in terms of the $n - 1$ st and $n - 2$ nd numbers, we need to build up F_n from earlier values in the sequence. We see later that there are several ways to do this, and we can carry out a computer experiment to compare the efficiency of these approaches.

Example 5-7 Playing Penney’s Game

A simple version of Walter Penney’s game [3] has two players each choose a sequence of 3 heads and tails. That is, player A selects a sequence of 3 heads and tails, shows this sequence to player B, and then, player B selects his/her sequence of 3 heads and tails. After that a (fair) coin is tossed until either player A’s or player B’s sequence appears. Whoever has the sequence that appears first wins. While, we can work out analytically player B’s strategy for choosing a sequence given player A’s sequence, we can also gain intuition by using the computer to carry out a simulation study of the game. To do this, we can create a function

to play one round of the game for a given pair of strategies and then play the game many, many times for various strategies and compare the number of wins.

We use these examples in this chapter to demonstrate how to write functions, including issues of specification of formal arguments, control flow, return values, efficiency, and programming style guide lines.

5.2 Steps in Writing Functions

Many of the examples in Section 5.1 can be addressed with one or two lines of code. It may seem as though there is little need to write a function to carry out these simple tasks, but they offer excellent examples of how to go about writing a function, and often they can be generalized to handle more complex situations. Moreover, it is typically the case that when we have a larger problem to solve, we break the work up into smaller tasks which are carried out by functions that contain only a few lines of code.

We begin with the problem of calculating BMI from Q.5-1 (page 194) to demonstrate the process of writing a function. Briefly, these steps are: express the task in words; write code to carry out this task for a specific example; abstract the code by identifying inputs and constants and using general names and, where appropriate, default values for these; define the function and incorporate the generic code into the body of the function; and test the function with the original special case and additional test cases. Of course, as we gain expertise in function writing, we can often collapse some of these steps; we often do not need to write a concrete version of the code and instead begin by identifying the function inputs and writing a generic version of the code.

Steps in Writing Simple Functions

Explain Begin by describing in words (and possibly formulas) what you want the function to do.

Code Write code to address a specific example of the task.

Abstract Identify the inputs in your code that you want the caller to specify. Rewrite your code, if needed, to use generic variable names rather than specific quantities/variables for these inputs.

Encapsulate Wrap this generalized version of the code into a function with the inputs as formal arguments. Make sure that your function returns the desired object.

Test Check the function works as expected with your original data. Try the function on test cases that you create to check various aspects of the function.

5.2.1 Calculating BMI from Height and Weight

Explain the Task

We begin by clearly stating the task that we want the function to carry out. In our example, we want the function to compute BMI from height and weight. We can look up the formula

on the Web to find:

$$BMI = \frac{703 * weight}{height^2},$$

for weight measured in pounds and height in inches.

Code the Task

Next, we write code to carry out the task. In this code, we use data from a particular example in order to make the task concrete. For example, we can use our family data from Q.1-1 (page 12). Recall that we have the two vectors `fheight` and `fdesiredWt`. We translate the formula into a single line of code that uses these two variables with

```
703 * fdesiredWt / fheight^2
a      b      c    ...
25.10714 19.73755 23.08594
```

It appears that our code is correct; there are no syntax errors and the results have reasonable values.

Abstract the Code

The code that we have written works for a particular set of data, i.e., for `fheight` and `fdesiredWt`. We want to abstract the code to use general variables for height and weight, such as `ht` and `wt`. These variables will be the inputs to our function. Our generalized code appears as

```
703 * wt / ht^2
```

If we try to evaluate this code at the *R* prompt, we get an error because these variables are not defined in our workspace.

Encapsulate Code in a Function

We are ready to create a function, and we begin by assigning the function to a name. We do this just like we do for any assignment, i.e.,

```
calcBMI =
function(ht, wt)
{
}
```

We have created a function that takes two arguments (`ht` and `wt`) and assigned it to the variable `calcBMI`. Currently, our function doesn't do anything. We need to place the code we wrote earlier (the generic version) between the curly braces in the function definition. We do this with

```
calcBMI =
function(ht, wt)
{
  703 * wt / ht^2
}
```

We have written a function!

Although this function is very short, we typically do not define a function at the prompt. This approach can be frustrating when we make small typographical errors. Instead, we write our function in a text file (e.g., an *R* script in RStudio) and source the function into *R* with the `source()` function (or click on the `source` button in the RStudio menu bar).

Other approaches, include writing code in a code chunk in an *Rmd* file and running the chunk, or writing code in a text editor such as Emacs, which has an interface to *R*.

Test the Function

After we source our `calcBMI()` function into *R*, the next step is to test the function. We can, for example, compare the values in `fbmi` to the return value from

```
head(calcBMI(fheight, fweight))
      a      b      c      d      e      f
25.10714 21.45386 24.40514 24.43039 18.02124 28.88625

head(fbmi)
[1] 25.16239 21.50106 24.45884 24.48414 18.06089 28.94981
```

The values are close but not exact. This is not too surprising as there is rounding error in our use of 703 for the constant and we don't know if `fheight` and `fweight` were used in computing `fbmi` or if more precise versions of these measurements were used. Nonetheless, `fbmi` and our calculated BMI values should be close, and we can compare these two sets of values to see if they are within, say, 0.1 of each other with

```
all(abs(fbmi - calcBMI(fweight, fheight)) < 0.1 )
[1] TRUE
```

Indeed, all of our calculated BMI values are within 0.1 of the reported values in `fbmi`.

5.2.2 Generalizing Code

The process of writing a function is often iterative. After our first attempt at writing our function, we typically make the function more robust and general. That is, we iteratively refine it by testing our code with different inputs, refining code to handle bad inputs, generalizing code to handle a greater variability of inputs, and streamlining our code so it is clearer and more efficient. We repeat this process until we are satisfied with the function.

For instance, it is good to avoid hard coded constants and to make them variables either within the function or formal arguments so that the caller can specify them. As an example, notice the constant 703 in our `calcBMI()` function. This constant accounts for the units of measurement, i.e., that height is measured in inches and weight in pounds. If our measurements are in other units, then the constant needs to change, e.g., when height is in meters and weight in kilograms, then the constant is 1. We can generalize our function to have a formal argument for this constant. It makes sense for it to have a default value, and in this case we choose 703 for the default because we expect our measurements to be in pounds and inches. Our revised function appears below:

```
calcBMI =
function(ht, wt, unitConstant = 703)
{
  unitConstant * wt / ht^2
}
```

Notice that we make the new argument, `unitConstant`, the 3rd formal argument in our function. If we have already written code that uses this function, then the placement of the new argument and its default value of 703 means that any existing code which calls this function continues to work as expected. Further, we typically place parameters that have a default value after those that are required in the function's signature.

5.2.3 Commenting Code

Another important part of function writing is documentation. We can add comments to our code by preceding text with the `#` character. All text after the `#` to the end of the line is ignored by *R*. It is there for people to read. We typically add informative remarks to remind ourselves and others of the purpose of command(s) and the expected inputs and output. Our `calcBMI()` function is quite simple, except the use of the constant to change units may be a bit unclear so we add a comment explaining this:

```
calcBMI =
function(ht, wt, unitConstant = 703)
{
# use unitConstant = 1 for wt in kg and ht in m
# for others: convert to kg and m to find constant,
# e.g., for lb and in: 0.4536 kg/lb / (0.0254 m/in)^2 is 703

  unitConstant * wt / ht^2
}
```

We close this section on how to write functions with 2 more examples. We consider the tasks of converting race times into seconds (see Q.5-2 (page 194)) and developing our own logarithm function that adds a small positive quantity before taking the logarithm (see Q.5-3 (page 194)). In each case, we explicitly go through the steps to writing a function, i.e., explain, code a concrete case, abstract, encapsulate, and test.

5.2.4 Reporting Run Times in Seconds

Explain

Our task is to convert a character vector containing times into a numeric vector of times measured in seconds, e.g., "3:55.8" is converted to 235.8.

Code

Let's set up some sample data for developing our function. We can take the first few values from the 1500m world records in Section 4.4.1, i.e.,

```
testTimes = c("3:55.8", "3:54.7", "3:52.6", "3:51.0")
```

We can start with the code we developed in that section. That is, we use one of the `POSIX()` functions in *R* to convert the string into a date-time object with

```
tempTime = as.POSIXlt(testTimes, format = "%M:%OS")
```

Then we convert these into numeric seconds with

```
tempTime$min * 60 + tempTime$sec
[1] 235.8 234.7 232.6 231.0
```

We now have completed the task for a specific set of values (those in `tempTimes`).

Abstract

If we are to convert this code into a function, we need the caller to provide the string of times. Let's call it `timeString` to make it clear that the input are strings. Do we need to specify any other formal arguments? As mentioned earlier, if we want to generalize this

function, then we may want the caller to supply the format of the string. It makes sense to provide a default format; we can use a value that works for our Wikipedia tables (e.g., "%M:%OS"). Let's name this argument *format*.

Encapsulate

We have settled on the parameters for the function so we can define our function with, e.g.,

```
timeInSec =
function(timeString, format = "%M:%OS")
{
}
```

Notice that we placed *timeString* first because it is the main argument to the function and it has no default value. We fill in the function body with the code above, first swapping the parameter names for the variables and constants in that code. This leads to the function below:

```
timeInSec =
function(timeString, format = "%M:%OS")
{
  tempTime = as.POSIXlt(timeString, format = format)
  tempTime$min * 60 + tempTime$sec
}
```

Notice that the *format* argument in our function is supplied by the caller, and we pass it into *as.POSIXlt()* as the value for this function's *format* argument.

Test

We test our function with

```
timeInSec(testTimes)
[1] 235.8 234.7 232.6 231.0
```

These values match those previously calculated. We leave to the exercises the task of developing additional tests for our code.

We used the *format* argument to generalize our function. Can you see any potential problems with it? If we look at the pre-IAAF times, we see that they appear as, e.g., "4:16 4/5". Can the *as.POSIXlt()* function handle this format? Also, what if the times are for a marathon? There is a Wikipedia table for marathon records and these times appear as, e.g., "2:27:49.0". Does our function work properly with this format? We leave the detailed answers to these questions to the exercises.

Deciding on the Arguments to a Function

We consider the following questions when determining the formal arguments to a function:

- What data/information must the caller provide? Create arguments for these data.
- Is there a reasonable default for an argument? If so, then provide it.
- Are any of the arguments required? Typically, we place required arguments before those with default values.
- Is a variable computed from the inputs in the first few lines of code in the function?

If so, consider adding this variable to the formal arguments with the expression as a default value.

- Are there constants in the code? If yes, give them names and place near the top of the function. Consider whether it generalizes the function if they are arguments.

5.2.5 Taking the Logarithm of ‘0’

Explain

Our task is to take the logarithm of a numeric vector, except that our ‘log’ function adds a small positive value to all of the elements. That is, we compute $\log(x+c)$, for some constant c . Often this constant is 1, but if the other values in the vector are small then the constant is typically a smaller value, such as 0.001. Also, this constant may depend on the data, such as 1/10 of the smallest positive value in the data.

Code

We make up some data for developing our function. Since our explanation noted a few different constants, let’s create a few vectors for use with these different constants,

```
x = 0:3
y = c(0.1, 0, 1, 0.2)
z = c(0.0005, 0.1, 1, 0)
```

Our code is very simple, we want only to compute the log of these vectors (with the addition of a small constant), e.g.,

```
log(x + 1)
[1] 0.00 0.69 1.10 1.39

log(y + 0.001)
[1] -2.293 -6.908  0.001 -1.604

log(z + min(z[z > 0])/10)
[1] -7.50559 -2.30209  0.00005 -9.90349
```

Abstract

To convert this code into a function, we need the caller to provide the numeric vector, which we simply call `x`. Also, we have tried our approach for 3 different constants so it makes sense to make the constant a parameter for the function too. We call it `amt` rather than `c` because it’s good programming practice to avoid using common function names for variable names. It can lead to unexpected results! It’s also good practice to use names that make clear the purpose of the parameter.

Encapsulate

We define our function with, e.g.,

```
logPlus =
function(x, amt = 1)
{}
```

As in earlier examples, we place `x` first because it is the main argument to the function and has no default value. We use 1 for the default value of `amt`, but it can be reasonable to use another default value. For example, if we want the default to depend on `x`, then our function definition appears as, e.g.,

```
logPlus =
function(x, amt = min(1, min(x[x > 0]) / 10))
{}
```

In this function signature, we set the default value of `amt` as the smaller of 1 and 1/10th the minimum of the positive data values. Notice that we have used the input from another parameter (`x`) to compute the default value of `amt`. We might be inclined to think this would cause an error, but it does not. See Section 5.4 for details on how functions are invoked in *R*, i.e., how the parameter values are used to create variables available within the function's workspace. We complete our function by simply adding one line of code to the body of the function, i.e.,

```
logPlus =
function(x, amt = 1)
{
  log(x + amt)
}
```

Test

We test our function and compare the results to our earlier trials with

```
logPlus(x)
[1] 0.00 0.69 1.10 1.39
```

Recall that the `x` used in the function call above is a variable in our global environment, and it is different from the `x` in the body of our function. We discuss this in greater detail in Section 5.4. Also, the return value from `logPlus(x)` matches the previous calculation, as do the calls: `logPlus(y, 0.001)` and `logPlus(z, min(z[z > 0])/10)`.

5.3 Defining a Function

We have seen several examples of function definitions in Section 5.2. More formally, the syntax for defining a function is:

```
functionName =
function(formal arguments)
{
  statements
}
```

The keyword `function` tells *R* that we want to create a function, and the formal arguments (aka parameters) to the function are supplied in the parenthesis as a comma-separated list of names. If a parameter has a default value, then it is supplied in this list. For example, with

`function(ht, wt, unitConstant = 703)`, we have specified 3 formal arguments `ht`, `wt`, and `unitConstant` and assigned `unitConstant` a default value of 703. We assign the function to a name in order to call it, e.g., we call `calcBMI()` with `calcBMI(fheight, fweight)`.

We have adopted the convention of placing the `function` keyword and the opening curly brace on their own lines. This is not required by *R* and others use different conventions. For example, the following syntax is equally valid:

```
functionName = function(formal arguments) {  
  statements  
}
```

Also, if our function contains only one line of code, we can drop the curly braces entirely and define our function as

```
functionName = function(formal arguments) one statement
```

We discuss additional style guidelines for coding in Section 5.8.

Defining a Function

The general format for a function is:

```
myFunc =  
function(arg1, arg2, arg3 = 17)  
{  
  body  
}
```

- The keyword `function` tells *R* that we want to create a function.
- The formal arguments (i.e., parameters) of the function are provided between the parentheses as a comma-separated list of parameter names. If a parameter has a default value, then it is supplied here. In this example, there are 3 parameters, called `arg1`, `arg2` and `arg3`; the first 2 parameters are required and the 3rd has a default value.
- We assign the function to a variable, in this case `myFunc` so we can use this name when we call the function. We need not assign the function a name, in which case it is an anonymous function. Anonymous functions are useful when they are supplied as inputs to other functions, and we do not expect to use them in other settings.
- The body of the function appears between the opening and closing curly braces. The body consists of the expressions needed to carry out the function's task. The function returns only one object when it completes execution.

A very common style of writing and managing functions is to create them in a text or script file. Then, when we want to test them, we use the `source()` function (or the Source button in the RStudio environment) to read and evaluate the commands in that file. This call to `source()` defines the functions as regular variables in our workspace that are ready for us to use and check. If we need to modify the function to fix a bug or make it more general, then we edit the code in the file and re-source the file. We can define many functions in

one file and we can include code that is not part of a function. For example, we can set up tests which are placed in the file after the function definitions; then, when we source in the file, the functions are re-defined and after that our tests are run. There are other ways to source our functions into the global environment, including using code chunks in *Rmd* files and text editors such as Emas that interface to *R*.

5.3.1 Anonymous Functions

We saw in Section 5.2 that the keyword `function` in the function definition tells *R* that we want to create a function, and we assign that function to a variable in order to call it. That is, we use the regular assignment operator to give our function a name. However, we do not have to assign the function to a variable. In this case, we have an anonymous function. Typically we use anonymous functions when the function is used only in one particular context. We give an example with applying a function over the elements of a list.

Example 5-8 Counting Years of Operation for Weather Stations

In Section 4.2.3, we explored data on rainfall in the Colorado Front Range. The data are organized in a list, `FrontRangeWeather`, that includes a list called `days` of numeric vectors of dates of the recorded precipitation for each weather station. The weather stations are in operation for varying amounts of time. Suppose that we are interested in the number of years that each weather station is in operation. Note, if a station is in operation for 10 days in, say, 1967 then we count that as 1 year.

The weather station data is in the file `FrontRangeWeather.rda`. After we load the `rda` file into our *R* session, we can find the years of operation for, say, the 18th weather station in the list with, e.g.,

```
length(unique(floor(FrontRangeWeather$days[[18]])))  
[1] 8
```

To find the number of years of operation for each of the 56 weather stations, we can use `sapply()`. However, unlike earlier applications of `sapply()` and `lapply()`, there is no single function in *R* to apply to each vector in `days` so we supply our own function. Our function needs to compute the length of the unique values in a vector of dates after these dates have been reduced to integers. Our function for this task is

```
countYears =  
function(x)  
{  
  length(unique(floor(x)))  
}
```

Here `countYears()` encapsulates the code for the 3 nested function calls we performed earlier for the 18th weather station. We apply `countYears()` to each element of `days` in `FrontRangeWeather` with

```
sapply(FrontRangeWeather$days, countYears)
```

The return value is a numeric vector with 56 values.

Since we don't imagine using this function in other situations, we don't need to name it. Instead, we can pass an anonymous function into `sapply()` with

```
sapply(FrontRangeWeather$days,  
       function(x) length(unique(floor(x))))
```

```
st050183 st050263 st050712 st050843 st050945 ...
 53      34      20      54      19
```

Notice that we have dropped the curly braces in our function definition because our function contains only one expression. Typically, anonymous functions are quite short.

Finally, we point out that we can carry out these 3 operations in sequence and so avoid creating a function. To do this, we use intermediate variables to store the results, e.g.,

```
years = sapply(FrontRangeWeather$days, floor)
uniqueYrs = sapply(years, unique)
sapply(uniqueYrs, length)
```

This approach is neither as clear or efficient as the approach that uses anonymous functions. It is less efficient because we have called `sapply()` 3 times and created extra copies of the `days` data, and it is less clear because this simple operation is split into 3 separate steps. However, this may be a natural approach to take when first developing our code. ■

5.4 Calling a Function

5.4.1 Argument Matching by Name and Position

When we call a function, *R* maps the inputs in the function call to the formal argument names in the function's signature and puts them into a call frame with variables corresponding to these formal argument names. One can think of this process in the following way. *R* first creates a list (or table) of variables with one variable name for each of the formal arguments, and default values are provided as values for each formal argument that has one. After that, *R* maps the inputs specified in the function call to these variables in the call frame and updates their values. *R* first matches arguments by name and then by position.

An example will help concretize this process. In our `calcBMI()` example, our function signature is

```
function (ht, wt, unitConstant = 703)
```

When we call `calcBMI()`, *R* creates a call frame with variables named `ht`, `wt`, and `unitConstant`. *R* then assigns the value 703 to `unitConstant`. At this point, *R* then maps the inputs specified in the function call to the variables in the call frame. Suppose we call `calcBMI()` with:

```
calcBMI(1.4, 44.2, unitConstant = 1)
```

R first matches named arguments. In this case, *R* recognizes the name `unitConstant` and assigns the value 1 to `unitConstant` in the call frame table. Now, since there are no other named arguments in the function call, *R* matches arguments by position. The value 1.4 is assigned to the first unmatched formal argument, which is `ht`. The next step is to map 44.2 to the next unmatched formal argument, which is `wt`. These steps for creating the call frame are displayed in Table 5.1. Now that the call frame is set up, *R* begins executing the code in the body of the function.

As another example, consider the following call to `matrix()`,

TABLE 5.1: Call Frame Set Up:

Signature: calcBMI = function(ht, wt, unitConstant = 703)

Call: calcBMI(1.4, 44.2, unitConstant = 1)

Initial Variable	Set Up Value	Match Variable	Name Value	Match Variable	Position Value
ht		ht		ht	1.4
wt		wt		wt	44.2
unitConstant	703	unitConstant	1	unitConstant	1

```
matrix(, 2, 3)
[,1] [,2] [,3]
[1,] NA   NA   NA
[2,] NA   NA   NA
```

This creates a 2 by 3 matrix with an NA value in each element. Let's consider how the arguments in the function call are matched to the formal arguments of the `matrix()` function, which are:

```
function (data = NA, nrow = 1, ncol = 1,
          byrow = FALSE, dimnames = NULL)
```

First, all 5 formal arguments have default values and the call frame is set up with these variables and values. Next, we handle all the named arguments in the function call. In this case, there are no named arguments, so we skip to the next stage of matching by position. The first argument is intentionally missing (i.e., we have not provided an argument before the first comma in `(, 2, 3)`). R considers this to be the first formal argument (`data`) and leaves its default value in the call frame. Then R processes the value 2. This is assigned to the next formal argument, which is `nrow`. Lastly, the 3 is matched to the next (3rd) formal argument, which is `ncol`. The call frame now has the values NA, 2, 3, FALSE, NULL for the variables `data`, `nrow`, `ncol`, `byrow`, and `dimnames`, respectively.

The Call Frame in a Function Call

When we call a function, R maps the inputs in the function call to the formal argument names in the function's signature and puts them into a call frame with variables corresponding to these formal argument names. The sequence of steps go as follows:

- Create a frame of variables with one variable name for each of the formal arguments.
- Assign the default value, for each formal argument that has one, to the corresponding variable in the frame.
- Map the inputs specified by name in the function call to the variables in the call frame. Names in the function call can match partially, provided they uniquely identify a formal argument of the function. If a name doesn't match and the function has a ... argument then this variable and value are added to the call frame.
- Map the remaining inputs provided in the function call by position.

TABLE 5.2: Call Frame Set Up:

```
plotRatio = function(r, k = 1, date = seq(along = r), ...)
plotRatio(1:10, 2, col = "red", main = "A Title")
```

Initial Var	Set Up Value	Match Var	Name Value	Match Var	Position Value
r		r		r	1:10
k	1	k	1	k	2
date	seq(along = r)	date	seq(along = r)	date	seq(along = r)
...		col	"red"	col	"red"
		main	"A Title"	main	"A Title"

When an expression is provided as the default value for an argument, this expression is not evaluated until code that references this variable is evaluated. Also, if a variable is referenced in the code that is not in the call frame and has not been created by earlier statements in the function, then *R* looks for this variable in the function's parent environment.

5.4.2 The ... parameter

The ... mechanism is a special kind of formal argument. If *R* cannot match an argument by name or by position and the function has ... in its signature, then the argument is added to the call frame. This is a mechanism by which we can have an arbitrary number of arguments for the call frame. For example, with `c()`, the signature is

```
function (... , recursive = FALSE)
```

The ... argument allows us to concatenate an arbitrary number of variables together. While ... does allow us to have any number of arguments in functions like `c()`, `save()`, `sum()`, and so on, this argument also has another purpose. It allows us to write top-level functions that use ... to take arguments which our function then passes on to lower-level functions by including ... in that function's call.

For example, consider the `plotRatio()` function defined below

```
plotRatio =
function(r, k = 1, date = seq(along = r), ...)
{
  plot(date, r, type = "l", ...)
  abline(h = c(mean(r), mean(r) + k * sd(r),
              mean(r) - k * sd(r)))
}
```

Notice that we allow the caller to pass additional arguments to `plot()` via The caller can use this mechanism to specify a title, axis labels, etc. For example, the following function call, `plotRatio(1:10, col = "red", main = "A Title")` passes the `col` and `main` arguments via the ... to `plot()` (see Table 5.2).

5.4.3 Lazy Evaluation

The call frame for

```
plotRatio(1:10, 2, col = "red", main = "A Title")
```

that is displayed in Table 5.2 shows the expressions `1:10` and `seq(along = r)` as the values for `r` and `date`, respectively. We might think the value for `r` should be the integer vector `1 2 3 ... 10`, rather than the expression `1:10`. The reason for this is that *R* does not evaluate these expressions and assign their value to the associated variable until *R* evaluates an expression that refers to this variable. This is called lazy evaluation.

As an example, consider the `logPlus` function that we wrote in Section 5.2.5. We reproduce it below for reference

```
logPlus = function(x, amt = 1) log(x + amt)
```

We modify this function in 2 ways to show how lazy evaluation works. We change the default value for `amt` to `min(x[x > 0])`. We also add code to check to see if any values in `x` are negative, and if they are, then we replace `x` with its absolute value. The revised function appears as

```
myLog =
function(x, amt = min(x[x > 0]))
{
  if (any(x < 0)) x = abs(x)
  log(x + amt)
}
```

When we call the function with `myLog(0:3)`, the call frame has the expression `0:3` associated with `x` and the expression `min(x[x > 0])` associated with `amt`. The expression `0:3` is evaluated the first time *R* evaluates an expression that contains `x`. This is in `any(x < 0)`. The expression `min(x[x > 0])` is not evaluated until the second line of code, i.e., when `amt` is used in `log(x + amt)`. At this point, *R* assigns 1 to `amt` and our function returns the logarithm of the values `1 2 3 4`, i.e.,

```
[1] 0.00 0.69 1.10 1.39
```

We can confirm that our function behaves as expected by comparing the return value with `log(1:4)`.

What happens when we call the function with `myLog(c(-2, -1, 0, 4, 5))`? The smallest positive value of the input is 4. Does our function return the logarithm of `6 5 4 8 9`? Let's try it.

```
myLog(c(-2, -1, 0, 4, 5))
```

```
[1] 1.10 0.69 0.00 1.61 1.79
```

```
log(c(6, 5, 4, 8, 9))
```

```
[1] 1.8 1.6 1.4 2.1 2.2
```

These values do not match! Why? The reason has to do with lazy evaluation.

When we call `myLog(c(-2, -1, 0, 4, 5))`, the call frame contains expressions for the values of `x` and `amt`. *R* does not evaluate these expressions until it needs to. With the first line of code (the `if` statement), the expression for the value of `x` in the call frame is evaluated and assigned to `x` so `x` is now `2 1 0 4 5`. The variable `amt` appears in the second line of code, so the expression that defines `amt` is not evaluated until we take the logarithm of `x` and at this point, `x` no longer has negative values. This means that `amt = min(x[x > 0])` assigns 1 to `amt` and the return value of our function call is the logarithm of `3 2 1 5 6`. When we think about it, lazy evaluation has provided a reasonable value for `amt`. If the expression `min(x[x > 0])` was evaluated earlier, then the amount may not be what we want. For example, consider `logPlus(c(-0.1, 1))`. Do we want `amt` to be `1` or `0.1`? The latter seems preferable.

5.4.4 The Search Path

In Section 1.13.2, we introduced the search path that *R* follows to find variables when we refer to them in our code. We saw there that *R* chains together a collection of places in which to search for variables and functions. When we type a command at the prompt, *R* begins to look for variables in the global environment. If a variable is not found, then *R* looks in the parent environment and works its way along the chain of environments until it finds the variable. If it doesn't find the variable in any environment then we receive an error. What happens when we call a function and an expression in the function uses a variable?

When we call a function, *R* sets up a call frame for the function. As noted in Section 5.4 and Section 5.4.3, *R* populates the call frame with the function's arguments. Additionally, as *R* executes the expressions in the function, if new variables are created, then these are added to the call frame. The parent of this call frame is the environment in which the function was defined, which is typically the global environment. When our code refers to a variable, *R* searches for it in the call frame, and if it's not there then *R* searches for the variable in the parent environment and along the search path.

To provide an example, we return to our simple function `calcBMI()` from Section 5.2.1 and define 2 alternative functions. We provide the original function for reference,

```
calcBMI =
function(ht, wt, unitConstant = 703)
{
  unitConstant * wt / ht^2
}
```

Our first variant of `calcBMI()` is

```
calcBMI.alt1 =
function(ht, wt, unitConstant = 703)
{
  convert(wt / ht^2)
}
```

We also define `convert()` with

```
convert = function(x) x * unitConstant
```

Notice that `convert()` simply multiplies its input by `unitConstant`. Our second version appears as

```
calcBMI.alt2 =
function(ht, wt, unitConstant = 703)
{
  convert = function(x) x * unitConstant
  convert(wt / ht^2)
}
```

Notice that the function definitions for all 3 versions of `calcBMI()` are identical. Also notice that the body of `calcBMI.alt2()` contains a function definition for a `convert()` function.

Before we explain the differences between our 2 variants, we call them a few times. We first call `calcBMI.alt1()` with

```
calcBMI.alt1(66, 130)
```

```
Error in convert(wt/ht^2) : object 'unitConstant' not found
```

This variant produces an error. However, when we call the other version of the function with the same inputs, i.e., `calcBMI.alt2(66, 130)` we receive

```
[1] 21
```

This seems somewhat curious. If we call `convert()` with `convert(130 / 66^2)` then we get the same error as above. Now let's define `unitConstant` at the prompt with

```
unitConstant = 17
```

Then we find

```
calcBMI.alt1(66, 130)
```

```
[1] 0.51
```

Now, `calcBMI.alt1()` now longer gives an error, but it gets the wrong answer. However, `calcBMI.alt2(66, 130)` still returns the correct answer of `21`.

What is going on here? It has to do with the search path and where `convert()` and `unitConstant` are defined. Both `calcBMI.alt1()` and `calcBMI.alt2()` are defined in `.GlobalEnv`. Also, the `convert()` function that `calcBMI.alt1()` calls is defined in `.GlobalEnv` and `convert()` function that `calcBMI.alt2()` calls is defined in `calcBMI.alt2()`'s call frame. We can confirm this with the `find()` function. That is, `find("calcBMI.alt1")`, `find("calcBMI.alt2")` and `find("convert")` all return `".GlobalEnv"`

When we call `calcBMI.alt1()`, *R* sets up a call frame and places `ht`, `wt` and `unitConstant` in it. Then, when `convert()` is called with `convert(wt / ht^2)`, *R* sets up another call frame. This call frame contains `x`, but it does not contain `unitConstant` because it is not an argument to `convert()` and it is not created in the body of `convert()`. When *R* evaluates the expression `x * unitConstant`, *R* searches for `unitConstant`. Since `unitConstant` is not in the call frame for `convert()`, *R* looks in the parent environment to the function call, which is `.GlobalEnv` because `convert()` was defined there. The first time we called `calcBMI.alt1()`, *R* does not find `unitConstant` in the call frame, it's parent environment, or any of the environments in the search path. This is why we get an error. The second time we call `calcBMI.alt1()`, *R* finds `unitConstant` in the global environment; it has a value of 17 and the function returns `0.51`.

On the other hand, when we call `calcBMI.alt2()`, *R* sets up the call frame with `ht`, `wt` and `unitConstant`, just as for `calcBMI.alt1()`. However, when *R* executes the first expression in

`calcBMI.alt2()`, the `convert()` function is added to the call frame. Next, *R* evaluates the 2nd expression, i.e., `convert(wt / ht^2)`, and sets up a call frame for this function call. Since `convert()` was defined in the first call frame, this is its parent environment. As before, *R* does not find `unitConstant` in call frame for `convert(wt / ht^2)`, but it does find it in the parent environment, i.e., the call frame for the `calcBMI.alt2(66, 130)`. This is why we get the answer `21`. This return value does not change when we assign `unitConstant` to 17 in the global environment because *R* finds `unitConstant` before needing to look in `.GlobalEnv`.

This example may seem silly, but when we collect code that we have been developing into a function, it is easy to forget some of the variables that need to be defined in the code or added to the function's signature. When we test our code, it works because these variables are found in the *R* session (i.e., in `globalenv()`). However, our function may no longer work in a new *R* session because a variable can't be found, or worse, the function gives the wrong results because the global variables have been changed. For these reasons, it's poor programming practice to write functions that depend on variables in the global environment. To help us avoid this, we can use the `findGlobals()` function in the `codetools` package [4], which identifies globals in our functions. For example,

```
library(codetools)
findGlobals(calcBMI.alt2)

[1] "{ " * " /" " ^" " ="
```

The `findGlobals()` function looks for globally defined variables and functions. We see here that for `calcBMI.alt2()` these are only functions in base *R*. On the other hand, `findGlobals(calcBMI.alt1)` returns

```
[1] " { "           "/"           " ^"           " convert"
```

This indicates that `convert()` is defined in the global environment, and when we check it, we find

```
[1] " * "           "unitConstant"
```

That is, `convert()` references `unitConstant`, which as we discovered already, is not defined in the function's call frame.

5.4.5 Partial Matching

R has a mechanism that allows us to abbreviate argument names when they unambiguously identify the particular argument. For example, rather than calling `matrix()` with `matrix(1:10, ncol = 5)`, we can call it with `matrix(1:10, nc = 5)` because `nc` uniquely matches the first 2 letters of `ncol` among the arguments to the function. This style of abbreviated argument names is called partial matching and can make code much harder to read and confusing. It can also lead to some very subtle and frustrating bugs so we suggest avoiding using it.

5.5 Exiting a Function

A simple way for a function to exit and hand control back to the caller is with an explicit `return()` function call. This `return()` can be called with no arguments or with a single value

which can be any *R* object. In many functions, we often want to return the last computed value. *R* makes it easy for us to do this by always returning the result from the last expression evaluated, i.e., we don't necessarily need to use `return()`. For example, there is no `return()` function in our `calcBMI()` function so the result from the evaluation of the last (and only) line of code in this function is returned, i.e., the result is `(unitConstant * wt / ht^2)`. For purposes of clarity, we can add a `return` to `calcBMI()` with

```
calcBMI =
  function(wt, ht, unitConstant = 703)
{
# unitConstant = 1 if wt in kg and ht in m
# for other units: determine constant by converting to kg and m
# e.g., for lb and in: 703 = 0.4536 kg/lb / (0.0254 m/in)^2
  return(unitConstant * wt / ht^2)
}
```

Now, we have made the return value explicit.

If we want to return two or more objects, then we must put them together into 1 object and return that single object. For example, suppose in `calcBMI()`, we want to return both the BMI vector and the constant that was used in creating it. To do this, we can create a named list with these two elements and then return that, e.g.,

```
return(list(bmi = (wt * unitConstant / ht^2),
           unit = unitConstant))
```

Although this is a list with two named elements, we are returning a single *R* object.

Explicitly calling `return()` allows us to exit from a function within loops and conditional statements. We show examples of this in Section 5.6. If we don't want to return anything, then we can call `return()` with no argument. Note, however, that this actually returns `NULL`. Also, if no expression is evaluated then `NULL` is returned.

5.6 Conditionally Invoking Code

In the examples we have worked on in the chapter so far, we have evaluated each statement in the function body sequentially i.e., one expression after another in order from top to bottom. Control flow structures allow us to selectively evaluate statements, repeat the evaluation of statements, and stop or suspend evaluation of code. One primary flow control statement is the `if-else` statement (and the related `ifelse()` and `switch()` functions). We cover these first, and then introduce mechanisms for looping, which augment the suite of apply functions introduced in Chapter 4. Although we typically control the flow of code in functions, conditional statements and loops are not restricted to function bodies.

The `if-else` Statement

The `if-else` statement allows us to perform computations conditionally. The basic format is:

```
if (condition) {
  statements-A
} else {
  statements-B
}
```

The condition in parentheses evaluates to TRUE or FALSE (i.e., a logical vector of length 1), and the statements in the first set of curly braces (the A set) are executed if the condition is TRUE, otherwise the B set of statements in the curly braces following the `else` are evaluated. The `else` construct in the `if-else` statement is optional, and multiple `if-else` statements can be nested. The following example demonstrates a simple use for an `if-else` statement.

Example 5-9 Generalizing the BMI Calculator

Suppose that we want to simplify our `calcBMI()` function so that the caller need only specify whether or not the units of measurements are metric or not. This way, they do not need to know the constant multiplier in the formula, $c * wt/ht^2$. If the units are English, we assume weight is measured in pounds and height in inches, and if metric, the units are kilograms and meters. Our new function definition has a `metric` argument with a default value of FALSE and we no longer need the `unitConstant` argument. Now, we check the value of `metric` and set `unitConstant` to either 703 (when FALSE) or 1 (when TRUE). Our revised function appears as

```
calcBMI =
  function(ht, wt, metric = FALSE)
{
  if (metric) {
    unitConstant = 1
  } else {
    unitConstant = 703
  }
  unitConstant * wt / ht^2
}
```

We call this revised function with, e.g.,

```
calcBMI(1.73, 72, TRUE)
[1] 24.06
```

```
calcBMI(68, 160)
[1] 24.33
```

A more advance function definition keeps `unitConstant` as a formal argument and sets its value based on the `metric` argument. That is, consider the function signature,

```
function(ht, wt, metric = FALSE,
        unitConstant = if (metric) 1 else 703)
```

Here we have placed an `if-else` statement in the function's signautre. A somewhat unusual feature of the `if-else` statement is that it is an expression and so returns a value. We have assigned the result of the `if-else` statement as the default value of `unitConstant`. This value is 1 or 703 and is assigned to `unitConstant` (recall from Section 5.4 that this assignment takes place when `unitConstant` is first used in the function's code). Also note that since the code blocks in the `if-else` statement each contain one expression, we can drop the curly braces. This approach of using both the `metric` and the `unitConstant` parameters has the advantage of allowing the caller to specify a `unitConstant` other than 1 and 703 so we maintain the flexibility of the first version of the function and the simplicity of the second version where we don't need to know the specific value for the constant for standard calculations of BMI. In this case, the function body is the same as for the first function, i.e.,

```
calcBMI =
function(ht, wt, metric = FALSE,
           unitConstant = if (metric) 1 else 703)
{
  unitConstant * wt / ht^2
}
```

We confirm that the function still returns the same values as with the previous modification, i.e.,

```
calcBMI(1.73, 72, TRUE)
```

```
[1] 24.06
```

```
calcBMI(68, 160)
```

```
[1] 24.33
```

However, we can also call this function by supplying a value for `unitConstant` with

```
calcBMI(1.73, 72, unitConstant = 1)
```

```
[1] 24.06
```

```
calcBMI(68, 160, unitConstant = 703)
```

```
[1] 24.33
```

In these examples, the value of `metric` is not checked because `unitConstant` is supplied. As a further example, if weight is in pounds and height is in meters, then we can supply a constant appropriate for these units, i.e., `calcBMI(1.727, 160, unitConstant = 0.4535)`. The result is 24.33; this is the same as `calcBMI(68, 160)` because 1.727 meters is 68 inches and 0.4535 is the constant needed when height is measured in meters and weight in pounds.

In the following sections, we explore the use of `if-else` statements in greater detail.

5.6.1 Conditionally Modifying Inputs to `log(0)`

We revisit the problem that we addressed in Section 5.2.5 of using a log transformation when some values are 0. The `logPlus()` function we created there had two inputs `x`, a vector of the values to be log-transformed, and `amt`, the amount to add to `x` before taking the logarithm. Suppose that we want our function to add this small amount to `x` only if 0s are present in the data. This means that we want our function to perform different computations, depending on the data.

We can modify our function to conditionally compute `log(x + amt)` when `x` has 0s and `log(x)` when there are no 0s. The condition that we check can be expressed as, e.g., `any(x == 0)`, which is TRUE if `x` contains one or more 0s and FALSE otherwise. Our modified function appears as

```
logPlus =
function(x, amt = 1)
{
  if (any(x == 0)) {
    y = log(x + amt)
  } else {
    y = log(x)
  }
  return(y)
}
```

Notice that we have created a variable `y` to hold the transformed values and explicitly return `y` in the last statement of the function.

The flow diagram below provides the sequence of evaluation of the expressions. Each vertical column to the right of the code represents a step in the sequence. When we call `logPlus(0:3)`, the first expression evaluated is the condition in the `if` construct. Since this condition evaluates to TRUE the second statement evaluated is `y = log(x + amt)`. Then, the `else` construct is skipped, and the third statement evaluated is the return.

logPlus(0:3)

<code>logPlus = function(x, amt = 1) {</code>	
<code>if (any(x == 0)) {</code>	X
<code> y = log(x + amt)</code>	X
<code>} else {</code>	
<code> y = log(x)</code>	
<code>}</code>	
<code>return(y)</code>	X
<code>}</code>	

On the other hand, when we call the function with `logPlus(1:3)`, then we have a different code flow, as shown in the diagram below. The first statement evaluated is the condition in the `if` construct, and since this is FALSE, the statement in the `if` block is skipped. In this invocation, the second statement evaluated is in the `else` block, and lastly, the call to `return()` is made.

logPlus(1:3)

logPlus = function(x, amt = 1) {							
if (any(x == 0)) {	x						
y = log(x + amt)							
} else {							
y = log(x)	x						
}							
return(y)	x						
}							

The `else` construct is not always needed and so is optional. To demonstrate, suppose that we want to throw an exception if the function is passed any negative values. Then we can modify `logPlus()` to check if `x` has any negative values and, if so, exit the function without taking the logarithm of the data. We augment our function with an additional `if` statement to handle this situation, e.g.,

```
logPlus =
function(x, amt = 1)
{
    if (any(x < 0)) {
        stop("Negative values in x")
    }
    if (any(x == 0)) {
        y = log(x + amt)
    } else {
        y = log(x)
    }
    return(y)
}
```

When the `stop()` function is called, execution is halted, the function is exited and the text supplied to `stop()` is printed to the console. We do not need to include an `else` here because the subsequent expressions are evaluated only when all data values are nonnegative.

We continue this example to demonstrate how to check multiple conditions in nested `if-else` statements. Let's suppose that instead of stopping the function when we find negative values in `x`, we shift all of the data by $1 - \min(x)$ and then take logs. Now we have 3 possible return values, and our code appears as

```
logPlus =
function(x, amt = 1)
{
    if (min(x) < 0) {
        y = log(x - min(x) + 1)
    } else if (min(x) == 0) {
        y = log(x + amt)
    } else {
```

```

    y = log(x)
}
return(y)
}

```

Below is an example of the sequence of evaluation of the expressions when we call this version of `logPlus()` with `-1:3` as input.

logPlus(-1:3)

<code>logPlus = function(x, amt = 1) {</code>	
<code>if (min(x) < 0) {</code>	X
<code> y = log(x - min(x) + 1)</code>	X
<code>} else if (min(x) == 0) {</code>	
<code> y = log(x + amt)</code>	
<code>} else {</code>	
<code> y = log(x)</code>	
<code>}</code>	
<code>return(y)</code>	X
<code>}</code>	

Lastly, we can modify this code to eliminate all of the `elses`, e.g.,

```

logPlus =
function(x, amt = 1)
{
  if (min(x) < 0) {
    y = log(x - min(x) + 1)
  }
  if (min(x) == 0) {
    y = log(x + amt)
  }
  if (min(x) > 0) {
    y = log(x)
  }
  return(y)
}

```

This version of the function correctly computes the ‘logarithm’ of the data, but it always checks all three conditions. To see this, compare the flow diagrams for this version of the function to the previous version (again when we call the function with `-1:3` as input). Note that this version evaluates 2 extra `if` conditions.

logPlus(-1:3)

logPlus = function(x, amt = 1) {	
if (min(x) < 0) {	x
y = log(x - min(x) + 1)	x
}	
if (min(x) == 0) {	x
y = log(x + amt)	
}	
if (min(x) > 0) {	x
y = log(x)	
}	
return(y)	x
}	

This latest version of the function, the one without any `elses`, is less efficient than the version with the nested `if-else` statements because the code must always check all three conditions. Another inefficiency comes from the computation of `min(x)` repeatedly. We can address this re-computation of `min(x)` by assigning it to a variable at the beginning of the function. We can also return from the function in multiple places to avoid checking all 3 conditions. The issue of efficiency is covered in more detail in [?]. We incorporate the computation of the minimum once and the use of the `return()` function in the `if` code blocks to obtain the following revision of our function:

```
logPlus =
function(x, amt = 1)
{
  m = min(x)

  if (m < 0) return(log(x - m + 1))
  if (m == 0) return(log(x + amt))
  return(log(x))
}
```

Note that the curly braces have been eliminated because these blocks of code contain only one statement, and the condition and associated expression are collapsed onto one line of code. See Section 5.8 for a further discussion of the style guidelines for writing `if-else` statements.

The conditions in an `if-else` statement are single logical values so we often have to map a logical vector of length n to one of length 1 for our condition. The functions `all()` and `any()` are often used for this purpose. We already have seen examples of `any()` in a condition, i.e.,

```
if (any(x < 0)) {
  stop("Negative values in x")
}
```

A similar condition that uses `all()` is

```
if(all(x > 0)) {
  y = log(x)
}
```

The `ifelse()` function

The `ifelse()` function is essentially an element-wise version of `if-else`. As just mentioned, the condition in an `if-else` statement must be a single logical value. If we want to iterate over several elements in a logical vector and do something for each depending on whether it is TRUE or FALSE, then we can use the `ifelse()` function as a convenience function. For a simple example, consider again the issue of taking the logarithm of a vector of numeric values. Suppose we want to return NA for any negative values. The `ifelse()` function assists us here, i.e.,

```
log(ifelse(x >= 0, x, NA))
[1]      NA -1.33  0.36  0.98 -0.33 -0.57
```

We have suppressed the warning and the first value is NA rather than NaN.

The first argument of the `ifelse()` function is a logical vector. The other two inputs should be vectors of the same length as this logical vector. That is, we have 3 vectors of the same length. If the i -th element in the condition vector is TRUE, then the i -th element of the 2nd vector (the `yes` argument) is assigned to the i -th element of the result. Otherwise, the i -th element of the result is taken from the 3rd vector, i.e., the `no` argument. Notice that in our example, the 3rd input vector is of length 1 so this value is recycled to create a vector of 6 NAs to match the length of the logical vector.

Alternatively, we can carry out this vectorized version of the `if-else` with subsetting, e.g.,

```
condition = x >= 0
ans = numeric(length(condition))
ans[condition] = log(x[condition])
ans[!condition] = NA
ans
[1]      NA -1.33  0.36  0.98 -0.33 -0.57
```

Here, we created a logical vector (`condition`) from the expression: `x >= 0`. Then, we create a vector to hold the results (`ans`). This vector is the same length as our condition and by default consists of 0s. We populate `ans` with values from `log(x)` or with NA, depending on the corresponding values in `condition`. That is, the 1st element of `ans` is set to the 1st element of `log(x)` when the 1st element of `condition` is TRUE, the 2nd element of `ans` is set to the 2nd element of `log(x)` when the 2nd element of `condition` is TRUE, and so on. Similarly, in the next statement, we use `!condition` to place NAs in all elements in `ans` where `x` is negative. More generally, the `ifelse()` function is equivalent to

```
ans = numeric(length(test))
ans[test] = yes[test]
ans[!test] = no[!test]
```

Here `test`, `yes` and `no` are equivalent to the 3 vectors of inputs to `ifelse()`. Lastly, we note that the result from `ifelse()` can be a matrix or array (but of course these are still vectors).

5.6.2 Converting Liquid Measures into Tbs. Using `switch()`

Recall from Q.5-4 (page 195) that our goal is to write a function to convert liquid measures into tablespoons. Specifically, the caller supplies the amount and units of the liquid, and the function returns the equivalent number of tablespoons. We have enough information to specify our function signature as

```
convertLiquid = function(x, units = "ml")
```

Here, we require the amount, which is specified in `x` and we provide a default value ("ml") for the units via the `units` argument.

We can carry out the conversion via multiple nested `if-else` statements, i.e.,

```
if (units == "ml") {
  y = x * 0.067628
} else if (units == "l") {
  y = x * 67.628
} else if (units == "oz") {
  y = x * 2
} else if (units == "tsp") {
  y = x / 3
} else if (units == "tbs") {
  y = x
} else {
  warning(paste("Unrecognized units: ", units, "\n"))
  y = NA
}
```

Notice that in the final `else` block we issue a warning that our function does not recognize the units specified by the caller. We provide a value for the conversion that is NA and do not stop the execution.

We can encapsulate this code into our `convertLiquid()` function, but instead, we use the more concise `switch()` function to check the various options for the input units of measurement. We do this with

```
convertLiquid = function(x, units = "ml") {
  units = tolower(units)
  y = switch(units,
    milliliter = , ml = x * 0.067628,
    liter = , l = x * 67.628,
    ounce = , oz = x * 2,
    tablespoon = , tbs = x,
    teaspoon = , tsp = x / 3,
    NA)
  if (any(is.na(y)))
    warning(paste("Unrecognized units: ", units, "\n"))
  return(y)
}
```

We have made our `convertLiquid()` function more robust by converting `units` to lower case before checking its value. We also made the function more robust by adding more options to `switch()` that handle the case when the caller supplies the unabbreviated name for the

units. For these options, we do not supply an expression, which means the next option that has an expression is evaluated so we have paired `milliliter` with `ml`, `liter` with `l`, and so on.

The `switch()` function

The `switch()` function is another type of branching statement. This function lets us use the value of a variable to identify which one of many different alternative expressions to evaluate. It can be more convenient than multiple `if-else` statements, when we can identify the expression to evaluate by the value of the condition itself. More specifically, `switch()` works in 2 distinct ways depending on whether the expression provided in the first argument evaluates to a character string or number. When it's a string, `switch()` then looks for a named argument that matches that value, e.g., when a string is "ml" then the expression assigned to the argument `ml` is evaluated and returned. If none of the argument names match and there is an unnamed argument, then its value is returned. In our example above, we return NA for the conversion amount when none of the argument names match. When a value for an argument is not supplied, e.g., `liter =`, then the next non-missing element is evaluated.

5.7 Iterative Evaluation of Code

Much of what is handled using loops in other languages can be more efficiently carried out in *R* with vectorized computations or using one of the apply functions. However, there are situations where we need looping, such as with algorithms that require recursion. There are two main looping constructs in *R*: the `for` and the `while` loops. The `for` loop is more common so we introduce it first.

For Loops

The basic format of a `for` loop is:

```
for (var in vector) {  
    statements  
}
```

Here, the variable `var` is set to the 1st value in `vector` and the code block between the curly braces is evaluated. Then, `var` is set to the 2nd value in `vector` and the code block is evaluated again, and so on for each element in `vector`. The `vector` often contains integers, but it can be any valid type. If `vector` is a variable, any changes to it within the loop do not affect the values of `var` in the looping. Also, if `vector` has length 0, then the loop is skipped.

5.7.1 Reading Thousands of Files of Data With a `for` Loop

Recall from Q.4-4 (page 153) that our goal is to read many files into *R*. There are so many files that we cannot possibly type the file names at the console, or in a script. Instead, our goal is to programmatically acquire the file names and read the file contents. After downloading and unzipping the data, we found that all of the files are within one directory (folder) called `loans`. Our first step is to capture all of the file names in a character vector; we can do this with the `list.files()` function in *R*. Suppose the folder name is a string in `dirName`. Then we obtain the names of all of the files with

```
fileNames = list.files(dirName, full.names = TRUE)
length(fileNames)
[1] 1975
```

The `full.names` argument indicates that we want the full path names from the top level directory to the file. We have nearly 2000 file names in `fileNames`.

To read any particular file, we can simply call, say, `fromJSON()` and pass the name of one of the files in our character vector, e.g.,

```
x = fromJSON(fileNames[7])
```

We want to generalize this code so that we read all the files. What kind of data structure should we use to hold all of these files? We can use a list. That is, we can loop over the file names, reading in each file and adding it to a list, e.g.,

```
rawData = vector("list", length = length(fileNames))
for (i in 1:length(fileNames)) {
  rawData[[i]] = fromJSON(fileNames[i])
}
```

Notice that our first step is to create a list structure that is large enough to hold all of the raw data. Then, our `for` loop reads each file, one at a time, and puts the result in the appropriate element of `rawData`.

We mentioned that the index in the `for` loop need not be integer values. For example, we can use the file names themselves as our index, i.e.,

```
rawData = vector("list", length = length(fileNames))
names(rawData) = fileNames
for (oneName in fileNames) {
  rawData[[oneName]] = fromJSON(oneName)
}
```

Here, we use the file names as names for the elements in our list. The `for` loop then cycles through these names, i.e., `oneName` takes on the first file name, this file is read into *R* and assigned to the element in the list of that name, then `oneName` takes on the 2nd file name, this file is read into *R* and assigned to the corresponding element in the list, and so on.

Hopefully, the thought has crossed your mind that this seems very much like what `lapply()` does. Recall that the apply functions such as `lapply()` applies a function to each element of a list. We can also supply a character vector to `lapply()`, e.g.,

```
rawData = lapply(fileNames, fromJSON)
```

This one line of code is simpler and focusses on the essence of the computation, i.e., to apply `fromJSON()` to each element of `fileNames`. Notice that with this approach, we do not need to create `rawData` in advance. This approach to looping with `lapply()` often can be faster than using the `for` loop.

Those familiar with for loops in other languages often lean toward using the `for`-loop construct rather than a vector calculation. For example, rather than compute BMI with `bmi = 703 * wt / ht^2`, we can use a for loop, e.g.

 n = length(ht)
bmi = vector("numeric", length = n)
for (i in 1:n) {
 bmi[i] = 703 * wt[i] / ht[i]^2
}

We advise strongly against this approach. Ignoring the vectorized nature of *R* can significantly impact the speed of execution and should be avoided.

The `for` loop is a more appropriate construct, when one iteration of the code block depends on a previous iteration. We provide an example in the next section.

5.7.2 Generating Fibonacci Numbers

We saw in Q.5-6 (page 195) that Fibonacci numbers can be defined recursively, for $n > 1$, by

$$F_n = F_{n-1} + F_{n-2}$$

Additionally, $F_0 = 0$ and $F_1 = 1$.

One way to generate, say, F_6 is to begin with F_0 and F_1 , add them together to get F_2 , and continue adding values together until we reach F_6 . For example, we can begin by creating a numeric vector to hold the 7 Fibonacci numbers (recall that we start with F_0) and setting the first two values with

```
fibNums = vector("numeric", length = 7)
fibNums[1] = 0
fibNums[2] = 1
```

Then, we compute each subsequent Fibonacci number from the previous two with

```
fibNums[3] = fibNums[1] + fibNums[2]
fibNums[4] = fibNums[2] + fibNums[3]
fibNums[5] = fibNums[3] + fibNums[4]
fibNums[6] = fibNums[4] + fibNums[5]
fibNums[7] = fibNums[5] + fibNums[6]
fibNums[7]

[1] 8
```

We can replace each of the computations for the 2nd through 7th Fibonacci numbers with a `for` loop, e.g.,

```
for (i in 2:6) {
  fibNums[i + 1] = fibNums[i] + fibNums[i - 1]
}
```

Notice that the `vector` in our loop consists of the integers 2, 3, ..., 6 and the indexing variable `i` is used to populate the 3rd through 7th elements of `fibNums`. This offset is because of 0-based counting in the series, i.e., the first element in `fibNums` is F_0 .

Now that we have a vector of the 1st 7 Fibonacci numbers, we can abstract this code so that it works for arbitrary Fibonacci numbers and encapsulate it into a function. We leave this as an exercise.

As an alternative approach, notice that we find a particular Fibonacci number from the previous 2 numbers. That is, if `fib.1.bak` is the previous Fibonacci number and `fib.2.bak` is the number before that, then the current Fibonacci number is `fib = fib.1.bak + fib.2.bak`. For example, to find F_2 , we start with the first 2 Fibonacci numbers, which in this case are the previous 2 Fibonacci numbers, and add them together, e.g.,

```

fib.2.bak = 0
fib.1.bak = 1
fib = fib.1.bak + fib.2.bak
fib

```

[1] 1

Can we use `fib.1.bak`, `fib.2.bak`, and `fib` to compute the next Fibonacci number? We can update `fib.1.bak` and `fib.2.bak` as follows:

```

fib.2.bak = fib.1.bak
fib.1.bak = fib

```

Then, we can compute the next Fibonacci number with

```

fib = fib.1.bak + fib.2.bak
fib

```

[1] 2

Notice that we are careful with the order of the re-assignments of `fib.1.bak` and `fib.2.bak`. If we switch the order, then both `fib.1.bak` and `fib.2.bak` have the same value (`fib`), and we do not get the correct result.

If we repeat these 3 steps, we eventually get the desired number in the sequence. That is,

```

fib.2.bak = fib.1.bak
fib.1.bak = fib
fib = fib.1.bak + fib.2.bak
fib.2.bak = fib.1.bak
fib.1.bak = fib
fib = fib.1.bak + fib.2.bak
fib.2.bak = fib.1.bak
fib.1.bak = fib
fib = fib.1.bak + fib.2.bak
fib

```

[1] 8

This is the same value that we obtained earlier, but we did not keep all of the intermediate Fibonacci numbers.

Again, we can write a `for` loop to carry out this calculation, rather than repeat these 3 lines of code the required number of times. Once the initial 2 values of the Fibonacci sequence are defined, (i.e., `fib.1.bak = 1` and `fib.2.bak = 0`), then we can compute any number with the following loop:

```

for (i in 2:n) {
  fib = fib.1.bak + fib.2.bak
  fib.2.bak = fib.1.bak
  fib.1.bak = fib
}

```

Note that we do not use our indexing variable in the `for` loop, i.e., the `i` variable does not appear in our code block.

Our next task is to encapsulate this code into a function. The only input that we need is the desired number in the sequence. The function appears as

```
fibFor =
function(n)
{
    if (n == 0) return(0)
    if (n == 1) return(1)

    fib.2.bak = 0
    fib.1.bak = 1
    for (i in 2:n) {
        fib = fib.1.bak + fib.2.bak
        fib.2.bak = fib.1.bak
        fib.1.bak = fib
    }
    return(fib)
}
```

In this function, we treat F_0 and F_1 specially, i.e., we immediately return their values. The remainder of the code is only evaluated for values of `n` that are 2 or greater. We simply set up the 2 initial values in the sequence and then we loop through to calculate each subsequent number until we arrive at the desired place in the sequence. The diagrams below show the flow of the execution for 2 function calls, `fibFor(1)` and `fibFor(4)`, respectively. The looping in the `fibFor(4)` call is evident in the diagram, where the same 3 statements in the code block are repeated 3 times to obtain F_2 , then F_3 , and finally F_4 .

fibFor(1)

fibFor = function(n) {							
if (n == 0) {	X						
return(0)							
}		X					
if (n == 1) {			X				
return(1)				X			
}							
fib.2.bak = 0							
fib.1.bak = 1							
for (i in 2:n) {							
fib = fib.1.bak + fib.2.bak							
fib.2.bak = fib.1.bak							
fib.1.bak = fib							
}							
return(fib)							
}							

fibFor(4)

fibFor = function(n) {							
if (n == 0) {	X						
return(0)							
}		X					
if (n == 1) {			X				
return(1)							
}							
fib.2.bak = 0		X					
fib.1.bak = 1			X				
for (i in 2:n) {				X		X	X
fib = fib.1.bak + fib.2.bak				X		X	X
fib.2.bak = fib.1.bak					X		X
fib.1.bak = fib						X	X
}							
return(fib)							X
}							

The definition of F_n in terms of F_{n-1} and F_{n-2} suggests an alternative approach for

creating this function. When we need F_{n-1} to determine F_n , we can simply call our function to compute it. That is, we can avoid the `for` loop by having our function call itself. Such a function is recursive, and we define it as follows:

```
fibRec =
function(n)
{
  if (n == 0) return(0)
  if (n == 1) return(1)
  return(fibRec(n-1) + fibRec(n-2))
}
```

When we compute F_4 with `fibRec(4)`, our function calls itself to find F_3 and F_2 . The call `fibRec(2)` also calls our function to find F_1 and F_0 . These two function calls, i.e., `fibRec(1)` and `fibRec(0)`, are special initial cases, and they simply return the value of F_1 and F_0 , respectively. We can easily see that this approach leads to a tremendous number of function calls for even moderate sized Fibonacci numbers. This is not a very efficient approach. To find F_4 , our function calls itself 8 times and 14 times for F_5 .

An alternative definition of the Fibonacci sequence leads to yet another approach for computing these numbers, which is not recursive. This definition relies on the golden mean, i.e.,

$$F_n = \frac{n^{\circ} \left(\frac{1}{\sqrt{5}}\right)^n}{\sqrt{5}},$$

where $\phi = (1 + \sqrt{5})/2$. This definition provides a much faster approach to computing the Fibonacci numbers. Let's write another function that implements this definition:

```
fibTau = function(n) {
  tau = (1 + sqrt(5))/2
  round((tau^n - (-1/tau)^n)/sqrt(5))
}
```

Notice that this definition works for non-integer values of `n`. Also, since `sqrt(5)` can only be approximated to a finite number of digits, the return value may not be an integer. We solve this issue with rounding.

We conclude this section with a comparison of the speed of execution for these 3 implementations. All 3 functions are quite fast, when we call them once with a relatively small input value. However, a comparison of the time to complete one execution of the function might not give us a reliable measurement. For example, if a background process is running on our computer, then it might impact the run time for one of the function calls and not the others. To account for these fluctuations in the measurement process, we repeat the call many times for each function and compare the total times. The idea is that over a longer period of time the impact of these background processes should even out. We use the `replicate()` function to call our 3 candidate Fibonacci functions 5000 times for each of the values 0, 1, ..., 25. The `system.time()` function helps us track the time it takes to execute these calls. We find the non-recursive version, `fibTau()`, is twice as fast as the `for`-loop version, `fibFor()`, e.g.,

```
system.time(replicate(5000, sapply(0:25, fibTau)))
user  system elapsed
0.571   0.008   0.586
```

```
system.time(replicate(5000, sapply(0:25, fibFor)))
```

user	system	elapsed
1.155	0.015	1.189

Here, we compare the 2 elapsed times, each of which is roughly the sum of the user and system times. The user time gives the CPU time spent by the current process and the system time gives the CPU time spent by the operating system on behalf of the current process. The operating system time is very small because it reports the time for things like opening and reading from files and querying the system clock. We see here that both `fibFor()` and `fibTau()` are quite fast with `5000*26` function calls taking the system (a MacBookAir with a 2 GHz Intel Core i7 processor) about 1 second to complete.

Unfortunately, the recursive version of the function is so slow that we cannot replicate the function call 5000 times without our computer hanging for several minutes so we scale back to 100 replications to find

```
system.time(replicate(100, sapply(0:25, fibRec)))
```

user	system	elapsed
63.862	0.317	64.664

To carry out only 100 repetitions, it takes `fibRec()` 55 times longer than it takes `fibFor()` to run 5000 repetitions so 5000 repetitions would take 55×50 , nearly 3000 times longer. We provide more examples of efficiency in Chapter 6 and Chapter 7.

5.7.3 Playing Penney's Game

Recall from Q.5-7 (page 195) that we want to write a function to play a simple coin tossing game. In this game, player A and player B each select a sequence of 3 heads and tails. The two players cannot select the same sequence so player A selects first. A coin is tossed until either player A's or player B's sequence appears, and whoever has the sequence that appears first wins the game.

With this description of the game, we can ascertain that the inputs of the function are the two sequences of heads and tails. Also, we may want to supply a default value for one of the sequences so a possible function signature appears as:

```
function(seq1, seq2 = c("H", "H", "T"))
```

This function does not determine a player's strategy. It simply takes two sequences of heads and tails, and simulates coin flips to play the game and determine the winner.

For this function, we need to flip a coin until we have a winner, i.e., we do not know in advance how many times we need to flip the coin. This is the type of setting where a `while` loop can be useful.

While Loops

A `while` loop evaluates a block of statements if the provided condition is TRUE; then, after evaluating the block of statements, the condition is checked again and if TRUE, the code block is evaluated again; this process continues until the condition evaluates to FALSE. The general syntax of a `while` loop is

```
while (condition) {
  statements
}
```

In this setting, we need to flip the coin time after time while a winner is still undetermined. Below is a possible function to play this game. We create a `winner` variable to use in the condition of our `while` loop. As long as the value is NA for `winner`, we flip the coin again. Each time we flip the coin we check to see if strategy A or B has won. If it has, we set `winner` to the winning strategy, then the condition in our `while` statement tests FALSE and we exit the loop.

```
flipWhile =
function(strat1, strat2 = c("H", "H", "T"))
{
  winner = NA
  flips = sample(c("H", "T"), size = 3, replace = TRUE)

  while (is.na(winner)) {
    if (all(flips == strat1)) {
      winner = 1
    } else if (all(flips == strat2)) {
      winner = 2
    }
    flips = c(flips[2:3], sample(c("H", "T"), size = 1))
  }
  return(winner)
}
```

Let's examine the flow of execution when we call `flipWhile()` with the HHH strategy. Since the code generates random coin flips, we do not know in advance the exact flow so to illustrate we imagine the flips are as follows: H T H H T T. With this sequence, the second strategy wins in the 3rd round. That is, the coin is flipped 3 times, the first sequence examined is HTH and there is no winner. Then the coin is flipped once more to get THH and still no winner. Another flip gives the sequence HHT, and at this point, strategy #2 is matched and wins. The flow diagram appears below.

```
flipWhile(c("H", "H", "H"))
```

flipWhile = function(strat1, strat2 = c("H", "H", "T")) {	
winner = NA	X
flips = sample(c("H", "T"), 3, TRUE)	X
while (is.na(winner)) {	X
if (all(flips == strat1)) {	X
winner = 1	X
} else if (all(flips == strat2)) {	X
winner = 2	X
}	
flips = c(flips[2:3], sample(c("H", "T"), 1))	X
}	
return(winner)	X
}	

Deciding Which Kind of Loop to Use

The following considerations help us choose a looping structure for our code.

Vector Given R's vectorized design, we avoid loops when the task can be performed with vector computations. For example, if we want to add `x` to `y` elementwise, then we don't need a loop. The expression, `x + y` performs this computation. That is, we want to avoid loops such as `for (i in 1:length(x)) z[i] = x[i] + y[i]`.

Apply The apply family of functions (e.g., `sapply()`, `lapply()`, `mapply()`, and `apply()`) perform looping operations by applying the supplied function to each element of a list, data frame, vector, or matrix. These apply functions provide a straight forward and compact syntax for tasks that involve the same computation on each element of a data structure.

For Loop When there is a fixed number of times that a task (or block of code) needs to be repeated or one iteration of the code block depends on a previous iteration, then a `for` loop is typically preferable.

While Loop If a task needs to be repeated until a condition occurs or while a condition holds, then a `while` loop is often the clearest approach.

Repetition When a task needs to be repeated a fixed number of times and the input does not change from one repetition to the next, then the `replicate()` function can be quite useful. Additionally, a `repeat` statement can be useful in this situation.

A `break` statement is added in the code block to repeat the computations and until a condition is met.

repeat Loops With **break** Constructs to Control Looping

The `repeat` construct executes a block of code over and over again and relies on us to determine when to exit the code block. That is, the `repeat` construct requires a `break` or some other means for exiting the loop as it does not include any test such as the condition used in a `while` loop nor does it loop over the values in a vector such as with a `for` loop. We typically use a `break` statement to exit the `repeat` code block. A `break` transfers control to the statement immediately following the code block. The `break` construct is also useful with `while` loops, which if we are not careful, might loop infinitely (or until we kill our R session). The `break` construct does not return a value as it simply transfers control within the loop.

We continue with Penney's game and create another version to demonstrate how to use `repeat` and `break`.

```
flipRepeat = function(strat1, strat2 = c("H", "H", "T")) {
  winner = NA
  flips = sample(c("H", "T"), size = 3, replace = TRUE)

  repeat {
    if (all(flips == strat1)) {
      winner = 1
    } else if (all(flips == strat2)) {
      winner = 2
    }
    if (!is.na(winner)) break
    flips = c(flips[2:3], sample(c("H", "T"), size = 1))
  }
  return(winner)
}
```

The `repeat` construct is similar to the `while` except that we must control the exiting from the loop with an explicit `break`. The advantage of `while` and `repeat` loops is that we do not need to specify the number of times a block of code is to be evaluated. Instead, we evaluate the code repeatedly while a condition evaluates to TRUE (or until it evaluates to FALSE).

We can use either `flipWhile()` or `flipRepeat()` to discover Player B's best strategy given Player A's strategy. Let's start by enumerating all possible sequences of 3 heads and tails. We use `expand.grid()` to do this with

```
coin = c("H", "T")
strategies = as.matrix(expand.grid(coin, coin, coin,
                                    stringsAsFactors = FALSE))
strategies
```

	Var1	Var2	Var3
[1,]	"H"	"H"	"H"
[2,]	"T"	"H"	"H"
[3,]	"H"	"T"	"H"

```
[4,] "T"  "T"  "H"
[5,] "H"  "H"  "T"
[6,] "T"  "H"  "T"
[7,] "H"  "T"  "T"
[8,] "T"  "T"  "T"
```

Our `strategies` matrix contains all 2^3 possible sequences of 3 flips.

For a particular sequence, say HHH, we want to find the best strategy to play against this sequence. We can do this by playing many rounds of the game for each alternative and tallying up the number of times the strategy beat HHH. Let's see how well the strategy THH, does against HHH. To do this, we call `flipWhile()` many times with `strat1 = strategies[1,]` and `strat2 = strategies[2,]`, and we tally the number of times the second sequence wins, e.g.,

```
sum(replicate(5000, flipWhile(strat1 = strategies[1, ],
                               strat2 = strategies[2, ])) == 2)
```

```
[1] 4349
```

We see that in the 5000 trials, the second strategy beat the first 4349 times! If these two strategies are equally likely, then we expect THH to beat HHH about $1/2$ the time, i.e., roughly 2500 times. Moreover, if the 2 strategies are equally likely then typical number of wins are almost always within 100 of 2500, (we can figure this out with another simulation, but we leave these ideas until Chapter 7).

Let's try another sequence for comparison. We expect TTT to have no advantage over HHH. Can you reason why that might be the case? When we play the game for another 5000 rounds for these 2 strategies, we find

```
sum(replicate(5000, flipWhile(strat1 = strategies[1, ],
                               strat2 = strategies[8, ])) == 2)
```

```
[1] 2467
```

In this case, the number of times TTT beat HHH is well within the range of expected of 2 equally matched strategies.

If we want to take a more systematic approach to comparing strategies, then we can create a function to help us. Our function can compare a given strategy to all of the others so it takes a single strategy as an input. We may also want to specify the number of times to play the game. In our first comparison, we played the game 5000 times. We suggest making this value an argument to our function so we can increase the number of rounds in the game if two strategies are close or decrease the repetitions if it takes too long to play 5000 rounds. We also include the strategies we are comparing against as an argument, in case we do not want to check our strategy against all the other possibilities. We encapsulate our earlier code into a function as follows:

```
winningStrat =
function(stratA, strategies, n = 5000)
{
  counts = apply(strategies, 1,
    function(strat) {
      sum(replicate(n, flipWhile(stratA, strat)) == 2)
    }
  )
}
```

```

best = order(counts)[nrow(strategies)]
return(paste(strategies[ best, ], collapse = ""))
}

```

(Note that in `winningStrat()`, we return the winning strategy as a single string with all 3 flips to make it easier to examine.) In this code, we have used 3 different looping mechanics: 1) we call `apply()` to loop over the rows in the matrix, `strategies` and compare each to `strata`; 2) we call `replicate()` to play the game again and again (`n` times); and 3) hidden within `flipWhile` we use a `while` loop in the coin flipping to play 1 round of the game.

We can apply our `winningStrat()` function to all sequences to find which strategy beats each sequence. We do this with:

```

winner = vector(mode = "character", length = nrow(strategies))

for (i in 1:nrow(strategies)) {
  winner[i] = winningStrat(strategies[i, ], strategies[-i, ])
}

winner

```

```
[1] "THH" "TTH" "HHT" "HTT" "THH" "TTH" "HHT" "HTT"
```

For comparison purposes, below are the corresponding sequences that each of these winning strategies is beating:

```
[1] "HHH" "THH" "HTH" "TTH" "HHT" "THT" "HTT" "TTT"
```

We can compare these two vectors and find that `HHH` is beaten most often by `THH`, `THH` is beaten most often by `TTH`, and so on. Can you find a pattern? If you can find a pattern, can you explain why this happens? We have just carried out a simulation study of Penney's game. Simulation studies can be very useful for developing intuition about random processes, such as this coin flipping game, and they can offer insight into the performance of statistical techniques. We address simulation studies in greater detail in Chapter 7.

5.8 Style Guidelines

There is more to writing a program than getting the syntax right, fixing the bugs, and making it run fast enough. Programs are read not only by computers but also by people. A well-written program is easier to understand and to modify than a poorly-written one. In general, when we write code, we strive for simple clear code with straightforward logic, meaningful names, neat formatting, and helpful comments.

We write code to carry out tasks, and through our code we communicate our ideas. We may be communicating these ideas to ourselves or to others who use our code. When we communicate through a written language, we adopt conventions for paragraph breaks, section titles, references etc. These conventions may vary from one piece of writing to another, but they remain consistent throughout the piece. When we write code, we adopt a set of conventions for formatting our code that makes it easier for us and others to read our code, and we keep these consistent within a function. In this section, we provide a set of conventions, or style guidelines, that we have adopted for this purpose.

Style Guidelines

Spacing Put a space after a comma, before and after infix operators, and before a left parenthesis (except in a function call).

Indentation Indent an additional 2 spaces for each sub-block of code. Do not mix tabs and spaces.

Names Use meaningful names; make function names verbs; use all lower case, except when the name has 2 (or more) parts, then, e.g., `varName`, `var.name`, and `var_name` are OK.

Curly Braces Place a closing curly brace on its own line; do not put an opening curly brace on its own line. One exception is the open curly brace in a function definition; another exception is in the `else` construct, where we always use

```
} else {
```

Lastly, omit curly braces when code is only one line, if desired, but be consistent.

Functions Provide arguments without default values first in the function signature. Comment code, including adding comments at the top of function that describe what the function does and what are the inputs and output.

Line Length Keep lines of code to about 80 characters. Break expressions at commas and align the wrapped line with the first argument in the previous line. Avoid using semicolons, i.e., put each expression on its own line.

Indentation and Spacing

As discussed in Chapter 1 we place spaces after commas in function calls and before and after operators, such as `=`, `+`, and `/`. Additionally, we indent code when we use control flow constructs. With functions, loops, and `if-else` statements, we aim to help us (and others) see the structure by indenting our code. For example, compare the following two functions. The code is identical, except in the use of optional spaces. The first uses no indentation or spacing.

```
function(x, epsilon) {
  if(any(x<=0))
    x[x<=0]=epsilon
  for(i in x) {
    g(i)
  }
}
```

The second version indents 2 spaces for each level of code and has spaces before and after operators and after commas.

```
function(x, epsilon)
{
  if (any(x <= 0))
    x[x <= 0] = epsilon

  for (i in x) {
```

```

        g(i)
    }
}

```

In the second version, the indentation makes it clearer when the for loop begins and ends and what code is in the if block. The spaces surrounding operators, such as `<=`, and following commas also help to highlight the code's structure.

In general, we align commands that are at the same level, and we indent code with 2 spaces for each level. For example, expressions within a function are indented 2 spaces; if there is a `for` loop in the function, then the code within the loop is indented an additional 2 spaces; and if there is an `if-else` statement nested within this loop, then the code block in this construct is indented another 2 spaces. Other conventions use more spaces, e.g., 4 spaces or 1 tab. Either way, consistency matters because it helps us easily determine the code blocks.

Breaking Expressions Across Lines

Long lines of code can be hard to read, especially if we have to scroll in order to see the entire expression. We break long expressions across multiple lines so that we can see the entire expression within a window of 60 to 80 characters. To do this, we typically break the expression at commas. The following example shows how to align code to make it easier to read.

```

counts = apply(strategies, 1, [1]
              function(strat) [2]
                sum(replicate(n, flipWhile(stratA, strat)) == 2) [3]
              [4]
            ) [5]

```

- [1] Break the function call after a comma in the argument list
- [2] Align the arguments in the continuation line with the first argument of the function.
Also, break the function definition after the opening curly brace.
- [3] Indent 2 additional spaces to indicate this code is in the body of the function.
- [4] Align the closing curly brace with the first line of the function definition.
- [5] Align the closing round brace with the opening round brace.

Braces

Parentheses (round braces) are needed for function calls and to control the order of operations. We recommend also using parentheses when they are not strictly required in order to resolve ambiguity. Curly braces are needed in function definitions and flow control. We adopt the convention of placing the closing curly brace on its own line. One exception is in an `else` construct where we always put the `else` on the same line as the closing curly brace for the `if`, i.e.,

```

if (condition) {
  statements
} else {
  statements
}

```

Additionally, we do not place opening curly braces on their own lines, with the exception of the opening curly brace for a function definition. Curly braces can be omitted when the function or code block contains only one expression. We recommend choosing between these two options (use curly braces with a one line code block or not) and remaining consistent.

The following function contains an `if-else` statement, and we have not used curly braces because the code blocks are each only one line:

```
newSqrt = function(x)
{
  if (any(x < 0)) sqrt(abs(x))
  else sqrt(x)
}
```

This is one of the few situations where code within a function behaves differently than code written at the command line. When we call `newSqrt()` the function behaves as expected, e.g.,

```
newSqrt(-1:3)
[1] 1.000 0.000 1.000 1.414 1.732
```

However, if we execute these commands interactively at the prompt, then we get an error, e.g.,

```
> x = -1:3
> if (any(x < 0)) sqrt(abs(x))

[1] 1.000000 0.000000 1.000000 1.414214 1.732051

> else sqrt(x)
Error: unexpected 'else' in "else"
```

(We have added the prompt in this code display to make it clear that these statements are evaluated at the command line and not within a function). The error occurs because the expression `if (any(x < 0)) sqrt(abs(x))` is complete. *R* does not know that there is an `else` that is part of this `if-else` construct. That is, the read-evaluate-print-loop does not wait for the `else`. We need to use curly braces and place the closing brace for the `if` block on the same line as the `else` and its opening curly brace.

Commenting Code

Comments help the reader understand how the code works. We typically add comments at the beginning of a function to briefly describe the purpose of the function and to explain the expected inputs and the return value. When we write comments, we keep Pike's guidelines in mind [2] and follow these do's and don'ts:

- Do clarify; don't confuse.
- Do be succinct; don't belabor the obvious.
- Don't comment bad code; instead, rewrite it.
- Don't contradict the code in the comments; do keep comments and code in synch.

Naming Conventions

As discussed in Chapter 1, we recommend using descriptive variable and function names. These names should suggest the nature of the values being stored in the variable or the purpose of the function. If a name consists of two (or more) words then capitalize the second (and later) words or put periods or underscores between them, e.g., `overWt`, `over.wt`, or `over_wt`. Additionally, use short names for local variables, and define 'variables' for constants for clarity and because these constants might change.

5.9 Beginning Notions of Debugging

One of the nice things about writing code in an interpreted language like *R* is that there is no compilation step. We simply `source()` our function into *R* and we can use it. However, a compilation step provides an opportunity to check certain things about the code that we don't have when we source our code into *R*. Only the syntax of our code is checked at this point. It is common to forget to close a quotation mark or a parenthesis, omit a comma between arguments in a function call, etc. The result is a syntax error, and *R* announces where it thinks this error is in our code by supplying a line number. Often, this identifies the problem precisely, but at other times, the error can be one of the prior expressions.

Debugging Different Sources of Errors

Syntax Error *R* checks the syntax of our code, when we source the code into *R*.

Common mistakes include a missing or extra parenthesis, curly brace, quotation mark, or comma. Also, invalid variable or function names cause syntax errors.

Abnormal Termination When the function is called, the function may stop without completing its task due to an error. These errors can result from an improper function call, e.g., not specifying the arguments correctly, or mistyping a variable or function name so the object is not found.

Error with no Message Even though our code executes without yielding an error message, we may have made a mistake in writing our code. That is, the code runs to completion, but returns an incorrect value. These errors can be detected via a test bank of inputs and expected outputs.

Example 5-10 Fixing Some Buggy Code

Below is a very simple function that takes a vector `x` and returns the lower and upper endpoints of the whiskers, i.e., the lower endpoint is

$$\text{lower quartile} - 1.5 \times \text{IQR}$$

and the upper endpoint is $1.5 \times \text{IQR}$ above the upper quartile. The function appears as

```
whisker.endpoints =
function(x)
{
  if (is.na(x)) warning("Careful x has NAs")
  qlu = quantile(x, probs = c(0.25, 0.75), na.rm = FALSE)
  iqr = IQR(x, na.rm = FALSE)

  return(x - 1.5 * iqr, x + 1.5 * iqr)
}
```

This function contains several errors, which we uncover as we discuss the preliminary ideas in debugging code.

When we source this function, *R* indicates that we have a syntax error. The error message is:

```
source('buggyCode.R', echo=TRUE)
```

```
Error in source("buggyCode.R", echo = TRUE) :
  buggyCode.R:3:8: unexpected symbol
2: {
3:   if (is.na(x) warning
      ^
```

The message points to a problem with line 3 of our code. Can you spot the issue? It appears that we are missing a closing parenthesis on our `if` condition. We add this right parenthesis to our code and try again. Our revised function appears as

```
whisker.endpoints =
function(x)
{
  if (is.na(x)) warning("Careful x has NAs")
  qlu = quantile(x, probs = c(0.25, 0.75), na.rm = FALSE)
  iqr = IQR(x, na.rm = FALSE)

  return(x - 1.5 * iqr, x + 1.5 * iqr)
}
```

Now, when we source this version of the function, we do not receive any error messages.

Aside from syntax checking, the interpreter does not examine the code in the function to try to detect any errors. Rather, it waits until the function is called. This is when errors from, say, mistyping a variable or function name appear. Another potential problem occurs if we have used a variable that was in our session when we developed our code interactively, and it is no longer present when we run the function in a new R session. In these circumstances, when we call the function, *R* complains about a variable not being found. We saw this kind of error in Section 5.4.4 and we used `findGlobals()` in the `codetools` to help identify these ‘free’ variables.

When we call our revised `whisker.endpoints()` we receive the following error message:

```
whisker.endpoints(1:19)

Error in return(x - 1.5 * iqr, x + 1.5 * iqr) :
  multi-argument returns are not permitted
In addition: Warning message:
In if (is.na(x)) warning("Careful x has NAs") :
  the condition has length > 1 and
  only the first element will be used
```

The error message indicates that we have supplied too many arguments to the `return()` function. Recall that we can return only one object. We must combine our lower and upper endpoints into one object. We can combine them into a vector with `c()`.

We also received a warning message, which we should not dismiss out of hand. Is this a warning message that concerns us? Indeed, the error message indicates that only the first element of the logical vector `is.na(x)` is being used in the condition of the `if` statement. We want to know if any of the values in `x` are NA, and if so then we issue a warning. We need to modify our condition to: `any(is.na(x))`. Our revised code is now:

```
whisker.endpoints =
function(x)
{
```

```

if (any(is.na(x))) warning("Careful x has NAs")
qlu = quantile(x, probs = c(0.25, 0.75), na.rm = FALSE)
iqr = IQR(x, na.rm = FALSE)

return(c(x - 1.5 * iqr, x + 1.5 * iqr))
}

```

Another source of errors occurs when something in the computations go wrong for more subtle reasons, and *R* terminates the call to the function with an error message. In this case, we read the error message and try to make sense of it. If we are particularly confused, we can copy and paste the error message into the text box of a search engine, add the letter R to the query, and find instances when others had a similar problem. Other online resources include stackoverflow ([stack overflow.com](http://stackoverflow.com)) and the *R*-help mailing list archive (tolstoy.newcastle.edu.au/R/). In Chapter 7 and Chapter 6, we demonstrate how we can go into the function when an error occurs and investigate what the states of the different variables are at the time of the error.

We call the latest version of our function with

```
whisker.endpoints(1:19)
```

```
[1] -12.5 -11.5 -10.5 -9.5 -8.5 -7.5 -6.5
[9] -5.5 -4.5 -3.5 -2.5 -1.5 -0.5  0.5 ...
[33] 27.5 28.5 29.5 30.5 31.5 32.25
```

We expected 2 numbers, the lower and upper endpoints of our the whiskers, but we were given 38 values instead. What went wrong? A more careful examination of the return value shows that we returned `x - 1.5 * iqr` instead of `qlu[1] - 1.5 * iqr`.

This is an example of an error occurring in our code but no error message is given. In this case, the code runs to completion, but returns an incorrect value. It can be hard to detect these situations because there is no abnormal termination of a function call, no error message. In fact, there is no sign that anything has gone wrong because the system does not know that anything has gone wrong. This is a situation where tests can be helpful, i.e., where we create various cases for inputs and check that the output matches what we expect. A systematic approach to developing tests is addressed in Chapter 6. We also can use EDA to analyze the output of our functions to see if the values are reasonable. As we uncover errors and fix them, we add more tests to the test suite to catch or watch for these errors.

We have been using `1:19` as a test case. At the console, we can compute the lower and upper quartiles and the inner-quartile range of `1:19`; these are 5.5, 14.5, and 9, respectively so the return value should be -8 and 28. We also want to set up a test of the warning message. We can use `c(NA, 0:19)` for this test; the return value should be the same as for the call with `1:19` as the input. We fix our problem of returning `x` rather than the `qlu`, and we call our function with these test cases. We find we get the expected results in the first cast but not in the second. That is,

```
whisker.endpoints(1:19)
```

```
25% 75%
-8    28
```

```
whisker.endpoints(c(NA, 1:19))
```

```
Error in quantile.default(x, probs = c(0.25, 0.75),
na.rm = FALSE) :
missing values and NaN's not allowed if 'na.rm' is FALSE
In addition: Warning message:
In whisker.endpoints(c(NA, 1:19)) : Careful x has NAs
```

The return value for the first call matches our expectations. However, for the second call, we have received the warning message as expected, but we have also received an error. Further inspection of our code, reveals that we mistakenly set the `na.rm` parameter to FALSE instead of TRUE. We correct our code again and our function now reads

```
whisker.endpoints =
function(x)
{
  if (any(is.na(x))) warning("Careful x has NAs")
  glu = quantile(x, probs = c(0.25, 0.75), na.rm = TRUE)
  iqr = IQR(x, na.rm = TRUE)

  return(c(glu[1] - 1.5 * iqr, glu[2] + 1.5 * iqr))
}
```

This function now passes our tests.

We provide additional examples of debugging in Chapter 7 and Chapter 6, where the functions are more complex and the errors are more subtle.

5.10 Summary

For general advice on programming including style guidelines see Kernighan and Pike [2]. For an example of guidelines specific to *R*, see Google's *R* Style Guide [1].

5.11 Control Flow and Debugging Functions

ifelse() Apply `if-else` in an element-wise fashion. The dimensions of the returned object are the same as the specified object in the first argument.

switch() Convenience function for multiple nested `if-else` statements. The behavior depends on whether the first argument is a string or a number. If the first argument is not a string, it is coerced to an integer and returns the result in the corresponding order of the coerced integer. If the first argument is a string, the resulting match is returned; if there is a match but the return value is missing, the next non-missing value is returned; if there is no match, the unnamed element is returned.

system.time() Return the amount of time it takes to execute the supplied call. The elapsed time is roughly the sum of the user and system times.

`replicate()` Apply supplied call `n` number of times. The result is simplified to a vector or matrix, when possible.

`return()` Return the supplied value from the function environment.

`warning()` Display the supplied warning message.

`stop()` Stop the execution and display the supplied error message.

`findGlobals()` (in `codetools`) Return the names of any variables or functions used within the function supplied that are in the global environment. When `merge` is FALSE, the returned value contains a vector of variable names and a vector of function names in a list.

5.12 Guided Practice

This section provides practice on writing functions. Make sure to test your functions on different arguments to get a sense of how the function behaves.

1. The following two functions return 0.
 - (a) Write a function called `returnZero1()` that takes in any object and returns 0. This function has one required argument `obj`, which can be any R object.
 - (b) Write another function `returnZero2()` that takes in no arguments and returns 0.
2. Write a function called `squared()` that returns the squares of each element in a numeric vector.
 - (a) First, write a function called `squaredNoCheck()` that squares each element in a numeric vector. The function has one required argument `x`, which is a numeric vector.
 - (b) Modify the previous function to check that the argument `x` is a numeric vector. If the input is not a numeric vector, still attempt to return the squared value each element in `x` but issue a warning message. Name the function `squared()`.
3. Write a function called `fAndG()` that calculates the output from two mathematical functions $f(x, y)$ and $g(x, y)$, where $f(x, y) = x^3 + 2xy + \sin(y)$ and g is a mathematical function of your choosing.
 - (a) First, write a function called `fAndgNoCheck()` that takes in two vectors `x` and `y` and computes $f(x, y)$ and $g(x, y)$. The function returns a data frame with two columns: the first column contains the output from the function f and the other column contains the output from the function g .
 - (b) Modify the previous function to check that `x` and `y` are the same length. If not, exit the function without computing anything and return an error message. Name this function `fAndg()`.

4. Write a function to compute the sum of the absolute deviations from the median. For example, for a vector `x = 1:3`, `x` has a median of 2 and the absolute deviations from the median are 1, 0, and 1, so the sum of the absolute deviations from the median is 2. The function name is `sadm()` that computes this statistic for a vector. The function has two arguments: `x`, which is the required vector, and `na.rm`, which has a default value of FALSE. The `na.rm` argument determines whether NAs are to be removed from the computation: if there are NAs in the vector `x` and `na.rm` is FALSE, the function returns NA.
5. Write a function called `minmax()` that returns a vector containing the positions of the smallest and largest values in a vector.
 - (a) First, write a function called `minmaxNumeric()`. This function has one argument: `x`, which is the required vector. You may assume that the vector is numeric. For now, you may also assume that the vector is not empty and does not contain NAs.
 - (b) Modify `minmaxNumeric()` to check that the required vector `x` is not empty. If the vector is empty, then there is neither a minimum nor maximum value and the function returns NAs for their positions. Name the function `minmaxCheckEmpty()`.
 - (c) Finally, modify `minmaxCheckEmpty()` so that it has an optional argument `na.rm`, which has a default value of FALSE. The argument `na.rm` is used to specify whether the NAs are to be ignored in the calculation or not. If the NAs are not ignored and there are NAs present, then we can't determine the desired positions so we return NAs for them. Name this function `minmax()`.
6. We want to study the distribution of the sum of three rolls of a die. To do this, we design a simulation study. In the first step, we write a function to generate the sum of three random tosses of a fair die. In the second step, we use this function to generate many of these sums.
 - (a) Write a function called `sumDice()` for the first step. This function has no arguments.
 - (b) Write a function called `simSumDice()` for the second step. This function has one required argument `B`, which indicates the number of simulations, and it should use the `sumDice()` function.
 - (c) Execute the study with `simSumDice()` to simulate 1,000 of these sums and plot the results in a meaningful way.
7. Write a function called `showValueChange()` that calculates the percent increase or decrease between two specified prices in a numeric vector. This function takes two arguments: the required vector of prices `x` and the optional argument `compare`, which is a numeric vector of length two that specifies the indices of the two prices to be compared. The default behavior of the function is to compare the first and last prices. The function returns the single numeric value that is the percentage change from the first specified price to the second specified price.

For example, if the vector of prices is \$2.30, \$3.10, and \$4.00, and we want to compare the first and third prices, then the function returns the percent increase from \$2.30 to \$4.00: $|4 - 2.3|/2.3 \times 100\% \approx 73.9\%$. Similarly, if the vector of prices is reversed (i.e., \$4.00, \$3.10, and \$2.30) and we want to compare the first and third prices, then the function returns the percent decrease from \$4.00 to \$2.30: $|2.3 - 4|/4 \times 100\% = 42.5\%$.
8. Write a function called `yLim()` that takes in two vectors, `x` and `y`, and returns the values in `y` that correspond to the same positions as the positions of the minimum and

maximum of `x`. That is, if `x` is `c(100, 13, 1, 20)` and `y` is `c(5, 7, 9, 0)`, then the return value is 9 and 5: the minimum of `x` occurs in its third position and the maximum of `x` occurs in its first position, so the third and first values in `y` are 9 and 5, respectively.

The arguments of this function are `x`, `y`, and `na.rm`. The first two arguments (`x` and `y`) are required and the third argument (`na.rm`) has a default value of FALSE. When `na.rm` is TRUE, the NA values are ignored when finding the minimum and maximum. If NA values are not ignored and NAs are present, then the return value is NA. Also, `x` and `y` should be the same length. If `x` is shorter, then a warning message is issued and the computation is still carried out. If `y` is shorter, then an error message is issued and nothing is returned. You may assume that there is a unique minimum and maximum in `x`.

9. We want to carry out a simulation study of trimmed means from a normal distribution. In particular, we are interested in studying the trimmed mean of a sample of n values from a normal distribution with mean 0 and sd s .
 - (a) First, write a function called `trimMean()` that samples from the normal distribution and then calculates the trimmed mean. The arguments of the function are: the required parameters `n` and `s` and the optional argument `trim`. The `trim` argument is the fraction of smallest and largest observations to be trimmed from the sample and has a default value of 0.05.
 - (b) Write a function called `simTrim()` that averages the trimmed means from `Rep` simulations. The function has the same arguments as `trimMean()` and an additional argument: the optional argument `Rep`, which is the number of times to carry out the simulation and has a default value of 100. Furthermore, the function should make use of the `trimMean()` function.
 - (c) Finally, write a function called `manySimTrim()` that returns the simulation result for different combinations of `n` and `s`. The return value from this function is a three-column data frame or matrix. The first column contains the values of `n`; the second contains the values of `s`; and the third contains the average of the `Rep` trimmed means for the particular combination of `n` and `s`. For example, the output of `manySimTrim(n = c(5, 100), s = c(1, 3, 5))` has a return value that appears as follows:

```
5 1 value
5 3 value
5 5 value
100 1 value
100 3 value
100 5 value
```

where `value` is the average of the `Rep` trimmed means for that particular combination of `n` and `s`. This function should use the `simTrim()` function and have the same arguments.

5.13 Exercises

Bibliography

- [1] Google, Inc. Google's R Style Guide. <https://google.github.io/styleguide/Rguide.xml>.
- [2] Brian W. Kernighan and Rob Pike. *The Practice of Programming*. Addison-Wesley Professional, 1999.
- [3] Walter Penney. Problem 95: penney-ante. *Journal of Recreational Mathematics*, page 241, 1969.
- [4] Luke Tierney. codetools: Code Analysis Tools for *R*. <http://cran.r-project.org/web/packages/codetools>, 2014. *R* package version 0.2-9.

6

Resampling for Inference and Prediction

CONTENTS

6.1	Introduction	245
6.2	The Bootstrap and Inference	247
6.2.1	Percentile Bootstrap of Median House Price	247
6.2.2	Studentized Bootstrap for Repair Times	251
6.2.3	Jackknife Estimate of the Standard Error for Average Repair Time	261
6.3	Permutations and Testing for a Gender Effect	267
6.4	Cross-Validation and Prediction	275
6.4.1	Nearest Neighbor Prediction	279
6.4.2	Hold Out Test Set	289
6.4.3	v-fold Cross Validation	295
6.4.4	Lazy Evaluation	299
6.5	Summary	299
6.6	Guided Practice	299
6.7	Exercises	299
	Bibliography	299

6.1 Introduction

We collect data for the purpose of drawing an inference about a population or making a prediction. To make an inference, we use information about how the data were generated, such as a random sample from a population, a random assignment to a treatment, or observations from a self selected group. These data give us partial knowledge, and from them we make inferences to the entire population or predictions for future individuals.

With resampling techniques, we use the computer to simulate the sampling process to gain an understanding of the variability in our estimates and predictions. That is, we mimic the process of sampling by picking another sample from our data and we examine the variability these samples and the resulting estimators and models. Then, we can provide interval estimates of quantities of interest, choose between competing models based on the accuracy of predictions, and test hypotheses. Typically, resampling techniques make few assumptions about the underlying model that has produced the data. However, they do rely on the underlying random mechanism that generated the data. In simulation studies, we made assumptions about the probability model generating the data, e.g., in EXAMPLE, we studied the log-normal distribution and in EXAMPLE, we studied the bivariate normal and Rayleigh distribution, then from this probability model we generate synthetic data. With resampling, we begin with observed data, and use these data as the basis for generating samples.

In this chapter, we use simulation to generate samples from our existing data. Specifi-

cally, we make extensive use of simple random sampling. Other types of sampling are used in resampling, when the data arise from different sampling methods; some of these are explored in the exercises. However, the basic idea remains the same: generate samples from the original sample and examine the variability in these ‘resamples’ to gain information about the accuracy and validity of inferences and predictions.

In this chapter, we examine several resampling methods. We use the bootstrap and jackknife to provide interval estimates of population parameters; permutation tests to compare responses between groups; and cross-validation to select a model for prediction. These resampling methods offer an easy entry point into statistical inference and serve as a platform for demonstrating programming concepts.

Example 6-1 Estimating Median House Price

When reporting housing prices, we typically summarize them with the median price because the distribution of prices is often skewed right and the median is more representative of the typical price. We have a random sample from the public record of 1000 homes located in Saratoga County, New York. The data includes information about the size of the house, the number of fire places, and price. CITATION ■

Example 6-2 Inferring Average Repair Time

The New York Public Utilities Commission monitors the response time for repairing landline phone service in the state. These repair times may differ over the year and according to the type of repair. We have repair times for one class of repairs at one time period. The commission is interested in estimates of the average repair time. CITATION ■

Example 6-3 Does Gender Matter in Student Evaluations of Teaching?

In an effort to measure the effect of instructor’s gender on teaching evaluations, a randomized controlled experiment with students in an online course was carried out. Students in the course were randomly assigned to one of 6 sections—2 taught by the professor, 2 by a female graduate teaching assistant (TA), and 2 by a male TA. The students in the course never met the TAs face-to-face; their interactions with the TAS were only through an online discussion board and email. Additionally, the TAs coordinated their grading so assignments were graded with the same rubric and returned at the same time. Each TA taught one section using his or her true name and one section using the name of the other TA. In other words, 2 sections were identified as being taught by a female TA and 2 by a male TA, but one of the ‘female’-sections was actually taught by the male TA and one of the perceived male sections was led by the female TA. The TAs had comparable levels of experience. All together 47 students were randomly assigned to these 4 sections and 43 of them submitted a teaching evaluation.

The teaching evaluation included a question about the TAs promptness in returning the assignments. Since the course assignments were returned at the same time in all sections, the average score for each section should be roughly the same, regardless of the gender of the TA. ■

Example 6-4 Digital Analysis for Diagnosing Cancer

In order to diagnose breast cancer, digital images of a small amount of tissue are examined by physicians. Highly specialized experts have been successful at using these images for diagnosis. However, this technique depends heavily on the skill and experience of the physician. In trying to make this diagnostic tool more broadly useful, hundreds of images have been hand-classified by experts and labeled as benign or malignant. In addition, 9 features

have been extracted from each image. Can we use these images to develop an automated classification technique to assist physicians in making a diagnosis?

6.2 The Bootstrap and Inference

A central statistical task is to understand the sampling distribution of a statistic coming from a random sample. For example, if your statistic is the median of a random sample of n items, then you will want to know its standard error (i.e., the standard deviation of the sample mean) to understand how much your results might vary if you were to repeat your sampling procedure (or experiment). In this case, if you assume the underlying population is normally distributed, then the sampling distribution is the t-distribution with $n - 1$ degrees of freedom. However, the sampling distribution of your statistic may not be such a simple formula. In this situation, you can use resampling methods (i.e., methods involving taking samples from the sample) such as the bootstrap.

The idea behind the bootstrap is to approximate the variability of the statistic you compute from your data by computing the statistic on random samples from your data. The bootstrap distribution of a statistic is its distribution over the collection of bootstrap samples. A bootstrap sample is a random (re)sample from the original data. The basic principle is to resample from the sample in the same way that the sample was taken from the population.

In the bootstrap, we draw n observations with replacement from the original data to create a bootstrap sample or resample, and calculate the mean for this resample. We repeat that many times, say 10000. The bootstrap means comprise the bootstrap distribution. The bootstrap distribution is a sampling distribution, for θ_{hat}^* (with sampling from the empirical distribution); we'll talk more below about how it relates to the sampling distribution of θ_{hat} (sampling from the population F). (In the sequel, when we say "sampling distribution" we mean the latter, not the bootstrap distribution, unless noted.)

The sample mean is approximately unbiased for the population mean. spread: The spread of each distribution estimates how much the sample mean varies due to random sampling. The bootstrap standard error is the sample standard deviation of the bootstrap distribution. shape: Each distribution is approximately normally distributed. A quick-and-dirty confidence interval, the bootstrap percentile confidence interval, is the range of the middle 95% of the bootstrap distribution;

6.2.1 Percentile Bootstrap of Median House Price

Our goal is to estimate the median house price for houses in Saratoga County, New York based on a random sample of 1000 houses from the county public records. We can use the median of the sample as an estimate for the median price for the county. However, we typically want to provide more information about our estimator. For example, we may want to provide an estimate of the variability in our estimator, i.e., if we take another sample of 1000 and calculate its median we are likely to get a different value. The standard error of the sample median can give us an idea as to the size of this variability. Or, we may want to provide an interval estimate for the median of all houses rather than a point estimate (i.e., the sample median). To do this, we need to provide a range that reflects the accuracy of our sample in representing the population. The bootstrap can help us with these tasks. First, let's explore the data.

The data are in the file *Saratoga.txt*, and the suffix indicates it is a plain text file. When we examine the contents, we find that the values are tab-separated and the first line contains variable names. We read these data into *R* and perform a few quick checks with

```
require(readr)
sar = read_delim("Saratoga.txt", delim = "\t")
dim(sar)

[1] 1728    16

names(sar)

[1] "Price"          "Lot.Size"        "Waterfront"      ...
[13] "Bedrooms"       "Fireplaces"      "Bathrooms"       "Rooms"

summary(sar$Price)

Min. 1st Qu. Median   Mean 3rd Qu.   Max.
5000 145000 189900 212000 259000 775000
```

From the summary we saw that the median house price is \$189,000 the mean is \$211,967. A histogram of prices appears in Figure 6.1. There we see that the distribution of housing prices is strongly skewed to the right with relatively few houses priced above \$400K. That skewness explains why the mean is higher than the median. The few houses priced between \$400K and \$800K have pulled the mean away from the typical price.

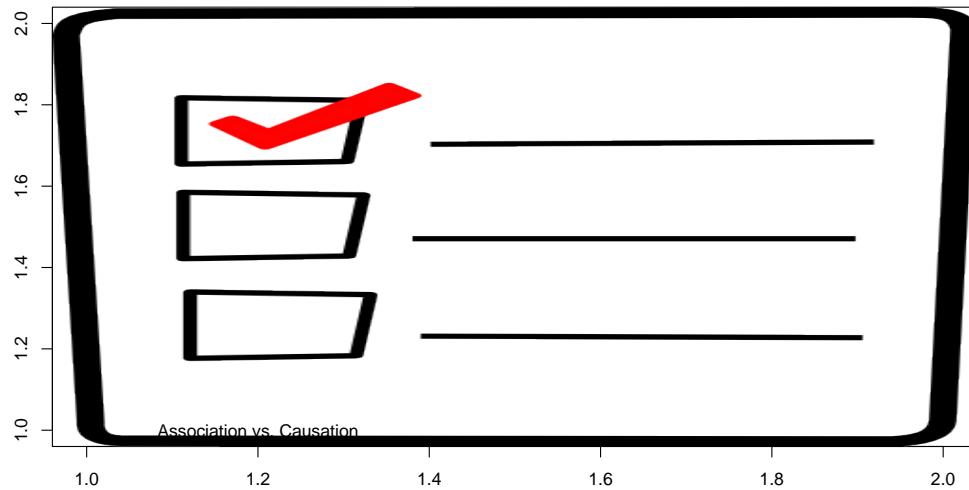


Figure 6.1: Histogram of Housing Prices in Saratoga County. *xx*

Let's get an idea about how resampling works by taking one sample from our data. The

idea is that we treat our sample as if it is the population, which we call the bootstrap population. Since we have a simple random sample, our original sample should be representative of the population so it's reasonable to use this sample as a place holder for the population. Now, to take a sample from this bootstrap population, we draw with replacement. We are assuming that the population is large enough that sampling with replacement from the population is essentially identical to sampling without replacement. Note that if we draw with replacement then the bootstrap sample will be identical to the bootstrap population (and our original sample). We take our sample with

```
bootEx = sample(sar$Price, replace = TRUE)
length(bootEx)
```

```
[1] 1728
```

Our bootstrap sample has a median of 187250, which is close but not the same as the original sample median. The histogram of this bootstrap sample appears on the righthand side of Figure 6.1. It is very similar in appearance to the bootstrap population.

If we repeat this process of taking a bootstrap sample from the bootstrap population and computing the median, then we can examine the distribution of these bootstrap medians. This distribution should resemble the sampling distribution of the sample median (i.e., the distribution of sample medians take from the true population). We compute 100,000 bootstrap sample medians with

```
bootMedians = replicate(100000,
                        median(sample(sar$Price, replace = TRUE)))
```

The histogram of these 100,000 medians appears in Figure 6.2. The spikiness is due to the median relying on a few particular values from the sample. Nonetheless, the quantiles of the bootstrap distribution tend to provide reasonable interval estimates for large sample sizes. For example, 95% of the bootstrap sample medians fall in the following interval:

```
quantile(bootMedians, probs = c(0.025, 0.50, 0.975))
```

```
2.5%    50%   97.5%
185000 189900 195000
```

This interval is called a bootstrap percentile confidence interval. This particular interval is a 95% bootstrap percentile confidence interval for the population median. With small samples, the bootstrap percentile confidence interval tends to be too narrow, meaning it doesn't capture the true population statistic 95% of the time. We consider other formats for bootstrap confidence intervals that have better coverage probabilities in Section 6.2.2. We complete this section by considering a few programming issues.

Suppose that we are interested in estimating other quantiles. In this case, we can wrap up the computation that generates a bootstrap sample and computes a bootstrap quantile, or possibly multiple quantiles. This function, called, say, `bootQuantile()` provides a cleaner call to `replicate()`. We first write `bootQuantile()` for a single quantile, and then consider whether our implementation can work for vector of quantiles.

```
bootQuantile = function(x, p = 0.5)
{
  quantile(sample(x, replace = TRUE), probs = p)
}
```

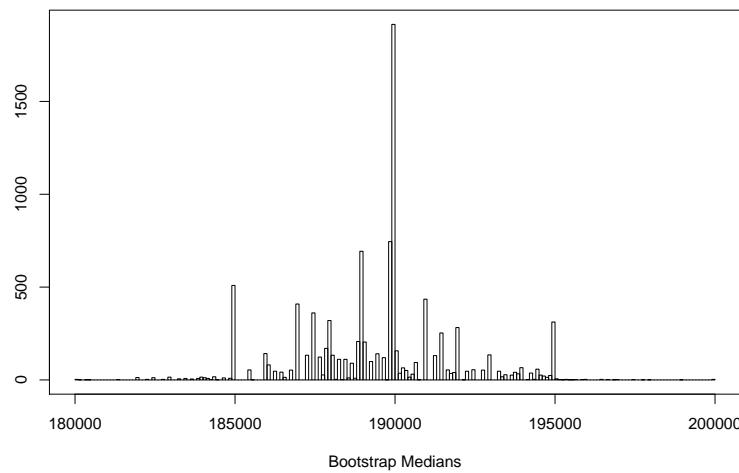


Figure 6.2: Histogram of Bootstrap Medians. *xx*

Since `quantile()` accepts a vector of probabilities, we can calculate several bootstrap quantiles at once so our function is vectorized. For example, we find bootstrap quartiles with

```
bootQuartiles =
  replicate(100000,
    bootQuantile(sar$Price, p = c(0.25, 0.50, 0.75)))
```

We make a 95% bootstrap percentile confidence interval for the lower quartile from these 100,000 bootstrap lower quartiles with

```
quantile(bootQuartiles[1, ], probs = c(0.025, 0.5, 0.975))
```

2.5140000 145000 149000

Notice that we are efficiently using the same bootstrap sample to compute all of the desired quartiles.

If our main focus is on the bootstrap percentile interval, then we can create a wrapper function that computes the bootstrap quantiles and returns the confidence interval only. In this case, we want to parameterize the size of the confidence interval, the number of bootstrap samples, and provide an option to return the bootstrap samples in addition to the confidence intervals. We provide default values for all of these parameters. Our function is as follows:

```
bootPercentileCI =
function(x, p = 0.5, size = 0.95, B = 10000, keepBootStats = FALSE)
{
  bootStats = replicate(B, bootQuantile(x, p = p))
  end = (1 - size) / 2
  if (!is.matrix(bootStats) )
    bootStats = matrix(bootStats, nrow = 1)
  CIs = t(apply(bootStats, 1, quantile,
                probs = c(end, 0.5, 1-end)))
  if (!keepBootStats) return(CIs)
  else return(list(bootStats, CIs))
}
```

We

```
set.seed(124681)
bootPercentileCI(sar$Price, c(0.25, 0.5, 0.75), B = 100000)

 2.5%      50%  97.5%
25% 140000 145000 149000
50% 185000 189900 195000
75% 250000 259000 265000
```

We see that the 95% bootstrap percentile confidence interval for the lower quartile matches the interval that we obtained earlier. Our code appears to be working correctly, but we should develop further test cases to confirm that our code works as expected. In particular, we want to make sure that the code works properly when we provide only one quantile and when we ask for the bootstrap statistics to be returned.

6.2.2 Studentized Bootstrap for Repair Times

The basic concept behind the bootstrap is quite simple. However, applying it can be somewhat of an art. There are several variations on the bootstrap confidence interval, which have been designed to correct for the interval being too narrow and other issues. One proposal is to find the distribution of the standardized statistic, i.e., where we subtract the population statistic from the sample statistics and divide by the standard error of the sampling distribution of the statistic. Then we use its percentiles of this distribution to create the interval.

More concretely, if θ is the population parameter that we are trying to estimate with $\hat{\theta}$, our sample's statistic, then we can base a confidence interval on the distribution of

$$\frac{\hat{\theta} - \theta}{SE(\hat{\theta})}$$

If we know the 2.5 and 97.5 percentile of this distribution (call them $q_{0.025}$ and $q_{0.975}$), then,

$$\begin{aligned} 0.95 &= \mathcal{P}(q_{0.025} \leq \frac{\hat{\theta} - \theta}{SE(\hat{\theta})} \leq q_{0.975}) \\ &= \mathcal{P}(q_{0.025}SE(\hat{\theta}) \leq \hat{\theta} - \theta \leq q_{0.975}SE(\hat{\theta})) \\ &= \mathcal{P}(\hat{\theta} - q_{0.975}SE(\hat{\theta}) \leq \theta \leq \hat{\theta} - q_{0.025}SE(\hat{\theta})) \end{aligned}$$

Then, $(\hat{\theta} - q_{0.975}SE(\hat{\theta}), \hat{\theta} - q_{0.025}SE(\hat{\theta}))$ is a 95% confidence interval for θ .

We saw in Chapter 7 that the sampling distribution of a statistic is often approximately normally distributed. Traditionally, the standard normal quantiles have been used to create a confidence interval for the population parameter. Also, the quantiles of the t -distribution have been used, for when the underlying data follow a normal distribution and the SE of the sample mean is estimated with the $SD(\text{data})/\sqrt{n}$, then the studentized statistic follows the t -distribution. The family of t -distributions is parameterized by the degrees of freedom, and for the studentized mean, this is $n - 1$. The computer allows us to approximate the distribution of a studentized statistic with the bootstrap, and avoid assumptions of normality of the data or of the statistic.

We can use the bootstrap to estimate the distribution of the studentized statistic, i.e., for

each bootstrap sample, we compute the bootstrap statistic, $\hat{\theta}^*$, and the bootstrap standard error of this statistic, $SE^*(\hat{\theta}^*)$ and use these to construct the studentized statistic,

$$\frac{\hat{\theta}^* - \hat{\theta}}{SE(\hat{\theta}^*)}$$

We estimate $q_{0.025}$ and $q_{0.975}$ from these bootstrap replicates. Depending on the form of $\hat{\theta}$, its standard error can be approximated by a simple function of the sample, and consequently, the standard error of the bootstrap statistic, $\hat{\theta}^*$, can be approximated by a simple function of the bootstrap sample. One example is the mean, probability theory tells us that the standard error of the sample mean is σ/\sqrt{n} , where σ is the standard deviation of the population, and this can be approximated by the standard deviation of the sample. This means that we can approximate the standard error of the bootstrap sample mean with SD^*/\sqrt{n} , where SD^* is the standard deviation of the bootstrap sample. Other times, we don't have a simple format for the standard error and we need an alternative method of approximation. We discuss 2 alternatives, bootstrap the bootstrap sample or jackknife the bootstrap sample, via an example.

There are several steps in the process of creating a confidence interval based on bootstrapping the studentized statistics. Before we identify them, we examine data that we are using to help us develop this method. These data consist of repair times for phone lines for one particular time period and class of service. They are available in *ilec.csv*. We examine the contents of this file to determine that the values are separated by commas, as the file extension suggests, and we also see that there are no variable names. We read the data from this file with

```
require(readr)
ilec = read_delim("ilec.csv", delim = ",", col_names = FALSE)
ilec = ilec[[1]]
```

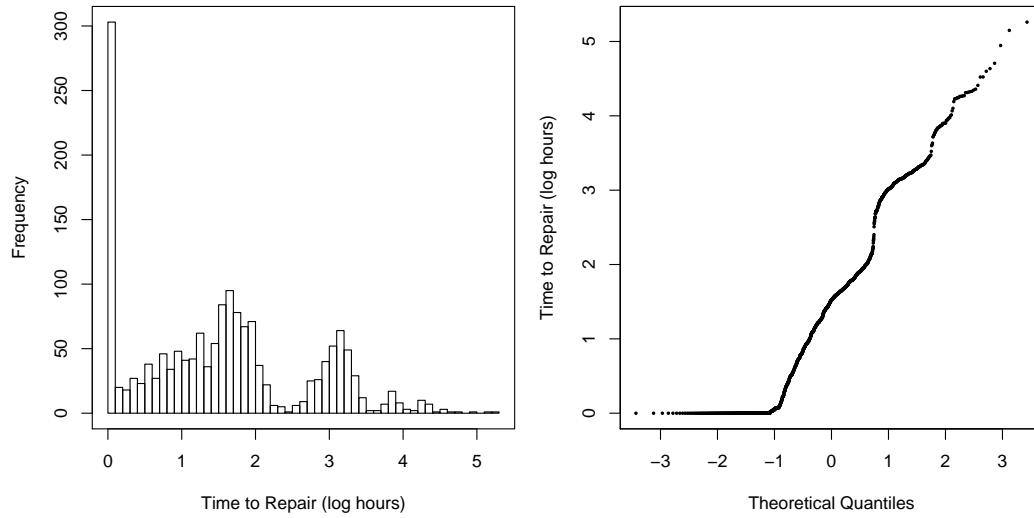


Figure 6.3: Histogram and Normal Quantile Plot of Log Waiting Times. *xx*

Before making any estimates of average repair time, we examine the data values. Figure 6.3 shows a histogram and normal quantile plot of the logarithm of repair time. We see

that the data are highly skewed right; there is a spike at 0 indicating that many repairs happen immediately; we see two main modes with the second mode at $\log(1 + 24)$, which confirms the observation that most repairs occur during the regular work day. Given the exceptionally long tail, let's consider making an estimate of the average time to repair for 95% of the customers, i.e., the average after trimming the top 5% of values. This requires us to create our own function to compute what we call the ‘upper 5% trimmed mean’.

Before we begin our programming tasks, let's examine the bootstrap distribution of the studentized sample mean for our sample of repair times. Given that the sample size is well over 1500, we imagine that the Central Limit Theorem (CLT, [?]) will have taken effect and the distribution of the normalized sample mean will be very close to the normal.

```
meanIlec = mean(ilec)
n = length(ilec)
bootStats =
  replicate(100000, {
    bootSample = sample(ilec, replace = TRUE)
    (mean(bootSample) - meanIlec) / (sd(bootSample)/sqrt(n))
  })
```

The curvature in the normal quantile plot in Figure 6.4 reveals that the bootstrap distribution has a longer left tail and a shorter right tail than the normal. Since the confidence interval uses the 2.5% and 97.5% quantiles of the distribution, this deviation is problematic. We compute the proportion of bootstrap sample statistics below and above the normal 2.5% and 97.5% quantiles with

```
bootCDF = ecdf(bootStudMean)
bootCDF(c(qnorm(0.025), qnorm(0.975)))
[1] 0.03612 0.98300
```

Our empirical quantiles confirm the problem: 3.6% of the bootstrap statistics are below -1.96 (the 2.5 percentile of the standard normal) and 1.7% are above 1.96 . Together 5.3% is close to 5%, but unfortunately, we do not have 2.5% in each tail.

Identify the Tasks

Now that we have explored how to compute standardized bootstrap means for the repair time data, we can identify the general tasks associated with this confidence interval procedure. These are:

- Take a bootstrap sample.
- Compute bootstrap statistics, e.g., mean and SE, from the bootstrap sample.
- Standardize the bootstrap statistic.
- Find quantiles of the standardized bootstrap statistic.
- Create a confidence interval for the population parameter.

Although we have identified 5 separate tasks, once we begin writing code, we may find it convenient to wrap some of these tasks into one function, and we may uncover additional tasks within 1 task that we want to split out. Task identification is a fluid process.

In general, we have seen that taking a bootstrap sample is as simple as

```
bootSample = sample(data, replace = TRUE)
```

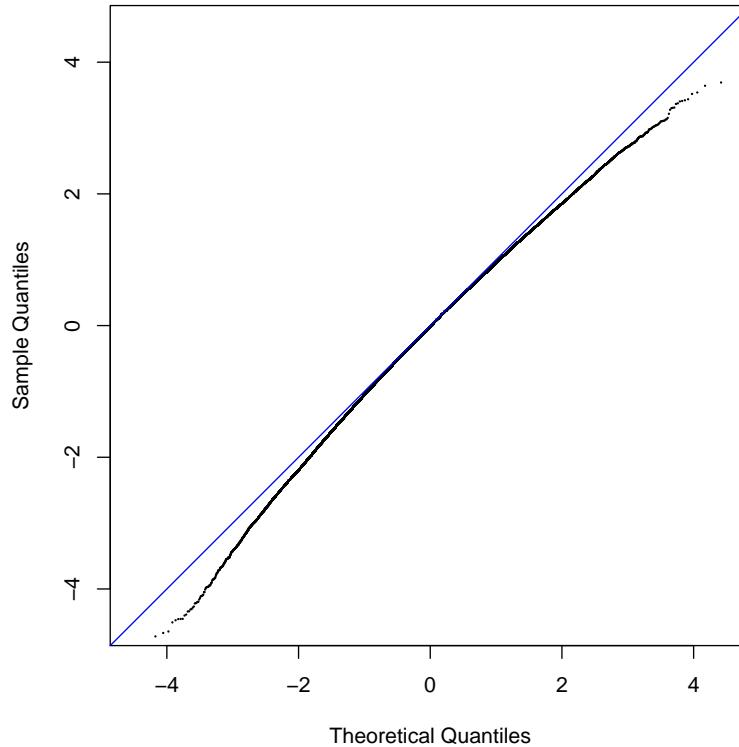


Figure 6.4: Normal Quantile Plot of Bootstrap Studentized Mean Waiting Time. *xx*

where `data` contain the original sample. Rather than wrap this 1 line of code into a function, let's keep it as is for now and tackle the next task: computing the summary statistic and its standard error for the bootstrap sample.

Before proceeding with our tasks, we need to be able to compute the mean of the specially trimmed data. We can imagine wanting to trim different amounts of data and wanting to trim from either the upper or the lower tail so we include parameters for these in our function definition. Our `trimMean()` function follows

```
trimMean = function(x, trim = 0.05, upper = TRUE)
{
  m = length(x) * trim
  mean(sort(x, decreasing = upper) [-(1:m)])
}
```

A Suite of Test Cases for Code

A simple test case to confirm that our function works correctly uses the integers from 1 to 20. We confirm the trimming works as expected for $1, 2, \dots, 20$ for various amounts of trimming and both the upper and lower tails with the following function calls:

```
trimMean(1:20)
trimMean(1:20, trim = 0.07)
trimMean(1:20, trim = 0.10)
trimMean(1:20, upper = FALSE)
trimMean(1:20, trim = 0.10, upper = FALSE)
```

The return values are 10, 10, 9.5, 11, and 11.5, respectively. Rather than have to individually check the return value from each of these function calls every time we want to test our code, we can set up a programmatic check of our test suite with the `test_that()` function in the `testthat` package. We compare each of these function calls to what we expect with

```
test_that("trimMean trims the proper amount", {
  expect_identical(trimMean(1:20), 10)
  expect_identical(trimMean(1:20, trim = 0), 10.5)
  expect_identical(trimMean(1:20, trim = 0.07), mean(1:18))
  expect_identical(trimMean(1:20, trim = 0.10), mean(1:18))
  expect_identical(trimMean(1:20, upper = FALSE), mean(2:25))
  expect_identical(trimMean(1:20, trim = 0.10, upper = FALSE),
                   mean(3:20))
}
)
```

In these tests, two errors were found by `test_that()`:

```
Error: Test failed: 'trimMean trims the proper amount'
* trimMean(1:20, trim = 0) not identical to 10.5.
1/1 mismatches
[1] 10 - 10.5 == -0.5

* trimMean(1:20, trim = 0.07) not identical to mean(1:18).
1/1 mismatches
[1] 10 - 9.5 == 0.5
```

In this case, we have identified two errors. The second error points to the need for clarification with our function. When `trim = 0.07`, we trim only 5% from the data, not 7%, because 7% does not correspond to an integer number of elements of our data. To make this ‘rounding’ more explicit, we may want to use `floor()` in our function when computing the number of elements to drop from `x`. Of course, we can use `ceiling()` rather than `floor()`, if we want to round up instead of down.

The first error is a bit more subtle. When `trim = 0`, then `m` is 0. The expression `-(1:m)` becomes `-(1:0)` so the first element is dropped. We can add a check for this issue in our code. Other trim amounts may lead to `m` being 0, e.g., any value of `trim` less than 0.05 will yield 0 for `m` for our test case with 20 data values. Our revised function is

```
trimMean = function(x, trim = 0.05, upper = TRUE)
{
  m = length(x) * trim
  if (m == 0) {
    mean(x)
  } else {
    mean(sort(x, decreasing = upper) [-(1:m)])
  }
}
```

With this revised function and an update to `test_that()` that addresses the issue with 7% trim, we receive no errors in our tests. As our code continues to develop, we can add more test cases to the `test_that()` call, and as we develop additional functions, we can set up tests for them. For example, now that we have identified 0 as a special case for trimming, that

leads us to think whether we need to check the inputs to ensure that `trim` is nonnegative. We leave this to the exercises.

The ... Parameter

Our second task is to compute the bootstrap statistic from our bootstrap sample. Now that we have `trimMean()` written and debugged, we can carry out this task. We want our code to be flexible so that we can, say, specify the amount of the trimming. We can do this with the `...` parameter that allows the caller to specify additional parameters that can be passed on to a function calls within the function. For example, we can create a `bootStat()` function to compute the bootstrap samples and their statistics with

```
bootStat = function(data, R = 1000, ...)
{
  sapply(1:R, function(idx, ...) {
    bootSample = sample(data, replace = TRUE)
    trimMean(bootSample, ...)
  }, ...)
}
```

Notice that we have used `...` three times in our function. These are all within the call to `sapply()`. In this `sapply()`, we are applying an anonymous function to each element of the vector of integers from 1 to `R`. Recall that in order to pass additional arguments to the function being called in `sapply()`, we need to provide them as arguments to `sapply()` following the function. This explains the third argument to `sapply()`. Our anonymous function needs `...` as an argument in order to pass the supplied arguments on to `trimMean()`, which the call to `trimMean()` passes on to the function. To help clarify the use of `...` we also provide a version of `bootStat()` that employs a `for` loop rather than an apply:

```
bootStatAlt = function(data, R = 1000, ...)
{
  stats = vector(mode = "numeric", length = R)
  for (i in 1:R) {
    bootSample = sample(data, replace = TRUE)
    stats[i] = trimMean(bootSample, ...)
  }
  return(stats)
}
```

This simplifies the use of `...` but requires us to set up the return vector in advance, assign each bootstrap statistic into it as it is evaluated, and return the completed set of bootstrap statistics.

We can compare the run time of these two alternatives. We set the seed for the random number generator before running each experiment so that the same random sets of data are used in the two tests, and we call each bootstrapping function 100 times to get better comparison figures.

```
set.seed(124612)
system.time(replicate(100, bootStat(rnorm(200), trim = 0.1)))

  user  system elapsed
8.886   0.383   9.555
```

```
set.seed(124612)
system.time(replicate(100, bootStatAlt(rnorm(200), trim = 0.1)))

  user  system elapsed
 8.894   0.357   9.552
```

We find that these two versions of the code are essentially the same in terms of run time. Efficiency gains from avoiding `for` loops occur when we replace them with vectorized calculations, not with an `sapply()`.

A Function as an Input to a Function

Let's generalize our `bootStat()` function to bootstrap any statistic provided by the caller. To do this, we simply add a parameter to `bootStat()` where the caller can provide a function to compute the desired statistic. We can either require this function to be supplied or provide a default function, such as `mean()`. We will not supply a default function for the statistic, and our revised `bootStat()` follows:

```
bootStat = function(data, R = 1000, statFun, ...)
{
  stats = vector(mode = "numeric", length = R)
  for (i in 1:R) {
    bootSample = sample(data, replace = TRUE)
    stats[i] = statFun(bootSample, ...)
  }
  return(stats)
}
```

This very small change to our function gives us a lot of flexibility in bootstrapping the sampling distribution of a statistic.

Let's test our `bootStat()` with a few test cases and `test_that()`. Given that we have a random component to our bootstrapping, we need to design our test cases carefully. We can for example, provide a data set of identical values so each bootstrap sample looks exactly like the bootstrap population. We can also add an exceptionally large outlier to this set of identical values where we are quite certain the outliers be trimmed from the bootstrap sample. Another possibility is to set the seed of the random number generator and examine the bootstrap samples produced. We have to be careful with this approach because any small change in code that uses random number generator can impact our result and so not produce what is expected. Of course, we can also accept a range of return values. We try all of these approaches with

```
test_that("bootStrap calls trimMean and trims the proper amount",
{
  expect_identical(bootStat(rep(20, 25), R = 10,
                            trimMean, trim = 0.05),
                  rep(20, 10))
  expect_identical(bootStat(c(rep(1, 199), 10000), R = 10,
                            trimMean, trim = 0.05),
                  rep(1, 10))
  expect_true(median(abs(bootStat(c(rep(0, 80), rep(1, 100)),
                                 R = 100, trimMean, trim = 0.20) -
                        rep(0.5, 100))) < 0.1)
  expect_identical({
    set.seed(111)
```

```

    bootStat(1:20, R = 1, trimMean, trim = 0.05)
  },
  mean(c(1, 1, 2, 2, 4, 4, 7, 8, 8, 9, 9, 9,
        11, 11, 12, 12, 12, 13, 15))
)
}
)

```

Our code clears all of these tests.

Function Place Holders

Let's take stock of the progress that we have made with our tasks. We have completed two tasks: take a bootstrap sample and compute bootstrap statistics. We also need to studentize the bootstrap statistic. Then, we find quantiles of the standardized bootstrap statistic. and use them to create a confidence interval. We can accomplish the standardization by passing `bootStat()` a function which computes the bootstrap statistic and studentizes it. Or we can modify `bootStat()` to accept 2 functions, one to compute the statistic and the other to compute the SE of the sampling distribution of the statistic. There are pluses and minuses to both approaches. Either way we want to create a function that computes an estimate of the SE of the sampling distribution of the upper trimmed mean. This is the topic of [?]. In the mean time, we create a place-holder function that simply returns $1/\sqrt{n}$ for any input, e.g.,

```

jkSE = function(data, statFun, ...)
{
  return(1/ sqrt(length(data)))
}

```

We often create draft functions like this because we don't always want to develop code in a specific sequence, and these place holders help us get around this issue. We can call this function with the desired input specifications and it returns something that meets the general specification for the function so that we can continue the development of functions that depend on it.

Constructing a Function Call from Inputs

We take a general approach and extend `bootStat()` to accept a list of functions and inputs so that the function computes multiple statistics on a single bootstrap sample. Then `bootStat()` returns a matrix of bootstrap statistics, with each column corresponding to the statistics for one of the input functions. When we take this approach, we no longer use ... because we want the flexibility of passing different arguments to the various input functions. We modify `bootStat()` as follows:

```

bootStat = function(data, statFun, R = 1000, args.stat = NULL)
{
  stats = matrix(nrow = R, ncol = length(statFun))
  for (i in 1:R) {
    bootSample = sample(data, replace = TRUE)
    stats[i, ] = sapply(1:length(statFun), function(j) {
      do.call(statFun[[j]], c(list(bootSample), args.stat[[j]])))
    })
  }
  return(stats)
}

```

The `do.call()` function constructs a function call from the function and the list of arguments provided. Note that we provide the bootstrap sample as the first argument to the call. The rest of the arguments are passed into `bootStat()` via the `args.stats` parameter. If no additional arguments are provided, then `c(list(bootSample), args.stat[[j]])` is a list with 1 element – the data vector.

Let's try invoking our revised function with a few simple cases

```
bootStat(1:20, R = 10,
         statFun = list(trimMean, jkSE),
         args.stat = list(list(trim = 0.1)) )
```

Error in args.stat[[j]] : subscript out of bounds

Although we do not have any additional arguments to pass to `jkSE()`, we need to supply a NULL list to indicate this, i.e.,

```
bootStat(1:20, R = 5,
         statFun = list(trimMean, jkSE),
         args.stat = list(list(trim = 0.1), NULL) )

[,1]      [,2]
[1,] 10.111111 0.2236068
[2,]  9.000000 0.2236068
[3,]  6.777778 0.2236068
[4,]  9.555556 0.2236068
[5,]  7.500000 0.2236068
```

As expected, the values in the second column are all $1/\sqrt{20}$. Let's try bootstrapping only 1 statistic with no inputs, e.g.,

```
bootStat(1:20, statFun = mean, R = 5)
```

We get an error

```
Error in statistic[[j]] : object of type 'closure'
is not subsettable
```

When we construct our function call, we assume that `statFun` is a list. Here we have supplied a function for `statFun` and the error message indicates that we cannot take a subset of a function. If we have only one statistic to be bootstrapped, it seems reasonable to not place it in a list. Also, in this situation, it seems reasonable to return a vector of bootstrapped statistics, rather than a matrix with 1 column.

We can modify `bootStat()` to allow for these possibilities.

```
bootStat = function(data, statFun, R = 1000, args.stat = NULL)
{
  if (!is.list(statFun)) {
    statFun = list(statFun)
    if (length(args.stat) > 1) args.stat = list(args.stat)
  }
  stats = matrix(nrow = R, ncol = length(statFun))
  for (i in 1:R) {
    bootSample = sample(data, replace = TRUE)
```

```

stats[i, ] = sapply(1:length(statFun), function(j) {
  do.call(statFun[[j]], c(list(bootSample), args.stat[[j]])))
})
}
if (ncol(stats) == 1) return(as.numeric(stats))
return(stats)
}

```

The following tests seem to indicate that our code is working as expected.

```

bootStat(1:20, trimMean, R = 10)

[1] 10.2  9.3  9.1 10.3 10.3 11.9  6.6  9.6 11.5  8.8

 testData = c(1:20, NA, NA, NA)
bootStat(testData,
         statFun = list(mean, function(x, n, ...) {sd(x, ...)/sqrt(n)}),
         R = 10,
         args.stat = list(list(trim = 0.2, na.rm = TRUE),
                         list(n = length(testData), na.rm = TRUE)) )

 [,1] [,2]
[1,] 10.8 0.99
[2,] 10.6 1.24
[3,]  8.8 1.26
[4,]  9.7 1.39
[5,]  8.4 1.16
[6,] 10.4 1.29
[7,] 11.5 0.92
[8,] 10.3 1.32
[9,] 11.5 1.25
[10,] 11.1 1.27

```

Notice that we have supplied an anonymous function as one of 2 functions to `statFun`. This anonymous function has ... as an argument and the these additional parameters are passed to `sd()`.

We complete our final 3 tasks with one function that creates the studentized version of the bootstrap statistic, finds the appropriate quantiles of this empirical bootstrap distribution, and constructs a confidence interval. We provide as inputs, the bootstrap statistics, the estimates of the SE of the bootstrap statistics, the level of confidence, and the statistic and SE of the original sample. All of these parameters are required, except for the level which we set to 95%. Our function is simply defined as

```

studentBootCI =
  function(sampleStat, se, bootStat, bootStatSE, level = 0.95)
{
  bootSU = (bootStat - sampleStat) / bootStatSE
  bootCDF = ecdf(bootSU)
  end = (1 - level) / 2
  qs = quantile(bootSU, probs = c(end, 1 - end))
  return(c(sampleStat - qs[2] * se, sampleStat - qs[1] * se))
}

```

With the exception of completing the function to estimate the SE of the sampling distribution of the bootstrap statistic, we have completed our tasks.

To test our functions, let's find a studentized bootstrap confidence interval for the mean repair time.

```
iLECBootStats =
  bootStat(iLEC, R = 100000,
            statFun = list(mean,
                           function(x, n, ...) {sd(x, ...)/sqrt(n)}),
            args.stat = list(NULL, list(n = length(iLEC)))) )

iLECBootCI = studentBootCI(mean(iLEC), sd(iLEC)/sqrt(length(iLEC)),
                            bootStat = iLECBootStats[ , 1],
                            bootStatSE = iLECBootStats[ , 2])

iLECBootCI
```

7.757 9.185

Notice that the interval is not symmetric about the sample mean because the sampling distribution of the bootstrap statistic is not symmetric, i.e.,

```
iLECBootCI - mean(iLEC)
```

-0.655 0.774

In the next section, we complete `jkSE()` to compute an estimate of the SE of a sampling distribution using the jackknife method. T

6.2.3 Jackknife Estimate of the Standard Error for Average Repair Time

The Jackknife is a resampling procedure that is computationally simpler than the bootstrap and is particularly useful in estimating standard deviations. Rather than resample from the sample and work with bootstrap samples the same size as the original sample, the delete-1 version of the jackknife drops 1 observation from the sample, computes the statistic from the remaining observations, and uses the variability between the leave-one-out statistics and the original statistic based on the entire sample to estimate the standard error of the sampling distribution of the original statistic. More specifically, suppose we collect a random sample, and use this sample to create the statistic, $\hat{\theta}$, to estimate the population parameter, θ . Then, the jackknife estimate of the standard error of $\hat{\theta}$ is

$$\sqrt{\frac{n-1}{n} \sum_{j=1}^n (\hat{\theta}_{-j} - \hat{\theta})^2},$$

where $\hat{\theta}_{-j}$ is the statistic computed from all but the j^{th} observation.

In our example of repair times, we want to estimate the SE of the bootstrap upper trimmed mean. As mentioned earlier, we want to write a general function to create the jackknife estimator for an arbitrary statistic. Previously, we identified three inputs: the sample, in our case this can be one of the bootstrap samples as well as the original sample; the function that computes the statistic from the sample; and any additional parameters that this function requires. Now that we have the theoretical form of the jackknife variance, we can fill in the body of our function:

```
jkSE = function(data, statFun, ...)
{
  n = length(data)
  sampleStat = statFun(data, ...)
  jackStats = sapply(1:n, function(j) statFun(data[-j], ...))

  return(sqrt(n - 1 / n * sum((jackStats - sampleStat)^2)))
}
```

Let's set up a very simple test case where it's easy to calculate the statistic when we drop one observation from the sample. We take our sample to be `xs = c(0, 1, 1, 1, 2)` and our statistic to be the mean. The mean of our sample is 1, and the 5 means when we drop out each observation in turn are 1.25, 1, 1, 1, 0.75. It's easy to compute the squared differences: 3 of them are 0 and the other 2 are 0.0625. Then since `n-1` is 4, the jackknife variance is `0.8 * 0.125` or 0.1 and the SE is the square-root of 0.1. When we test `jkSE()` with these data and the mean statistic, we get

```
jkSE(xs, mean)
```

```
[1] 2.230
```

Clearly, there is something wrong with our function. We can examine our code to find a simple mistake that we made, but let's instead step through the code one line at a time to check it.

Debugging by Stepping Through Code within a Function Call

We can examine the values of variables within a function call and invoke other expressions with the `browser()` function. If we place the expression `browser()` in our function, then when this call is invoked, execution of the code in the function is paused and we gain access to the *R* interpreter. We can examine the variables in the call frame, invoke code that we write at the command line, and step through the subsequent lines of code in the function. Let's add a call to `browser()` in `jkSE()` as the first line of function, i.e.,

```
jkSE = function(data, statFun, ...)
{
  browser()
  n = length(data)
  sampleStat = statFun(data, ...)
  jackStats = sapply(1:n, function(j) statFun(data[-j], ...))
  return(sqrt(n - 1 / n * sum((jackStats - sampleStat)^2)))
}
```

Now, when we invoke `jkSE()` we are immediately placed in browser mode with

```
jkSE2(xs, mean)
```

```
Called from: jkSE2(xs, mean)
```

```
Browse[1]>
```

At the browser's prompt we can execute an expression. For example, we can ask for a list of the objects in the environment with

```
Browse[1]> ls()  
[1] "data"      "statFun"
```

```
Browse[1]>
```

We can also tell the browser to proceed to the next statement, with the single letter command `n`, e.g.,

```
Browse[1]> n  
debug at #4: n = length(data)  
  
Browse[2]> n  
debug at #5: sampleStat = statFun(data, ...)
```

```
Browse[2]> ls()  
[1] "data"      "n"          "statFun"  
  
Browse[2]> print(n)  
[1] 5
```

The browser is not between the 4th and 5th lines of code; that is, the 4th line has been executed and the 5th line will be executed when we issue the command `n`. At this point, we see that the variable `n` has been created and we check that its value is 5. We can continue in this fashion, checking the changes after each line of code. Here we proceed to the point immediately prior to the `return()` and check the values of the intermediate variables with

```
Browse[2]> n  
debug at #6: jackStats = sapply(1:n, function(j) statFun(data[-j], ...))  
  
Browse[2]> n  
debug at #7: return(sqrt(n - 1/n * sum((jackStats - sampleStat)^2)))  
  
Browse[2]> ls()  
[1] "data"      "jackStats"  "n"          "sampleStat" "statFun"  
  
Browse[2]> jackStats  
[1] 1.25 1.00 1.00 1.00 0.75  
  
Browse[2]> sampleStat  
[1] 1
```

```
Browse[2]> sum((jackStats - sampleStat)^2)
[1] 0.125
```

```
Browse[2]> c
[1] 2.230
```

We see that all of our calculations so far are correct. And, we now notice that we made a very simple mistake with the order of operation in the expression `n - 1 / n * ...!` What we really meant was `(n - 1) / n *` This is a simple fix.

The `browser()` function allows us to enter the call frame of our function and poke around to see if all of the variables have their expected values and if snippets of code are running as planned. In addition to the command `n` to proceed to evaluate the next statement, other commands recognized by `browser()` include `c` to exit the browser and `f` to finish the current loop or function. Additionally, we can set an option through `options()` so that we always enter browser-mode when an error occurs, e.g., `options(error = recover)` turns this mode on and `options(error = NULL)` turns off any error recovery.

Our revised function appears as

```
jkSE = function(data, statFun, ...)
{
  n = length(data)
  sampleStat = statFun(data, ...)
  jackStats = sapply(1:n, function(j) statFun(data[-j], ...))

  return(sqrt((n - 1) / n * sum((jackStats - sampleStat)^2)))
}
```

The tests now provide the expected return values, e.g.,

```
all( c( identical(jkSE(rep(1, 20), mean), 0),
       { xs = c(0, 1, 1, 1, 2)
         identical(jkSE(xs, mean), sqrt(0.1)) }))

[1] TRUE
```

Here we have written our own tests and checks rather than use the `test_that()` and `expect_identical()` functions.

We complete this section by creating a 95% studentized bootstrap confidence interval for the 1% upper trimmed mean. Our mean function is `trimMean()` with `trim` set to 0.01. Our SE function is `jkSE()` with the `statFun` argument set to `trimMean()` and we pass this function `trim` is passed via the `...` argument.

```
ilecBootStats =
  bootStat(ilec, R = 10000,
            statFun = list(trimMean, jkSE),
            args.stat = list(list(trim = 0.01),
                            list(statFun = trimMean, trim = 0.01)) )

studentBootCI(trimMean(ilec, trim = 0.01),
              jkSE(ilec, trimMean, trim = 0.01),
              bootStat = ilecBootStats[ , 1],
              bootStatSE = ilecBootStats[ , 2])
```

6.927 8.170

This confidence interval for the 1% upper trimmed mean is about 1 hour lower than the interval for the mean. This interval is also shorter.

Unfortunately, it took about 20 minutes to run 10,000 bootstrap replications for our trimmed mean. One reason for this is that to jackknife the standard error for each bootstrap sample, we call `trimMean()` 1664 times to compute the trimmed mean with one observation dropped. Each time, `trimMean()` is called, the data are sorted and the average is taken with the same data less one observation. There are clearly faster ways to do this. For example, we can add a parameter to `trimMean()` to indicate whether or not the data are sorted. However, this is a situation where it makes sense to depart from writing code in a very general fashion and develop a jackknife specifically for the trimmed mean.

With a little algebra we can find a very simple calculation for the leave-one-out trimmed mean. Suppose we have n data values, x_1, \dots, x_n and that we are trimming the m largest before taking the mean. If the data values are arranged in decreasing ordered, i.e., x_1 is the largest and x_n the smallest, then the upper trimmed mean can be written as

$$\tilde{x} = \frac{1}{n-m} \sum_{j=m+1}^n x_j.$$

We can express the leave-one-out trimmed mean, \tilde{x}_{-i} , in terms of \tilde{x} and x_i . That is, for $i = m+1, \dots, n$,

$$\tilde{x}_{-i} = \frac{1}{n-m-1} \sum_{j=m+1}^n x_j - \frac{1}{n-m-1} x_i)$$

and

$$\tilde{x}_{-i} - \tilde{x} = \frac{1}{n-m-1} (\tilde{x} - x_i),$$

And, for $i = 1, \dots, m$, we have $\tilde{x}_{-i} = \tilde{x}_{-(m+1)}$. Note that m depends on n and the trim proportion, and we are assuming that m remains the same when we drop one observation out. This is the case for our example with 1664 and 1663 observations and a trim of 0.01. Both result in 16 observations being trimmed. We can use this representation to more efficiently compute the jackknife estimate of the standard error.

Our specialized function appears as,

```
jkTrimSE = function(data, trim = 0.05, upper = TRUE)
{
  n = length(data)
  # Assume n - 1 and n produce the same m
  m = floor(length(data) * trim)
  dataSort = sort(data, decreasing = upper)
  sampleStat = mean(dataSort[-(1:m)])
  dataSort[1:m] = dataSort[m + 1]
  leave1outDiffs = (sampleStat - dataSort) / (n - m - 1)
  return(sqrt((n - 1) / n * sum(leave1outDiffs^2)))
}
```

Notice that we use vectorized calculations to compute the difference between the leave-one-out trimmed mean and the original trimmed mean, and we sort the data once, not n times.

Equality in Comparing Values

One way to check that our revised function works correctly is to compare the estimates of the SE for the 1% trimmed mean of the `ilec` data obtained from `jkTrimSE()` and the more general version `jkSE()`. We can do this with

```
identical(jkTrimSE(ilec, trim = 0.01),
          jkSE(ilec, trimMean, trim = 0.01))
[1] FALSE
```

It appears that the two functions produce different results. When we print the contents of the two function calls to the console we find that both return

```
[1] 0.31358402
```

What's going on? Recall that the value printed is not necessarily the same as the value returned by the function or stored in the variable. We have limited the number of digits printed. We can print more digits of the result to uncover the difference:

```
print(jkTrimSE(ilec, trim = 0.01), digits = 20)
[1] 0.31358402219802034017

print(jkSE(ilec, trimMean, trim = 0.01), digits = 20)
[1] 0.31358402219801773114
```

The two results are indeed different, but they are identical to the first 13 places past the decimal point. This observation raises 2 questions: Why are they different? Are they close enough for our purposes? These values are different because the operations used to compute these quantities are not identical, and the computer is a finite-state machine, meaning that it stores numbers to a certain fixed level of precision. The intermediate computations lead to possibly different intermediate values which can yield slightly different final values. For our purposes, having agreement to say the 6th decimal place seems more than adequate. We can compare the return values from our 2 function calls with this tolerance in mind with

```
abs(jkTrimSE(ilec, trim = 0.01) -
    jkSE(ilec, trimMean, trim = 0.01)) < 0.000001
[1] TRUE
```

When we write tests for our code, we need to keep this limitation in mind.

With this more efficient version of the jackknife, we can make 100,000 studentized bootstrap calculations in only a few minutes. The interval estimate of the average repair time for 99% of the customers matches the previous interval obtained with 10,000 bootstraps to 3 decimal places, i.e.,

```
ilecBootStatsFast =
  bootStat(ilec, R = 100000,
            statFun = list(trimMean, jkTrimSE),
            args.stat = list(list(trim = 0.01),
                            list(trim = 0.01)) )

studentBootCI(trimMean(ilec, trim = 0.01),
              jkTrimSE(ilec, trim = 0.01),
              bootStat = ilecBootStatsFast[, 1],
              bootStatSE = ilecBootStatsFast[, 2])
[1] 6.939 8.172
```

If we had the computational power to run the original bootstrap 100,000 times, the results would be closer; even so they are close enough given the accuracy of the measurements. The revised bootstrapping took a fraction of the time. We can use `system.time()` to compare these 2 approaches with 1000 bootstrap samples,

```
set.seed(135531)
tfast = system.time(
{bootStat(ilec, R = 1000,
          statFun = list(trimMean, jkTrimSE),
          args.stat = list(list(trim = 0.01),
                            list(trim = 0.01)) )})
set.seed(135531)
tslow = system.time(
{bootStat(ilec, R = 1000,
          statFun = list(trimMean, jkSE),
          args.stat = list(list(trim = 0.01),
                            list(statFun = trimMean, trim = 0.01))))}
```

Notice that we set the seed to the random number generator to the same value before starting each timing. This way the same bootstrap samples will be generated. We find the revised jackknife function is more than 600 times faster. That is, the ratio of the original to the revised times is

user	system	elapsed
640.40	588.26	623.46

To make this comparison more concrete, when we ran 100,000 bootstraps with the specialized jackknife, it took about 1 minute to complete on our laptop. To run 100,000 bootstraps with our original approach takes more than 10 hours!

6.3 Permutations and Testing for a Gender Effect

Permutations of the data provide a versatile mechanism to test hypotheses. While permutation tests are not new to the field of statistics, they have become more prevalent with increased computational capabilities. The basic idea behind permutation tests goes as follows:

- Consider the problem/question of interest, the design of the experiment, and the nature of the data collected. Our understanding of the process that led to the data guides us through the development of the test.
- Select a test statistic or metric that measures the effect of interested, e.g., large values indicate a difference in the centers of the distribution for two groups.
- Compute the test statistic for the data collected, i.e., the original sample.
- If there is no effect, then the data can be scrambled (permuted), and the shuffled data should look like the real data. Under this assumption of no effect, the test statistic computed from the scrambled data has the same chance of occurring as the statistic from the original set of data. With this reasoning, we find the permutation distribution of the test statistic. We can either compute all possible permutations or estimate the distribution by re-sampling.

- Compare the value of the test statistic calculated from the original data to the sampling distribution. If the observed test statistic is a surprise, i.e., so large that a value as large or larger is rarely observed, then this is evidence against the hypothesis of no effect.

Permutation tests give a simple approach for carrying out an hypothesis test where we can use simulation to compute the sampling distribution of the test statistic. However, the permutation of the data values needs to be developed carefully to reflect the design and the effect that we are interested in measuring. When we scramble the data, we work conditionally on the values that were observed and randomize the labels, e.g., the group that the data value belongs to.

To make these ideas more concrete, let's examine the experiment described in Q.6-3 (page 246). The aim of this experiment was to study how perceived instructor gender affects student evaluations of teaching. To do this, students in an online course were randomly assigned to one of 6 sections. We ignore the sections taught by the instructor and focus on the 4 sections led by the 2 TAs (one female and one male). Each TA taught one section under his/her real name and one section using the name of the other TA. It seems reasonable to assume that if there is no gender effect, then a students' evaluation of a TA should be the same whether the TA is perceived to be male or female. So we can shuffle the students assigned to the 2 sections led by the female TA between these 2 sections similarly randomize the students in the 2 sections led by the male TA, and we should see little difference in evaluations between the perceived male and female TAs.

Before we start randomizing students in sections, let's examine the data so that we have a better sense of the nature of the information collected. As mentioned earlier, we study only one aspect of evaluations—the promptness of the TA in returning assignments. The data are available in the file *Macnell-RatingsData.csv*. As the filename extension suggests, the contents of the file is plain text and the data values are separated by commas. We can confirm this by viewing the file with a plain text editor, and when we do, we find that the first line of the file contains variable names. We read the file and examine a few variables with

```
require(readr)
set = read_delim("Macnell-RatingsData.csv", delim = ",")
dim(set)

[1] 47 20

names(set)

[1] "group"          "professional"   "respect"        ...
[9] "prompt"          "consistent"    "fair"           ...
[17] "gender"          "age"          "tagender"       "taidgender"

with(set, table(tagender, taidgender))

taidgender
tagender 0 1
0 11 12
1 13 11
```

By checking against the published counts, we find that the TAs gender and perceived gender are both reported as 0 for female and 1 for male. We can convert these variables to factors with

```
set$tagender = factor(set$tagender, levels = 0:1,
                      labels = c("female", "male"))
set$taidgender = factor(set$taidgender, levels = 0:1,
                       labels = c("id_female", "id_male"))
```

This makes it easier to examine the data.

For example, we can compare the scores for TA promptness for the different groups of students. In Figure 6.5, the first 2 columns of scores are for the female TA with the column on the left corresponding to the group of students who perceived the TA as female. The next 2 columns are the scores for the male TA, and those in the left column are for the TA identified as female. The pattern in the data is striking. We see that all low scores of 1 and 2 went to the TA whom the students thought was female, whether or not the TA was actually female, and all but one of the scores of 3 went to the TAs who were perceived to be female.

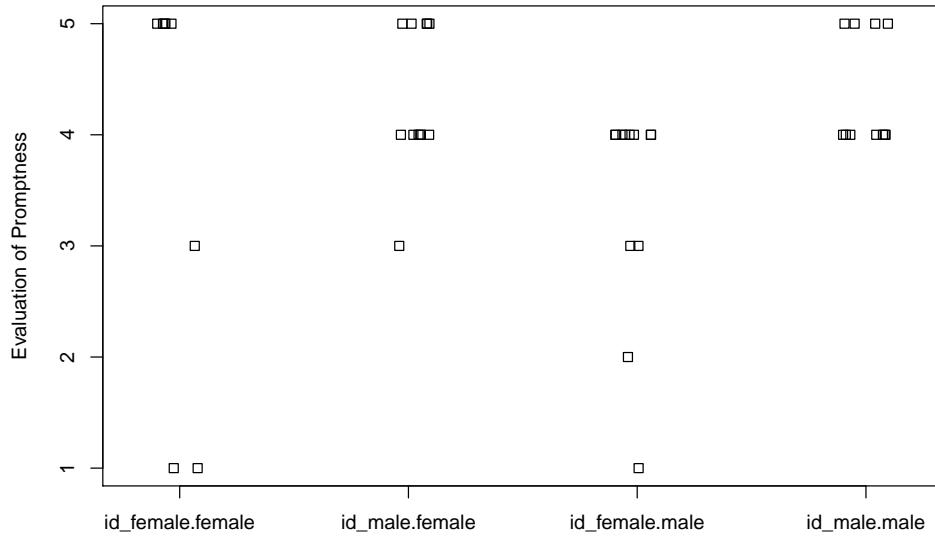


Figure 6.5: Strip chart of Evaluation of TA Promptness by Groups of Students. *xx*

We want to devise a statistic that compares the scores for the perceived males to the perceived females. We can imagine that a student's evaluation of the promptness of the TA in returning assignments may be colored by the student's other interactions with the TA so even though the assignments were graded with the same rubric and returned at the same time, these other interactions with the TA may affect the evaluation of promptness. For this reason, we pool the scores for the students who thought their TA was female and compute the average score, and similarly compute the pooled mean for the students taught by a TA that was identified as female. Our statistic then compares the difference in these pooled means. For our data, we find this difference to be:

```
with(set,
  mean(subset(prompt, taidgender == "id_female"), na.rm = TRUE) -
  mean(subset(prompt, taidgender == "id_male"), na.rm = TRUE))
[1] -0.798
```

The students taught by the female TA are divided roughly equally between the 2 groups—12 perceived the TA to be male and 11 perceived the TA as female—and the students taught by the male TA are so divided. If gender doesn't matter, then these 2 pooled means should be close to 0. How likely is it that random assignment of the students to the TAs can lead to a difference as large as -0.798 or larger?

We want to compare what actually happened to all the possibilities that might have happened under a different assignment of students to sections. In allocating students to sections, we want to keep the number of students in each section constant and we want to keep the students assigned to the female TA in one of her 2 sections and likewise for the students in the male TAs section. That is, we want to examine all possible rearrangements of the 23 students taught by the female TA into a section of 11 (identified as female) and a section of 12 (identified as male) and all possible combinations of the 24 students taught by the male TA with 13 chose for the section perceived to be taught by a female and the remaining 11 in the other section. Each of these random allocations is equally likely so we can potentially compute the exact probability of observing a value of -0.798 under this model. However, there are far too many combinations to enumerate them all so we use simulation to estimate this permutation distribution. From this approximate distribution we can estimate how likely it is to get a score of -0.798 or lower from the proportion of observed scores that fell at or below -0.798.

We identify the steps of this process to be:

- i Randomly assign 11 of the female TA's scores to the identified-female group, and the remainder to the identified-male group. Randomly assign 13 of the male TA's scores to the identified-female group and the remainder to the identified male group.
- ii Compute the difference in average scores for the identified-female and identified-male groups.
- iii Repeat these steps many times to create an approximate sampling/permuation distribution for the statistic, i.e., the difference in average scores for the 2 groups.
- iv Use the approximate distribution to estimate the probability of observing a statistic as low as or lower than the one that we observed in our original data.

How many samples do we need to generate? With 1,000 permutations, we can observe chances that are at least 1 in 1,000, i.e., 0.001. Also, there is sampling error associated with these probabilities because we are not observing all possible permutations. For a probability near 0.05 this error is about 1%. With 100,000 permutations, we can reduce the uncertainty near 0.05 to 0.1%, and an observed chance can be as small as 0.00001. A useful strategy is to start with 1000 permutations and continue to larger numbers only if the observed chance is small enough to be interesting, such as less than 0.1.

We first determine the various counts we need for working with the permutations and sections. That is, we find the number of students taught by each TA and the number of students in each section with

```
taCount = table(set$tagender)
secCount = table(set$tagender, set$taidgender)
```

We take a permutation of the female and male TA's scores with

```
permuteScoresF = sample(set$prompt[set$tagender == "female"])
permuteScoresM = sample(set$prompt[set$tagender == "male"])
```

We can confirm that the `sample()` function simply permutes the scores with

```
identical(sort(permuteScoresF),
          sort(set$prompt[set$tagender == "female"]))

[1] TRUE
```

We easily can assign these permuted scores to sections, because we simply take the first 11 of the scores in `permuteScoresF` for the section with the TA identified as female and the remainder for the other section. We similarly split the scores for the male TA into 2 sections and compute the difference in pooled means with

```
mean(c(permuteScoresF[1:secCount[1]],
      permuteScoresM[1:secCount[2]]), na.rm = TRUE) -
mean(c(permuteScoresF[-(1:secCount[1])],
      permuteScoresM[-(1:secCount[2])]), na.rm = TRUE)

[1] -0.047619048
```

Note that we have included the nonresponders in the randomization, but they are excluded when we compute the average score for the group.

Hopefully, you are already thinking about ways to make this code simpler and more efficient, but before we start streamlining our code, we replicate our randomization and resampling of scores and compute the test statistic (the difference in means) 1,000 times, i.e.,

```
sampleStats = replicate(1000, {
  permuteScoresF = sample(set$prompt[set$tagender == "female"])
  permuteScoresM = sample(set$prompt[set$tagender == "male"])
  mean(c(permuteScoresF[1:secCount[1]],
        permuteScoresM[1:secCount[2]]), na.rm = TRUE) -
  mean(c(permuteScoresF[-(1:secCount[1])],
        permuteScoresM[-(1:secCount[2])]), na.rm = TRUE)
})
```

In Figure 6.6, we see that the sampling distribution of the test statistic is approximately normally distributed. Moreover, the proportion of the simulated test statistics that fall below the observed value of -0.798 is very small, i.e.,

```
sum(sampleStats <= -0.798) / length(sampleStats)

[1] 0.007
```

That is, fewer than 7 of the 1,000 random permutations yielded a test statistic that small. The observed difference is very surprising, and when we rerun the resampling process 100,000 times we get an even smaller value of `0.0046`. We reject the hypothesis of there being no difference in score for the TAs perceived to be male vs. female. Given the original randomization of students to sections, we can conclude that the difference is caused by students rating a TA perceived as male more highly than a TA perceived as female, despite the TA being the same person.

Profiling Code

Now let's turn to the topic of efficiency. Our code took quite a bit of time to run with 100,000 replications. When we timed it, we found the following

user	system	elapsed
18.373	0.568	21.327

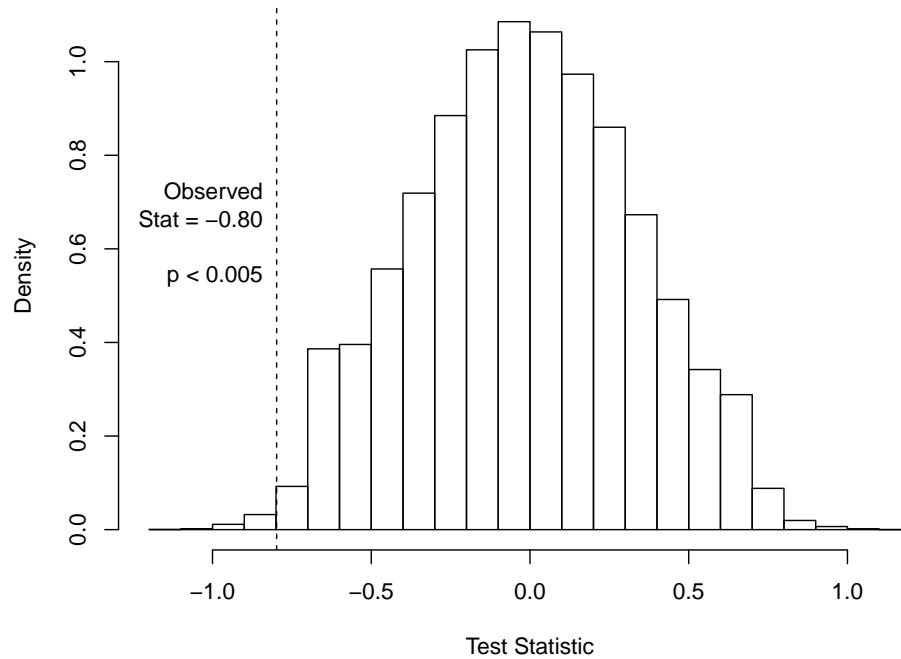


Figure 6.6: Histogram of Test Statistic for Difference in Student Evaluations. *xx*

We have some experience with writing *R* code at this point and have some ideas why the code might be slow. However, we take a more systematic approach and track the time the code spent executing each function call in order to pinpoint precisely the bottlenecks. We use `Rprof()` to do this and `summaryRprof()` to summarize the results from the profiling. We first set the seed for the random number generator so that as we change our code, we can compare run times from the same starting point. Our profiling appears below:

```
set.seed(124681)
Rprof("permute.prof")
sampleStats = replicate(100000, {
  permuteScoresF = sample(set$prompt [set$tagender == "female"])
  permuteScoresM = sample(set$prompt [set$tagender == "male"])
  mean(c(permuteScoresF[1:secCount[1]],
        permuteScoresM[1:secCount[2]]), na.rm = TRUE) -
  mean(c(permuteScoresF[-(1:secCount[1])],
        permuteScoresM[-(1:secCount[2])]), na.rm = TRUE)
})
Rprof(NULL)
head(summaryRprof("permute.prof")$by.self, 10)

      self.time self.pct total.time total.pct
"Ops.factor"      2.40    12.81      5.08    27.11
"mean"            2.00    10.67      3.46    18.46
"sample"          1.74     9.28     14.92    79.62
"$"              1.16     6.19      6.38    34.04
"sample.int"       1.14     6.08      1.14     6.08
```

"[.data.frame"	1.00	5.34	3.72	19.85
"[["	0.92	4.91	4.64	24.76
"match"	0.88	4.70	1.26	6.72
"mean.default"	0.88	4.70	0.98	5.23
"NextMethod"	0.84	4.48	0.84	4.48

We see that `Ops.factor()` takes about 13% of the time. However, our code has no direct calls to this function. We do see that we use factors when creating our permutation so we can guess that this is where the problem lies, but we can also approach this search more systematically.

Tracing a Function Call

The `trace()` function allows us to evaluate an expression at the beginning of the call to a function. We can trace `Ops.factor()` with

```
trace("Ops.factor", quote(print(sys.calls())))
```

```
Tracing function "Ops.factor" in package "base"
[1] "Ops.factor"
```

We simply call `sys.call()` when `Ops.factor()` so that we can see the functions further up in the call stack that have led to `Ops.factor()` being called. Now, when we rerun our code (with only 1 replication to avoid volumes of output) we find

```
sampleStats = replicate(1, {
  permuteScoresF = sample(set$prompt[set$tagender == "female"])
  permuteScoresM = sample(set$prompt[set$tagender == "male"])
  mean(c(permuteScoresF[1:secCount[1]],
        permuteScoresM[1:secCount[2]]), na.rm = TRUE) -
  mean(c(permuteScoresF[-(1:secCount[1])],
        permuteScoresM[-(1:secCount[2])]), na.rm = TRUE)
})

Tracing Ops.factor(set$tagender, "female") on entry
[[1]]
replicate(1, {
  permuteScoresF = sample(set$prompt[set$tagender == "female"])
  permuteScoresM = sample(set$prompt[set$tagender == "male"])
  mean(c(permuteScoresF[1:secCount[1]], permuteScoresM[1:secCount[2]]),
       na.rm = TRUE) - mean(c(permuteScoresF[-(1:secCount[1])],
                               permuteScoresM[-(1:secCount[2])]), na.rm = TRUE)
})

[[2]]
sapply(integer(n), eval.parent(substitute(function(...) expr)),
      simplify = simplify)

[[3]]
lapply(X = X, FUN = FUN, ...)

[[4]]
FUN(X[[i]], ...)
```

```

[[5]]
sample(set$prompt[set$tagender == "female"])

[[6]]
Ops.factor(set$tagender, "female")

[[7]]
.doTrace(print(sys.calls()), "on entry")

[[8]]
eval.parent(exprObj)

[[9]]
eval(expr, p)

[[10]]
eval(expr, envir, enclos)

Tracing Ops.factor(set$tagender, "male") on entry
...

```

untrace("Ops.factor")

Untracing function "Ops.factor" in package "base"

A call to `untrace()` cancels the tracing. We have confirmed that this function call is related to the evaluation of the expression `sample(set$prompt[set$tagender == "female"])` and the similar expression for the male TA. Also, note that the \$-operator takes about 6% of the time. It appears in this same call to `sample()` and nowhere else in the code so eliminating it from the expressions that are being replicated thousands of times can also make our code more efficient.

We can also trace the "`[[.data.frame]`" call and find that it is called 4 times in each replication, twice in each call of `sample()` for `set$tagender` and `set$prompt`. Modifying the `sample()` calls may make our code more efficient.

Move Expressions Out of Loops When Possible

The call to `sample()` to permute the scores for a TA includes taking a subset of all the scores to get those that belong to that particular TA. In the code that we have written, we take this subset every time we permute the scores. This subset doesn't change from one replication to the next so we would be better off taking the subset once, before entering the loop. That is, we create

```
TAscores = list(female = set$prompt[set$tagender == "female"],
                male = set$prompt[set$tagender == "male"])
```

Then our code becomes

```
sampleStats = replicate(100000, {
  permuteScoresF = sample(TAscores[["female"]])
  permuteScoresM = sample(TAscores[["male"]])
  mean(c(permuteScoresF[1:secCount[1]],
        permuteScoresM[1:secCount[2]]), na.rm = TRUE) -
```

```

    mean(c(permuteScoresF[-(1:secCount[1])],
          permuteScoresM[-(1:secCount[2])]), na.rm = TRUE)
  })

```

We time this revision to our code and find:

	user	system	elapsed
	5.34	0.21	6.19

With this simple change, our code is about 3.5 times faster.

We can profile our code again to find out what functions now take up more time, and we find:

	self.time	self.pct	total.time	total.pct
"mean"	1.94	35.93	3.42	63.33
"mean.default"	0.86	15.93	0.96	17.78
"sample.int"	0.84	15.56	0.84	15.56
"sample"	0.72	13.33	1.66	30.74
"["	0.40	7.41	0.40	7.41
"FUN"	0.16	2.96	5.26	97.41
"length"	0.14	2.59	0.14	2.59
"lapply"	0.12	2.22	5.38	99.63
"::"	0.06	1.11	0.06	1.11
"c"	0.06	1.11	0.06	1.11

Now the `mean()` and `sample()` functions take the most time, which seems appropriate. We see that `[` operator takes about 7% of the time. This makes us think about other approaches to creating the permutations of scores. For example, is it any faster to simply permute the indices? We might also consider the data structures that we are using. Are there more efficient structures for the task? We cannot use a matrix because the number of scores in the sections are unequal.

6.4 Cross-Validation and Prediction

Cross-validation is a resampling method that is very useful for model assessment and selection. The basic idea behind cross-validation is to randomly divide the sample into test and training sets, then use the training set to build a model and the test set to assess the model. The assessment stage helps us choose between models; this can involve selecting the order of a polynomial fit in least squares regression or determining the number of nearest neighbors to consult in a nearest neighbor prediction. We assess a model by its prediction error for the data in the test set. Cross-validation reduces problems of overfitting the model to the data and underestimating the error rate because we do not use the same data to both fit and predict.

We assess a model with prediction error; to do this, we fit a model to data and then examine the error when using that model to predict new data. Let's use Q.6-4 (page 246) as an example. An observation consists of 9 features derived from an image and the diagnosis, i.e., the field that we are trying to predict. We can use the k th nearest neighbor method as follows: we use the original observations to partition the feature space into disjoint regions and associated with each region are k observations—these are the k closest observations to

any point in the region. (To determine closeness we use the values the 9 image features.) Then when a new observation arrives, we find the region it belongs to, or, equivalently, the k closest observations in the original data, and we use the diagnosis for these k observations to predict the diagnosis for the new observation. For example, if the majority of the neighbors have a benign diagnosis then we predict that this new observation is also benign. The typical prediction error in this situation is the proportion of malignant cases that are misdiagnosed as benign and the proportion of benign cases that are misdiagnosed as cancerous. The first kind of error is called a Type I error and the other is Type II error.

As a second example, suppose that we have the age and running times in a past 5k race, and we are interested in predicting the run time for registrants for an upcoming 5k race. It appears that the relationship between time and age follows a smooth curve. We can use the method of smoothing splines to fit time to age, but it requires that we specify the value of a tuning parameter. If the tuning parameter is too small, the curve becomes very wiggly and over fits the data. On the other hand, if the parameter is too large, then the curve is too smooth and does not represent the underlying relationship well. We use the training data to fit the model(s) for different values of the tuning parameter, e.g., \hat{f}_λ is the function fitted to the training data using the value λ for the tuning parameter. For a new, test observation that was not used to build the model, we have the runner's age x_i and we predict a typical run time for someone of that age with $\hat{f}_\lambda(x_i)$ and to quantify the prediction error for these test observations we use mean square error,

$$\frac{1}{m} \sum_{i=1}^m (y_i - \hat{f}_\lambda(x_i))^2.$$

We use this test error to choose between competing values for λ .

Model Selection and Prediction

Prediction problems involved the following components:

- Observations with a response, typically represented as y , and xxx variables, typically represented as x .
- A collection of competing models, f_θ , where e.g., θ may be the number of neighbors, the tuning parameter in a smoothing function, or the degree of a polynomial. These are generic models; they do not include data-specific models, such as partitions of the feature space, smooth functions, or coefficients for a polynomial.
- A method of fitting a model to the data, e.g., a metric for determining distance between points in the feature space, or least squares for selecting coefficients for a polynomial.
- A loss function, which often coincides with the technique for fitting the model, that we use to quantify the prediction error. Categorical responses often use 0-1 loss where the loss is 0 if the predicted and true category match and 1 if they don't. Quantitative response often use square error loss or absolute error. We find the average loss when predicting test data from models built with training data.
-

There are several ways to cross-validate in this model selection process. In each case, the data that have been selected for the training set are used to fit or build the model and the data in the test set are used to estimate the prediction error. We describe 2 common types of cross-validation: leave-one-out and v -fold.

Leave-one-out Cross-Validation

The main idea is that we take one observation as the test ‘set’ and the remaining observations as the training set. We then cycle through all of the observations one at a time, each time fitting a model with all but one observation and assessing the prediction error with the one observation left out. We minimize the sum/average of these prediction errors over the competing models to select one. This type of cross-validation is popular when fitting linear models using least-squares because there is a simple calculation to fine the model fitted with one of the observations left out from the model fitted with all of the observations.

v -Fold Cross-Validation

This form of cross-validation partitions the data at random into v subsets of roughly equal size. Each subset is called a fold; it serves as a test set and we use it to compute prediction error for models fitted with the remaining $v - 1$ folds of the data. We cycle through the folds, each time fitting a model with the remaining $v - 1$ and using the fitted model to predict the response in the test fold. This version of cross-validation has been found to provide reasonable results without being computationally intensive. Using 5 or 7 folds tends to work well in many situations.

Hold-out Set

Typically want to report the prediction error for this final model that was chosen. To do this, we typically hold-out some data in advance of model fitting and cross-validation. This hold-out set is not used for any fitting or model selection. Then, after the model has been selected (e.g., the number of neighbors, the degree of the polynomial, the tuning parameter value) with cross-validation, we fit the model using all of our data (except the hold-out set) and use the data that has been held out as a final estimate of the prediction error for the selected model. Typically, we set aside any where from 1/4 to 1/2 of the data for this purpose.

We demonstrate the use of a hold out set and v-fold cross validation in selecting the number of neighbors to use in predicting a cancer diagnosis based on the feature data described in Q.6-4 (page 246). These data are available in the file *breast-cancer-wisconsin.txt*. From the file name, we ascertain that the file is plain text, and after viewing it, we see that it has commas delimiting the values and no column names so we read it with

```
require(readr)
bc = read_delim("breast-cancer-wisconsin.txt", delim = ",",
                 col_names = FALSE)
```

Let’s check a few of the variables’ values to see if we have read the data correctly. We do this with

```
head(bc)
```

	X1	X2	X3	X4	X5	X6	X7	X8	X9	X10	X11
1	1000025	5	1	1	1	2	1	3	1	1	2
2	1002945	5	4	4	5	7	10	3	2	1	2
3	1015425	3	1	1	1	2	2	3	1	1	2
4	1016277	6	8	8	1	3	4	3	7	1	2
5	1017023	4	1	1	3	2	1	3	1	1	2
6	1017122	8	10	10	8	7	10	9	7	1	4

When we use `summary()` to summarize the variables, we find: variables `X2` through `X10`, excluding `X7`, have numeric values that range from 1 to 10; the `X7` variable is a `character` vector, even though our inspection of the first 6 values indicate that it is numeric; and `X1` appears to be an identification variable. We continue our examination of `X7` and call `table()` to examine the various values. We find:

```
table(bc$X7)
```

?	1	10	2	3	4	5	6	7	8	9
16	402	132	30	28	19	30	4	8	21	9

Apparently there are 16 records with ? for a value for this variable, but otherwise it has integer values ranging from 1 to 10. The file *breast-cancer-wisconsin.names.txt* contains some documentation for the data, and after reading it we confirm that the ?s denote missing values. We also determine that X2, ..., X10 measure various features of the image and their values range from 1 to 10, and that X11 is the classification, which is 2 for benign and 4 for malignant. With this information, we can assign useful variable names, treat ? as NA, and create a `factor` classification variable. We can use the `bc` data frame to do this, or read the data in again and specify the variable names and NA values through `read_delim()`. The file is not large so either approach works. We choose the latter, and read and transform the data with

```
varNames = c("id", "thickness", "size", "shape", "adhesion",
            "single.cell", "nuclei", "chromatin", "normal",
            "mitosis", "class")

bc = read_delim("breast-cancer-wisconsin.txt", delim = ",",
                 col_names = varNames, na = "?")

bc$class = factor(bc$class, levels = c(2, 4),
                  labels = c("benign", "malignant"))
```

We can recheck X7, which is now `nuclei` to confirm that the ?s have been properly translated to NA, e.g.,

```
summary(bc$nuclei)
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.	NA's
1.000	1.000	1.000	3.545	6.000	10.000	16

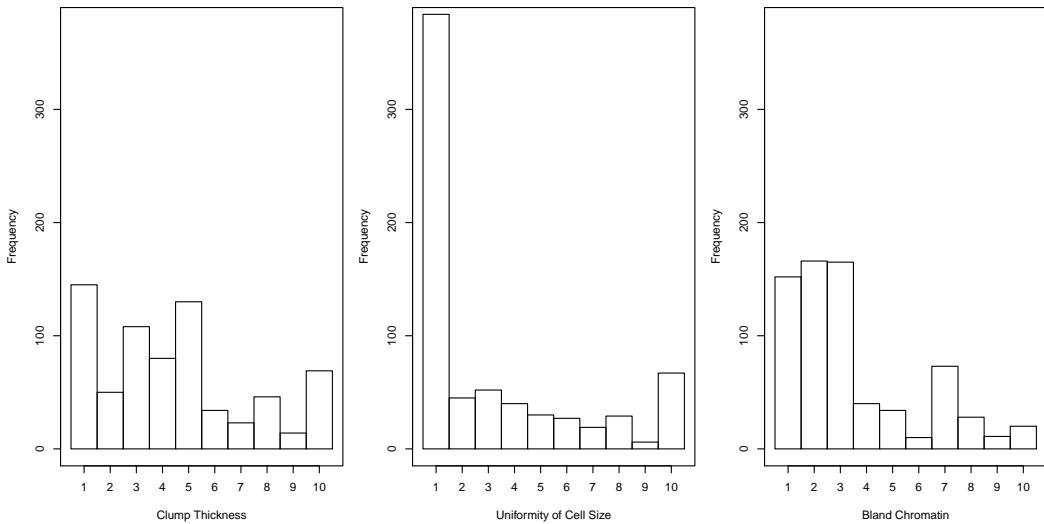
Also, we see that 458 of the images have been hand-classified as benign and the remaining 241 as malignant with

```
summary(bc$class)
```

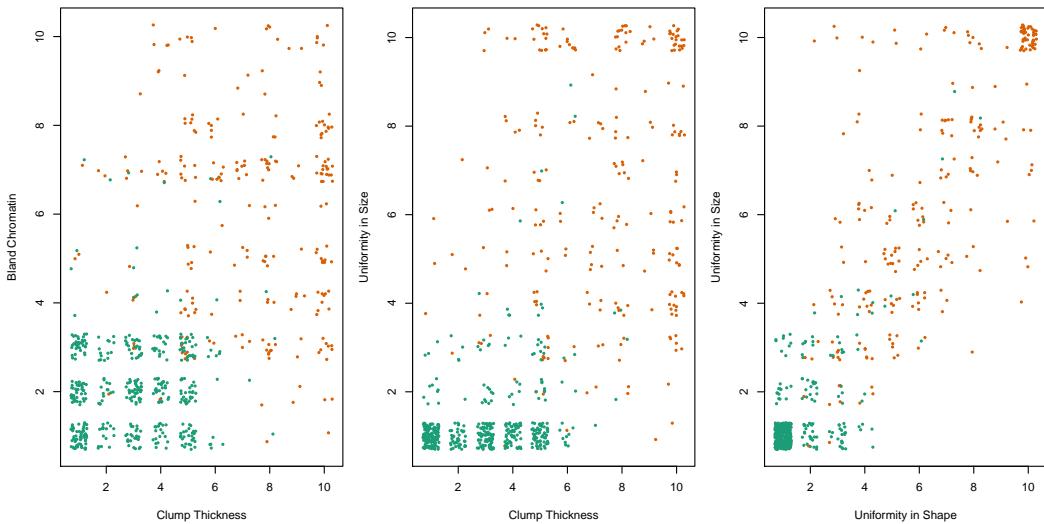
benign	malignant
458	241

We have successfully read the data and have a sense of its dimensions, variable values, etc. Before turning to the prediction problem, let's learn more about the 9 features and their relationship to the classification through exploratory data analysis.

Let's begin by examining the univariate distributions of the features. Given that they take on integer values, we want to control the placement of the histogram bars so that they are centered on the integers. Given that there are only 9 features, we can easily look at all of their distributions. We find that most have skewed right distributions with many small values. [?] shows the histograms for 3 of them. We see that the distribution of uniformity of cell size is highly skewed with more than half taking the value of 1 and the remaining values nearly uniformly distributed from 2 through 10. On the other hand, clump thickness is much more evenly spread across the range from 1 to 10.

Figure 6.7: Histograms of Image Features in Breast Cancer Diagnosis. xx

Let's next examine relationships between these feature variables and the classification. Figure 6.8 shows scatter plots between 3 pairs of variables. We add a small amount of random noise to avoid the over plotting caused by the discreteness in the values. We also use color to denote the classification so that we can determine how successful we might be in using these features for classification. We are looking for separation between the two classes of images, and indeed we see good separation between benign and malignant observations in all 3 scatter plots.

Figure 6.8: Scatter Plots of Image Features in Breast Cancer Diagnosis. xx

6.4.1 Nearest Neighbor Prediction

The k nearest-neighbor method, or k -NN for short, is a relatively simple and intuitive approach to prediction. The nearest-neighbor method (for $k = 1$) works as follows: we have training data for images with features and a known diagnosis; when we get a new observation, i.e., a new set of features for an image with unknown diagnosis, we find the observation in our training data that is closest to this new observation. By close we mean the features are most similar. Then, we simply predict the diagnosis for the new image as the diagnosis of that closest training observation. For k -nearest neighbors where k is larger than 1, we find the k closest training points (in the features domain) and estimate the new observation's diagnosis from an aggregate of the diagnoses of the k training points.

We naturally think of measuring the distance between two sets of features with Euclidean distance, i.e.,

$$\sqrt{(f_1^* - f_1)^2 + \cdots + (f_9^* - f_9)^2},$$

where f_i denotes the value for the i -th feature for a training observation and f_i^* is the value for the i -th feature for the new image whose diagnosis we are trying to predict. Typically, we standardize each variable used in computing the distance between observations; however, since the features are already range from 1 to 10, we find that standardizing does not improve the performance of our near-neighbor predictor.

Before embarking on the task of selecting k using cross-validation and holding out a subset of the data for estimating the prediction error, let's consider the simpler problem of predicting the diagnosis for each of the observations in our original data set based on its 5 nearest neighbors in the data. What are the tasks we need to carry out to do this? We need to

- Find the distance between every pair of observations in the data.
- Find the 5 nearest neighbors to each observation.
- Use the 5 neighbors' diagnoses to vote on an observation's diagnosis
- Assess the prediction errors—the proportion of misclassified benign cases and the proportion of misclassified malignant cases.

Selecting a Data Structure

Since we are computing distances between vectors of 9 feature values, it may be helpful to organize the data in a structure other than the data frame. Specifically, a matrix offers a more natural, and more efficient, structure for computing distances. We can organize the data so that we have 9 columns of features. We are not computing distances with the record identifier, `id`, or the diagnosis, `class`, so we do not include these in our feature matrix but we do set up a vector to hold the classifications. Also, looking ahead, it will be easier to work with the classification as a 0-1 values for counting and voting. We create these new structures with

```
bcFeatures = as.matrix(bc[, -c(1, 11)])
bcTruth = (bc[, 11] == "malignant") + 0L
```

Now we can easily find the distances between observations.

Creating Example Data for Checking Code

We want to check our code carefully to make sure that it works the way that we expect. To help us in this process, we can take a small subset of the breast cancer results that is small enough for us to examine all the records and check the computations. We select a few

records from each classification to form a miniature training set and even fewer to form a test set. We want to avoid having the dimensions of our test and train match the number of neighbors (5) or the number of features (9) and we want them to have different dimensions from each other to make it easier for us to check our code. For example, we can more easily check the dimensions of an intermediate variable if these values are all different. We choose the following records for our sample:

```
sampleTrainIndx = c(1:6, 13:22, 36:39)
sampleTestIndx = 23:25
```

We use these indices to create our sample feature matrix and class vectors:

```
sampleTrainFeatures = bcFeatures[sampleTrainIndx, ]
sampleTestFeatures = bcFeatures[sampleTestIndx, ]
sampleTrainTruth = bcTruth[sampleTrainIndx]
sampleTestTruth = bcTruth[sampleTestIndx]
```

Our intermediate goal is to develop code to find the 5th nearest neighbor prediction for an observation in the sample test set using the sample training set.

We need to find the distances between an observation in the test set and each of the observations in the training set. The `dist()` function computes distances between all pairs of observations. This is not quite what we want so we take a more direct approach, e.g.,

```
apply((sampleTrainFeatures - sampleTestFeatures[1, ])^2, 1, sum)
[1] 11 148 16 172 18 417 37 13 283 83 11
[12] 20 240 26 229 259 7 420 48 205
```

Our expression returned a numeric vector with 20 elements so it is the correct class and size. We wrote our code thinking that the set of feature values for the test observation would be recycled and subtracted from each of the rows in the matrix of training features, but is that what's happening? Let's dig a bit deeper and figure out what the sum of squared differences is between the first training observation and the first test observation. These records are

```
sampleTrainFeatures[1, ]
```

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

```
sampleTestFeatures[1, ]
```

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

We can simply eyeball these two vectors and see that something is amiss. Most of the features have the same value and the only differences are 3 and 5 in the fist component and 2 and 3 in the 7th, which yields a sum of squared difference of 5, not 11! Clearly, the recycling is not working as expected. We can modify our code and continue our checking to find the following gives the expected results:

```
m = nrow(sampleTrainFeatures)
l = ncol(sampleTrainFeatures)
distances =
  apply((sampleTrainFeatures -
    matrix(sampleTestFeatures[1, ], nrow = m, ncol = l,
    byrow = TRUE))^2, 1, sum)
```

These distances:

```
[1] 5 146 2 154 6 427 33 9 281 83 1
[12] 2 236 10 219 243 1 376 36 201
```

match what we expect from our ‘hand’ calculations.

Now that we have the set of distances, we want the 5 closest. We use `order()` to find them with

```
order(distances) [1:5]
```

```
[1] 11 17 3 12 1
```

In other words, the 5 smallest values are in the 11th, 17th, 3rd, 12th, and 1st positions of `distances`. Simple inspection of `distances` confirms this. These are the observations that are used to predict the classification for our test record. If we use majority rule, we find

```
sampleTrainTruth[order(distances) [1:5]]
```

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

All 5 nearest neighbors are benign so our prediction is benign. This prediction is correct, because `sampleTestTruth[1]` is 0. Rather than inspect all 5 classifications, we can simply construct a proportion with `mean()`, e.g.,

```
mean(sampleTrainTruth[order(distances) [1:5]])
```

The possible values are 0, 0.2, 0.4, ..., 1. According to majority rule, we predict malignant if the proportion of malignant neighbors is at least 1/2. However, we can, e.g., use a stricter cut point and predict malignant classification only if the proportion of such neighbors exceeds 3/4. Moreover, we can use a weighted average with the votes weighted inversely proportional to their distance from the test observation.

Now that we have completed a simple example, let’s generalize our nearest neighbor prediction method and create a function for this purpose. We call our function `nnPred()`. What are its inputs? In our example, we used a test and training set and classifications for the training set. We also used the test set classification to check how well our prediction worked. Do we need that as an input? The estimation of prediction error is a different task and we leave it to the work of another function. Are there any other arguments our function needs? We need to know the number of neighbors. If we wanted to generalize our code, we can also make the metric used in computing distances, the weights applied to the votes of the neighbors, and a threshold for determining the classification. Rather than provide a threshold as a parameter, let’s return the vote summary. For now, we will also leave out the metric and voting weights from our function signature, which appears as

```
nnPred = function(trainSet, testSet, trainClass, k)
```

Simplifying and Generalizing Code with Helper Functions

This exercise of determining the inputs suggests that we write helper functions to separate tasks and allow for flexibility in updating our code. Each of these functions performs one distinct subtask within the overall task of nearest neighbor prediction. By separating out these tasks as functions and we can build and test code incrementally. The natural helper functions we have identified are to calculate distances, find the nearest neighbors, and use the neighbors’ classifications to make predictions. Based on our earlier coding example, we can identify the inputs and output of these helper functions as follows:

- **calcDist()** – Compute the distance between train and test observations.
Parameters: *trainSet*, an m by l training matrix; and *testSet*, an n by l test matrix.
Return value: m by n distance matrix.
- **findNN()** – Find the nearest neighbors in the training set for each observation in the test set.
Parameters: *distances*, an m by n distance matrix; and *k*, the number of neighbors to search for.
Return value: An m by k integer matrix with the indices of the k smallest values in each row of the distance matrix provided.
- **tallyVotes()** – Tally the classifications of the nearest neighbors.
Parameters: *neighbors*, an m by k matrix of indices; *classifications*, a n vector of classifications.

Although we have not written the helper functions yet, with specifications of the inputs and output we can write our top level function as

```
nnPred =
function(trainSet, testSet, trainClass, k)
{
  distances = calcDist(testSet, trainSet)
  closestK = findNN(distances, k)
  voteNN(closestK, trainClass, k)
}
```

This function clearly demonstrates the tasks involved in developing the nearest neighbor predictions. Now our next steps are to complete the helper functions, **calcDist()**, **findNN()**, and **voteNN()**.

We can write these functions starting from the code in our example. For **calcDist()**, we have

```
calcDist = function(testSet, trainSet)
{
  m = nrow(trainSet)
  l = ncol(trainSet)
  if (!is.matrix(testSet)) testSet = matrix(testSet, nrow = 1)
  apply(testSet, 1,
    function(row) {
      apply((trainSet - matrix(row, nrow = m, ncol = l,
                                byrow = TRUE))^2, 1, sum)
    })
}
```

Notice how we have generalized our earlier code to handle a matrix of test observations. We will want to test **calcDist()** with both vectors and matrices for the test set.

We can use the sample training and test set we have extracted from the original matrix to test our code. We begin testing by providing the matrix of test observations with

```
calcDist(sampleTestFeatures, sampleTrainFeatures)
```

```
[,1] [,2] [,3]
[1,]    5   NA   16
[2,] 146   NA  157
[3,]    2   NA    5
[4,] 154   NA  169
[5,]    6   NA   13
[6,] 427   NA  438
...

```

We have encountered a couple of surprises in the output: the shape of the matrix is 20 by 3, not 3 by 20, and the distances for the 2nd test case are all NA. Happily, we do find that the first column matches the distances that we computed earlier for the 1st observation in the test set. On closer inspection of the test data we see that the 2nd observation includes an NA for one of its features, i.e.,

```
[1] 8 4 5 1 2 NA 7 3 1
```

This NA leads to the NAs for all of the distances in the return value. We can easily update our function to account for these issues with

```
calcDist = function(testSet, trainSet)
{
  m = nrow(trainSet)
  l = ncol(trainSet)
  if (!is.matrix(testSet)) testSet = matrix(testSet, nrow = 1)
  distances = apply(testSet, 1,
    function(row) {
      apply((trainSet -
        matrix(row, nrow = m, ncol = l, byrow = TRUE))^2,
        1, sum, na.rm = TRUE)
    })
  return(t(distances))
}
```

When we apply this revised `calcDist()` to our test and train samples we no longer have NAs and the shape is as expected. Let's run one additional test and compute the distances between each pair of elements in the test sample and compare the results with the return value from `dist()`, i.e.,

```
calcDist(sampleTestFeatures, sampleTestFeatures)
```

```
[,1] [,2] [,3]
[1,]    0    79     5
[2,]   79     0    94
[3,]    5    94     0
```

```
dist(sampleTestFeatures)
```

	1	2
2	9.427	
3	2.236	10.283

The `dist()` function returns only the lower triangle of the distance matrix to save space since it is symmetric. However, our calculations do not match at all. It occurs to us now that we forgot to take the square root of the sums of squares! Fortunately, this error does not impact the ordering of the distances. We confirm that `sqrt(5)` is about 2.236, but `sqrt(79)` is 8.89, not 9.427 and the distance between the 2nd and 3rd observations is also incorrect. What is happening? Does it have something to do with the presence of NAs in the 2nd observation? When we reread the documentation for `dist()`, we find the statement, “If some columns are excluded in calculating a Euclidean, Manhattan, Canberra or Minkowski distance, the sum is scaled up proportionally to the number of columns used.” Let’s seek further clarification in a Web search. When we search on the terms `r dist na`, we find a helpful discussion on Stack Overflow (<http://stackoverflow.com/>) about how the sum of squares are scaled to account for the smaller number of terms. The scaling factor is $l / (l - n)$, where l is the number of elements and n is the number of missing values. Let’s adopt this approach to computing distances and revise `calcDist()` as follows:

```
calcDist = function(testSet, trainSet)
{
  m = nrow(trainSet)
  l = ncol(trainSet)
  if (!is.matrix(testSet)) testSet = matrix(testSet, nrow = 1)
  distances =
    apply(testSet, 1,
      function(row) {
        sqs = (trainSet -
          matrix(row, nrow = m, ncol = l, byrow = TRUE))^2
        naCt = apply(is.na(sqs) + 0L, 1, sum)
        sumSqs = apply(sqs, 1, sum, na.rm = TRUE)
        sumSqs * l / (l - naCt)
      })
  return(t(sqrt(distances)))
}
```

When we test our revised function, we find that the results now match those from `dist()`.

Checking correctness using Different Computational Approaches

We need to be careful when we want to check that our code is correct if we are checking the result against another that is calculated using an alternative approach. For example, we might expect the following comparison to return TRUE,

```
identical(calcDist(sampleTestFeatures, sampleTestFeatures),
           as.matrix(dist(sampleTestFeatures)))
[1] FALSE
```

If we use different computations, even possibly a different order of computations, then the results may not be equal because of rounding error. We can print the difference between the elements of these 2 matrices to see if that is the problem, e.g.,

```
print(abs(calcDist(sampleTestFeatures, sampleTestFeatures) -
           as.matrix(dist(sampleTestFeatures))), digits = 22)
  1 2 3
1 0 0 0
2 0 0 0
3 0 0 0
```

We find that the elements agree up to 22 digits, which is the maximum allowed in `print()`. And, when we use `==` to compare the elements, we find that they are all TRUE, i.e.,

```
all(calcDist(sampleTestFeatures, sampleTestFeatures) ==
  as.matrix(dist(sampleTestFeatures)))
[1] TRUE
```

Even if the elements are not identical, we might consider our calculations correct if the differences were small, e.g., less than 0.0000001, which we can check with an expression like

```
all(abs(calcDist(sampleTestFeatures, sampleTestFeatures) -
  as.matrix(dist(sampleTestFeatures))) < 0.0000001)
```

This approach is more reasonable when comparing numeric values. So how are these 2 results different? We can check that their classes match, and we can check that their row and column names match. We do find a difference here; the return value from `dist()` has row and column names but `calcDist()` return value does not. That is,

```
rownames(calcDist(sampleTestFeatures, sampleTestFeatures))
NULL

rownames(as.matrix(dist(sampleTestFeatures)))
[1] "1" "2" "3"
```

We didn't notice this distinction earlier because when we printed the return value to the console, these results appeared the same because the `print()` function adds row and column names to make it easier to read off values.

The next helper function we want is `findNN()`, which provides the indices of the nearest neighbors. This is simply,

```
findNN = function(distances, k)
{
  apply(distances, 1, order)[1:k, ]
```

We test our code with

```
sampleDists = calcDist(sampleTestFeatures, sampleTrainFeatures)
sample5NN = findNN(sampleDists, 5)
sample5NN

 [,1] [,2] [,3]
 [1,] 11    7   17
 [2,] 17    19   8
 [3,] 3     10   3
 [4,] 12    14   12
 [5,] 1     1   11
```

As with `calcDist()` the shape of the return value is not as expected. Otherwise, the results are correct. We modify our function by returning the transpose of this matrix. Now, when we evaluate `sample5NN = findNN(sampleDists, 5)`, `sample5NN` is a 3 by 5 matrix.

The next helper function tallies the classifications for the nearest neighbor. As mentioned earlier, for now, we simply average the classifications.

```
voteNN = function(closeNeighbors, trainDiagnosis, k)
{
  votes = matrix(trainDiagnosis[ closeNeighbors ],
                 byrow = FALSE, ncol = k)
  apply(votes, 1, mean)
}
```

We can again test this code with our sample test and train observations, e.g., `voteNN(sample5NN, sampleTrainTruth, 5)` returns the results expected, `0.0 0.4 0.0`.

Adding Checks and Warnings

When we tested our code, we accidentally used a value of `k` that was too large and got unexpected results. This isn't necessarily a problem if the function is only ever being called by one of our other functions and so the inputs will always be correct. However, if the function might be used more generally, then we probably want to add checks on the inputs and issue meaningful warnings and error messages if there's a problem. With `voteNN()` we expect the number of columns in `closeNeighbors` to match `k`. If we are not provided with enough neighbors then we will want to throw an exception. On the other hand if we are given too many neighbors, we can simply use those that we need. In this case, we should give a warning to let the caller know of the potential problem. Additionally, we need `trainDiagnosis` to have at least `k` columns, for otherwise, we cannot find `k` nearest neighbors and we stop execution of the code and provide an explanation for the problem. In fact, we can provide a default value for `k` that depends on the first 2 arguments in the function so the caller need not provide it and yet it is a reasonable value given the other inputs, e.g., the smaller of the number of elements in `trainDiagnosis` and the number of columns in `closeNeighbors`. Our revision is show below. We use `warning()` to issue a warning message and continue execution of the code and `stop()` to issue an error message and halt execution.

```
voteNN =
function(closeNeighbors, trainDiagnosis,
          k = min(ncol(closeNeighbors), length(trainDiagnosis)))
{
  if (length(trainDiagnosis) < k | ncol(closeNeighbors) < k)
    stop(paste("need at least k elements in trainDiagnosis and
              k columns in closeNeighbors"))
  if (ncol(closeNeighbors) > k) {
    closeNeighbors = closeNeighbors[, 1:k]
    warning("More than k neighbors provided in closeNeighbors,
            using only the first k")
  }

  votes = matrix(trainDiagnosis[ closeNeighbors ],
                 byrow = FALSE, ncol = k)
  apply(votes, 1, mean)
}
```

Now when we call `voteNN()` and supply a value for `k` that is larger than the number of elements in the classification vector or the number of columns in the matrix of nearest neighbors, the function issues an error message, such as

```
Error in voteNN(sample5NN, sampleTrainTruth, k = 25) :
  need at least k elements in trainDiagnosis and
  k columns in closeNeighbors
```

When we do not provide a value for `k`, the function call `voteNN(sample5NN, sampleTrainTruth)` returns `0.0 0.4 0.0`, and when the value supplied for `k` is smaller than the number of neighbors supplied in `closeNeighbors`, e.g., with `voteNN(sample5NN, sampleTrainTruth, k = 4)`, we return

```
[1] 0.0 0.5 0.0
```

and issue the warning,

```
Warning message:  
In voteNN(sample5NN, sampleTrainTruth, k = 4,  
           provideAll = TRUE) :  
  More than k neighbors provided in closeNeighbors,  
  using only the first k
```

Of course, there are many other checks that we can add to our code, e.g., we can check that `closeNeighbors` and `k` are both integer and that `trainDiagnosis` is numeric. There's a balance between allowing the regular error handling to take over and providing special error messages and handling in our functions. We have let *R* issue errors for these basic problems with the type of input and included only checks for dimension matching.

Tracing the Source of an Error

The helper functions are complete and tested so we are ready to try our nearest neighbor method. We start with our sample data,

```
nnPred(sampleTestFeatures, sampleTrainFeatures,  
       sampleTrainTruth, 5)
```

```
Error in apply(distances, 1, order)[1:k, ] : subscript out of bounds
```

This is unexpected given that we have tested all of the pieces pretty thoroughly. We can use `traceback()` to trace the sequence of call that led to the error with

```
traceback()  
3: t(apply(distances, 1, order)[1:k, ]) at #3  
2: findNN(distances, k) at #5  
1: nnPred(sampleTestFeatures, sampleTrainFeatures, sampleTrainTruth, 5)
```

The error is within the `findNN()` function, which was the simplest of the 3 helper functions that we wrote. If we are still unclear about the problem, we can put a call to `browser()` into `findNN()` prior to where the error occurs, or we can set our default options to enter browser mode when an error occurs, we do this with

```
options(error = recover)
```

Now, when we call our `nnPred()` function with

```
nnPred(sampleTestFeatures, sampleTrainFeatures,  
       sampleTrainTruth, 5)
```

the `recover()` function is called when the error occurs. The `recover()` prints the list of current calls, as with `traceback()`. It then prompts the user to select a call frame to enter. Then the *R* browser is invoked from the corresponding environment. We select the 2nd call frame, i.e., the environment within `findNN()`,

```
Error in apply(distances, 1, order)[1:k, ] : subscript out of bounds  
Enter a frame number, or 0 to exit  
1: nnPred(sampleTestFeatures, sampleTrainFeatures, sampleTrainTruth, 5)  
2: #5: findNN(distances, k)  
3: #3: t(apply(distances, 1, order)[1:k, ])  
  
Selection: 2
```

Called from: nnPred(sampleTestFeatures, sampleTrainFeatures, sampleTrainTruth, 5)

Now we can type *R* expressions to be evaluated in that environment. We examine the contents of the variables in the environment with

```
Browse[1]> ls()  
[1] "distances" "k"
```

```
Browse[1]> k  
[1] 5
```

```
Browse[1]> dim(distances)  
[1] 20 3
```

We can exit the browser with typing *c* at the browser prompt, and then we are returned to the `recover()` prompt. We exit `recover()` by selecting 0 for the call frame.

We earlier had this problem with the dimensions of the distance matrix, and we fixed it. This implies the input to the `calcDist()` function that creates the distance matrix must be incorrect. However, when we check this code in `nnPred()`, we find

```
distances = calcDist(testSet, trainSet)
```

The error is simple: we invoked `nnPred()` incorrectly. That is, we called the function with `nnPred(sampleTestFeatures, sampleTrainFeatures, sampleTrainTruth, 5)`

However, the `nnPred()` function signature is

```
nnPred = function(trainSet, testSet, trainClass, k)
```

We provided `trainSet` with the test data `sampleTestFeatures` and vice versa! Why did we make such a simple mistake? In part because we reversed the order of the test and train arguments in our helper functions, putting the test set first and the train set 2nd. Since the design of the functions is up to us, it makes sense to maintain consistency across the function signatures. We choose to put the test set first as we think about the problem as looking for the distances between the test set observations and the train set. We have a quick fix with

```
nnPred =  
function(testSet, trainSet, trainClass, k)  
{  
  distances = calcDist(testSet, trainSet)  
  closestK = findNN(distances, k)  
  voteNN(closestK, trainClass, k)  
}
```

6.4.2 Hold Out Test Set

We now have a mechanism for predicting a diagnosis for breast cancer using nearest neighbors. To measure the predictive ability of the nearest neighbor method, we need a test set for the breast cancer data. That is, rather than use all of the observations in `bcFeatures`, we hold out a portion of these observations for assessing the predictive capability of the nearest neighbor method. We want the data that are held out to be similar to the data used as the training set. Random sampling works well for this purpose. By randomly choosing the observations for the test set, we are highly likely to obtain a sample that looks like the population (in this case the population is the 699 images). Of course, the sample will not look exactly like the population, but relatively large samples resemble the population quite closely. As mentioned earlier, a typical hold-out set consists of 1/4 to 1/3 of the observations.

We select the indices of our hold-out and training sets with

```
nObs = nrow(bcFeatures)
set.seed(24613)
holdOutIndex = sample(nObs, 225, replace = FALSE)
trainIndex = (1:nObs) [ - holdOutIndex]
```

Note that we set the seed for the random number generator so that if needed, we (and others) can recreate this data split. We also sample indices rather than the observations of `bcFeatures` themselves because we also need to divide `bcTruth` into 2 parts. We split our feature matrix and classification vector using these indices, i.e.,

```
holdOutSet = bcVars[holdOutIndex, ]
trainSet = bcVars[trainIndex, ]
trainDiagnosis = bcTruth[trainIndex]
holdOutDiagnosis = bcTruth[holdOutIndex]
```

We can compare the presence malignant observations between the 2 groups with

```
mean(trainDiagnosis) - mean(holdOutDiagnosis)
[1] 0.004
```

The hold out and training sets have very similar proportions of benign and malignant observations. We can also compare the correlations between features that are used for prediction between the 2 sets of data with

```
cor(trainSet, use = "complete.obs") - cor(holdOutSet, use = "complete.obs")
            thickness    size    shape adhesion single.cell nuclei ...
thickness        0.00 -0.01   0.02     -0.02      0.06 -0.03
size           -0.01  0.00 -0.01      0.01      0.07 -0.11
shape          0.02 -0.01   0.00     -0.01      0.04 -0.11
adhesion       -0.02  0.01 -0.01      0.00      0.09 -0.06
single.cell    0.06  0.07   0.04      0.09      0.00 -0.06
nuclei         -0.03 -0.11 -0.11     -0.06     -0.06  0.00
chromatin      0.01 -0.01 -0.01     -0.04      0.08 -0.01
normal          0.13  0.03 -0.01     -0.10      0.08 -0.08
mitosis         0.08  0.05   0.02      0.04      0.16 -0.08
```

Many of the correlations are very close between the 2 sets of data. However, there are some differences. For example, we find that `nuclei` is more highly correlated with `size` and `shape` in the hold out set.

Let's use the training data to make predictions for the data that we have set aside. What value of `k` should we use? Let's try 25. We make the predictions with

```
holdOutPred = nnPred(holdOutSet, trainSet, trainDiagnosis, k = 25)
```

To determine how successful our prediction method is, we want to determine the sensitivity and specificity. That is how well are we able to find the malignant cases (sensitivity) and to find the benign cases (specificity). Let's prepare a function to perform these calculations as we suspect that we will want to use it frequently. We define `calcErrors()` as

```
calcErrors = function(voteClass, trueClass, threshold = 0.5)
{
  numTest = length(trueClass)
  num1 = sum(trueClass)
  num2 = numTest - num1
  predClass = (voteClass >= threshold) + 0L
  sensitivity = sum(predClass[ trueClass != 0 ]) / num1
  specificity = sum((1 - predClass[ trueClass == 0])) / num2
  total = (num1 * sensitivity + num2 * specificity) / numTest
  return(c(sensitivity, specificity, total))
}
```

For the data that we set aside, we have the following accuracy:

```
calcErrors(holdOutPred, holdOutDiagnosis)

[1] 0.922 0.980 0.960
```

The 25 NN method works well; it properly diagnosed 92% of the malignant cases and 98% of the benign cases. The diagnosis was made based on whether the vote tally exceeds 0.5, i.e., when at least half of the neighbors have benign classifications, then the prediction is benign. We might wonder whether we can do better with another threshold. When we try 0.7 and 0.1 we find

```
calcErrors(holdOutPred, holdOutDiagnosis, 0.7)

[1] 0.896 0.986 0.956
```

```
calcErrors(holdOutPred, holdOutDiagnosis, 0.1)

[1] 0.961 0.966 0.964
```

Notice that when we set the threshold very low, i.e., when only 10% of the neighbors are malignant, then we detect 96% of the malignant cases and the detection for benign cases doesn't drop much. The total number of cases correctly detected remains the same, 96%, but we have improved our ability to properly diagnose malignant cases at the expense of making 1.4% more mistakes for the benign case. This seems like a reasonable trade off.

The actual choice of the threshold is a model selection problem, as is the choice of `k`. We might wonder, how well 5 NN or 15 NN performs. We can rerun `nnPred()` for values of `k` from 1 to 50, and compare the results. This raised a few questions, one statistical and one

computational. Computationally, we ask ourselves whether we have designed our function, `nnPred()`, efficiently? And our statistical question is what are the drawbacks to using the hold out set to both assess the predictive ability of the model and select the best model? We address the statistical question in the next section on cross-validation, and we address the computational question here.

Efficiency and Vectorization

If we plan to invoke our function many times or to use it with large amounts of data, we need to consider the question of efficiency. One design point to consider is whether our functions can take vectors/lists as inputs. In the case of `nnPred()`, we can ask whether we can specify `k` as a vector and get the predictions for the various values in `k`. Our `nnPred()` calls `calcDist()`, `findNN()` and `voteNN()`. The first doesn't involve `k` because we simply need to find distances between all pairs of observations with one observation from the test set and the other from the training set. The `findNN()` function returns the indices for the neighbors from 1 to `k`. This function doesn't need modifying either because for a value of `k`, such as 25, we receive the 25 nearest neighbors so we also have available to us the `k` nearest neighbors for any value less than 25 at no additional work. The `voteNN()` function can be easily updated to use these nearest neighbors to make predictions based on any `k` less than 25. One simple approach to modifying `voteNN()` is to return either the prediction for `k` NN, or for every value between 1 and `k`. We can specify this with an additional argument, `provideAll`, which we set to the default value of `FALSE`. This requires us to add `provideAll` to `nnPred()` too.

Our revised `predNN()` is simply,

```
nnPred =
function(testSet, trainSet, trainClass, k, allNN = FALSE)
{
  distances = calcDist(testSet, trainSet)
  closestK = findNN(distances, k)
  voteNN(closestK, trainClass, k, allNN)
}
```

The revision to `voteNN()` is only slightly more complex. We use `cumsum()` to efficiently compute the kNN for intermediate values with

```
voteNN =
function(closeNeighbors, trainDiagnosis,
         k = min(ncol(closeNeighbors), length(trainDiagnosis)),
         provideAll = FALSE)
{
  if (length(trainDiagnosis) < k | ncol(closeNeighbors) < k)
    stop(paste("need at least k elements in trainDiagnosis and \n
               k columns in closeNeighbors"))
  if (ncol(closeNeighbors) > k) {
    closeNeighbors = closeNeighbors[, 1:k]
    warning("More than k neighbors provided in closeNeighbors,\n
            using only the first k")
  }

  votes = matrix(trainDiagnosis[ closeNeighbors ],
                 byrow = FALSE, ncol = k)
  if (provideAll) {
```

```

    t (apply(votes, 1, function(row) cumsum(row) / (1:k)))
} else apply(votes, 1, mean)
}

```

Now, we can get nearest neighbor estimates for each k from 1 to the value specified in the function's input, e.g.,

```

nnPred(sampleTestFeatures, sampleTrainFeatures,
       sampleTrainTruth, 4, TRUE)

[,1] [,2] [,3] [,4]
[1,] 0 0.0 0.000 0.0
[2,] 1 0.5 0.667 0.5
[3,] 0 0.0 0.000 0.0

```

Let's modify `calcErrors()` so that it accepts a matrix of predictions.

Anticipating Errors

As we wrote our code, we made several mistakes with matching dimensions between the arguments to the function so we want to add some checks to our function to watch for certain problems. Before, writing code to do this, let's consider the kind of mistakes made and confusions we had in developing our code.

- `trueClass` is a vector of classifications that we want to check our vote tallies against. When we working with one k , then `voteClass` can be a vector the same length as `trueClass`. If not, we consider this an error.
- If we supply a matrix for `voteClass` then we expect the rows to correspond to observations and the columns to predictors so the number of rows in `voteClass` should match the length of `trueClass`. If not, we consider this an error.
- We earlier used a factor for the classification vector. This function expects `trueClass` to be a 0-1 vector. If it's logical, we can design our code to implicitly coerce the logical to 0-1 values. For other classes of vectors,

For the first two potential problems, we can't proceed with our calculations of prediction error so we need to throw an exception. We use `stop()` with a special error message to respond to the first 2 errors

```

numTest = length(trueClass)
if (!is.matrix(voteClass)) {
  if (length(voteClass) == numTest) {
    voteClass = matrix(voteClass, nrow = numTest)
  } else stop("lengths of voteClass and trueClass must match")
} else {
  if (nrow(voteClass) != numTest) stop("Number of rows in
                                         voteClass must match length of trueClass")
}

```

Notice that in addition to checking for the problems with the inputs, we also reformat `voteClass` to a 1-column matrix if it is a numeric vector. This way our code works the same way for vector or matrix inputs. Our code anticipates problems with the inputs, some of these lead to errors, and others incorrectly compute the prediction error without throwing an exception. We have used the `stop()` function to provide function-specific error messages for these errors.

Catching Errors

For the third issue above, rather than throw an exception if passed a factor or character vector for `trueClass`, we can convert `trueClass` to a vector of 0s and 1s. In this case, we want to give an error message that indicates the changes we have made to `truClass`. Rather than anticipate every kind of error that the caller can make, e.g., providing a factor, character, we can catch the error as it occurs and take over the error handling. Rather than aborting the execution, we can instead create a 0-1 vector from `trueClass`. We use `tryCatch()` to do this with

```
tryCatch({n1 = sum(trueClass)},
        silent = TRUE,
        error = function(e) {
          x = trueClass[1]
          warning(paste("expect trueClass to be 0-1.
                         Set values of", x, "in trueClass to 1"))
          trueClass <- (trueClass == x) + 0L
          n1 <- sum(trueClass)
        })
```

We have provided 3 inputs to `tryCatch()`. The first is the expression(s) for *R* to try to evaluate. If these evaluate successfully then execution continues with the expression following the call to `tryCatch()` and `n1` is created. The 2nd argument indicates that we do not want *R* to provide any error or warning messages. The value of the 3rd argument is a function that is called when an error is encountered in evaluating the expressions in the first argument. Our function issues a special warning message that depends on the contents of `trueClass`. Since we want to continue by converting `trueClass` into a 0-1 vector, we create such a vector by setting all elements to 1 that match the first element of `trueClass` and the others are set to 0.

Assignments into Other Environments

Notice that we use the double left arrow, `«-`, to make this assignment. We also use `«-` to assign a value to `n1`. Since the first attempt to assign `n1` a value triggered an error, `n1` has not been placed in the function's call frame. The `«-` assigns these values to these variables in the parent environment of the function. The parent environment of this function is `calcError()` because our error handling function is defined in the call frame of `calcError()`. There are other ways to specify the environment in which to assign variables. These are covered in greater detail in XXXX.

The version of `calcError()` that incorporates this error handling appears as

```
calcErrors = function(voteClass, trueClass, threshold = 0.5)
{
  ## voteClass is a matrix rows = #obs and cols = k
  ## trueClass is a 0-1 vector or logical with length = #obs
  numTest = length(trueClass)
  if (!is.matrix(voteClass)) {
    if (length(voteClass) == numTest) {
      voteClass = matrix(voteClass, nrow = numTest)
    } else stop("lengths of voteClass and trueClass must match")
  } else {
    if (nrow(voteClass) != numTest) stop("Number of rows in
                                              voteClass must match length of trueClass")
  }
}
```

```

tryCatch({n1 = sum(trueClass)},
        silent = TRUE,
        error = function(e) {
          x = trueClass[1]
          warning(
            paste("expect trueClass to be 0-1. Set values of",
                  x, "in trueClass to 1"))
          trueClass <- (trueClass == x) + 0L
          n1 <- sum(trueClass)
        })

predClass = (voteClass >= threshold) + 0L
n2 = numTest - n1

sensitivity =
  apply(predClass[trueClass != 0, , drop = FALSE], 2, sum) / n1
specificity =
  apply(1 - predClass[trueClass == 0, , drop = FALSE], 2, sum) / n2

total = c(sensitivity, specificity,
          (n1 * sensitivity + n2 * specificity) / numTest)

return(if(length(total) == 3) {
  total } else matrix(total, nrow = 3, byrow = TRUE))
}

```

Notice that about 2/3 of the code we have written is for error handling. As mentioned earlier, if we do not plan to have our function used by others or in other applications, then extensive error handling is not needed.

Now that our functions can compute predictions and errors for a range of k s, we can compare the predictive ability for various k s. We compute these errors with

```

votes250 = nnPred(holdOutSet, trainSet, trainDiagnosis, 250, TRUE)
sensSpec = calcErrors(votes250, holdOutDiagnosis)

```

When we perform timings, we find that computing nearest neighbor estimates for all values of k from 1 to 250 takes only 25% more time than computing one the 250 NN estimate. This is a case of the efficiency of vectorization and designing our code to reuse intermediate results in the nearest neighbor computation.

We see in Figure 6.9 that small values of k tend to best for finding malignant cases. As k increases the error in predicting malignant cases rises dramatically while the error in correctly predicting benign cases shrinks to 0. This is in part a function of the proportion of benign cases in the training set. In our case, about 2/3 of the cases are benign so when we use a large number of neighbors we tend to have a benign majority. In this situation, benign cases tend to be correctly predicted and malignant cases incorrectly predicted. For these reasons, small k and weighted averages tend to make better predictions. However, as discussed earlier, we do not want to use the data we set aside to both select a model (determine k) and assess the model's predictive ability. Instead, we use cross-validation with the training set for model selection.

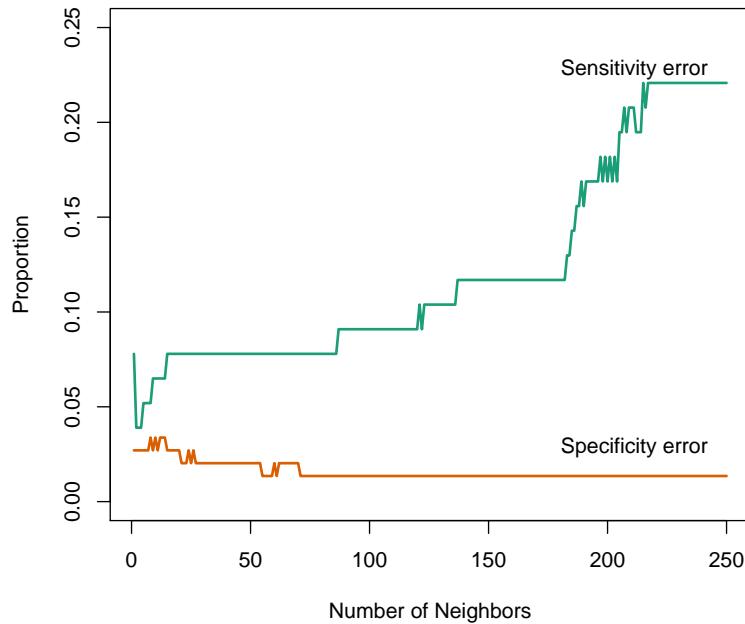


Figure 6.9: Cancer Prediction Errors for Held Out Data. *xx*

6.4.3 *v*-fold Cross Validation

The technique of cross-validation imitates the test-training set dynamic while using all of the data for both testing and training. As mentioned in XXX, we partition our data into v folds and use each fold in turn as a test set with the remaining data as the training set. If the folds are created at random then the distribution of the features and classifications in each fold should roughly follow that of the whole set.

Suppose we want to carry out 5-fold cross-validation. We begin by determining the number of observations in each fold with

```
v = 5
numInFold = floor(nrow(trainSet) / v)
numCVObs = numInFold * v
```

We use `floor()` here because `v` may not divide evenly into the number of observations. When this happens, we round down so that each fold contains the same number of observations. This means that we drop up to `v - 1` observations from the data. In our case, we do drop 4 observations. We drop these at random with:

```
set.seed(242424)
dropOnes = sample(nrow(trainSet), v-1)
trainCVSet = trainSet[-dropOnes, ]
trainCVDiagnosis = trainDiagnosis[-dropOnes]
```

Alternatively, we can work with a varying number of observations in the folds, but we need to be careful when we manipulate the folds.

We can use the `sample()` function to distribute observations into the folds. When we do not provide a sample size to `sample()`, the default is to sample all of the values supplied

so if we do not sample with replacement then we obtain a random permutation of the observations. We can use this permutation to partition the observations into the folds. We set the seed for the random number generator and permute the integers from 1 to `numCVObs` with

```
set.seed(24613)
permuteIndex = sample(numCVObs)
```

We can use these integers to index the data and create the folds. If we format the data into a matrix with `v` rows, then each row corresponds to a fold and the values within that row are the indices of the observations in that fold, i.e.,

```
CVIndex = matrix(permuteIndex, nrow = v, ncol = numFold)
```

(Note that if the folds have an unequal number of elements then the matrix is not an appropriate data structure.) Cross-validation provides a great example of the power of subsetting. For example to select the classifications for the observations not in the first fold, we use `trainCVDiagnosis[CVIndex[-1,]]`.

To carry out the cross-validation, we can use the helper functions that we developed earlier. Let's try it for the first fold. We compute the distance between the first fold observations and all of the others with

```
distFold1ToRest = calcDist(trainCVSet[CVIndex[1, ], ],
                           trainCVSet[CVIndex[-1, ], ])
```

The we find, say, the 25 nearest neighbors for that fold with

```
k = 25
fold1NN = findNN(distFold1ToRest, k)
```

We use these neighbors to vote for the diagnosis with

```
xfold1Class = trainCVDiagnosis[ CVIndex[-1, ] ]
fold1Votes = voteNN(fold1NN, xfold1Class, k, TRUE)
```

And finally, we compute the prediction error with

```
fold1Class = trainCVDiagnosis[ CVIndex[1, ] ]
fold1Errs = calcErrors(fold1Votes, fold1Class)
```

We can repeat this procedure over all `v` folds to obtain cross-validated predictions and errors for all observations.

Let's consider the distance calculations that we make for all of the folds. How many times is an observation used in these calculations? Once as a test observation and `v-1` times as a training observation. For any 2 observations not in the same fold we need to find the distance between them because at some point one will be in the test set and the other in the train set. Actually, we compute this distance twice because each observation is in the test set at some point. All together we compute $n(n - n/v)$ pairwise distances so we are duplicating effort. If we can compute all pairwise distances once, then we save $n(n - n/v) - n(n - 1)/2$ or roughly $n^2(1/2 - 1/v)$ computations. Additionally, now that we are computing all pairwise distances for `trainCVSet`, we can use the `dist()` function instead of `calcDist()`. The `dist()` function is in base R (i.e., not in a package) and so has been highly optimized. In other words, even if the number of computations were the same, `dist()` is likely to be much faster than our `calcDist()`. We can compare them with

```
system.time(replicate(25, for (i in 1:v) {
  calcDist(trainCVSet[CVIndex[i,], ],
            trainCVSet[CVIndex[-i,], ])))
user  system elapsed
24.423   0.391  26.423

system.time({replicate(25, as.matrix(dist(trainCVSet)))})
user  system elapsed
0.424   0.162   0.734
```

Indeed, `dist()` is about 35 times faster than our function. For these reasons, we compute all distances with a single call to `dist()`, i.e.,

```
allCVDistances = as.matrix(dist(trainCVSet))
```

Then, we index into `allCVDistances` to obtain those required for a fold, e.g., for the first fold, we work with

```
allCVDistances[ CVIndex[1, ], CVIndex[-1, ] ]
```

Our cross-validate function is similar to our `nnPred()`

```
cvNNPred = function(featureSet, trueClass, v, k, allK = TRUE) {

  numObs = nrow(featureSet)
  numFold = floor(numObs/v)
  numCVObs = numFold * v
  numLost = numObs - numCVObs

  dropOnes = sample(numObs, numLost)
  if (length(dropOnes) != 0) {
    featureSet = featureSet[-dropOnes, ]
    trueClass = trueClass[-dropOnes]
  }

  permuteIndex = sample(numCVObs)
  CVIndex = matrix(permuteIndex, nrow = v, ncol = numFold)
  allCVDistances = as.matrix(dist(featureSet))
  votesTrain = matrix(nrow = numCVObs, ncol = k)

  for (foldIndex in 1:v) {
    distFoldToRest = allCVDistances[ CVIndex[foldIndex, ],
                                      CVIndex[-foldIndex, ] ]
    foldNN = findNN(distFoldToRest, k)
    xfoldClass = trueClass[ CVIndex[-foldIndex, ] ]
    votesTrain[ CVIndex[foldIndex, ], ] =
      voteNN(foldNN, xfoldClass, k, allK)
  }

  return(list(votesTrain, trueClass))
}
```

We call `cvNNPred()` with our data, excluding the observations we set aside, with

```
cvPreds = cvNNPred(trainSet, trainDiagnosis, v = 5, k = 25)
```

Then we find the prediction errors with

```
sensSpec =
  calcErrors(cvPreds[[1]], cvPreds[[2]], threshold = 0.15)
```

Note that we can cross-validate the threshold as well; we leave that to the exercises. A line plot of the prediction errors appears in Figure 6.10. There we see that a value of 6 for k works well for both types of errors.

Now that we have used cross-validation to select k , we perform our final assessment of the prediction error with the hold-out data, i.e.,

```
votes250 = nnPred(holdOutSet, trainSet, trainDiagnosis, k = 6)
1 - calcErrors(votes250, holdOutDiagnosis, threshold = 0.15)

[1] 0.0130 0.0338 0.0267
```

The error for 6 NN was somewhat smaller for the malignant cases and larger for the benign cases when we cross-validated. There we had

```
1 - sensSpec[ , 6]

[1] 0.0062 0.0485 0.0340
```

At this point, we typically ask whether there patterns in the features of the cases where we made mistakes that we can use to better understand the difficult cases and improve our classification method. We leave this investigation to the exercises, but note that ensemble methods, which combine two different approaches often out perform a single prediction method. For example, we can try using recursive partitioning to classify the images. A hybrid that combines the predictions from these two approaches may be superior than either approach alone.

6.4.4 Lazy Evaluation

6.5 Summary

6.6 Guided Practice

6.7 Exercises

Bibliography

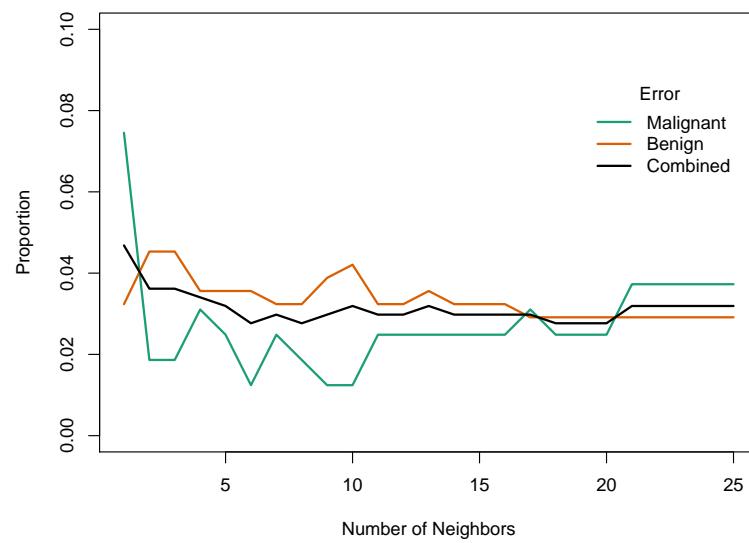


Figure 6.10: Cross-Validated Prediction Error for Cancer Diagnosis. *xx*

7

Simulation and Stochastic Modeling

CONTENTS

7.1	Introduction	301
7.2	Monte Carlo Studies	303
7.2.1	Monte Carlo Study of Magnetic Resonance Scans	304
7.3	Random Number Generation	312
7.3.1	The Cumulative Distribution Function	313
7.3.2	Congruential Methods	313
7.4	Inverse Probability Sampling	315
7.4.1	Simulating the Censored Log-Normal	316
7.4.2	Simulating the Truncated Normal Distribution	318
7.5	Acceptance-Rejection Sampling	320
7.5.1	Tail of a Distribution	323
7.6	Simulation Study of a Location Estimator for a Mixture Distribution	325
7.7	Simulating the Biham-Middleton-Levine Model for Traffic	333
7.7.1	The Initial Traffic Configuration	335
7.7.1.1	Testing the Grid Creation Function	338
7.7.2	Displaying the Grid	341
7.7.3	Moving the Cars	345
7.7.4	Evaluating the Performance of the Code	349
7.8	Summary	357
7.9	Exercises	358
	Bibliography	358

7.1 Introduction

Statisticians use simulation to learn about a random process. We simulate a system with random sampling and use the sample to provide insight into the theoretical properties of the system. The focus in this chapter is on the Monte Carlo method, which provides a direct and simple approach to simulation (see Section 7.2). To carry out any simulation, we need a random number generator (RNG), software that, as its name suggests, generates random values. We describe a simple RNG in Section 7.3 with the purpose of illustrating that RNGs are pseudo random number generators in that they generate values that look random in many ways but are actually deterministic. Most statistical software provide RNGs for a wide variety of probability distributions, and we should use them when available. In Section 7.3 we also describe the suite of RNG and related functions available in *R*. At times we need to develop our own RNG to simulate a random process, and the examples below include such cases. In these examples, we generate random values by taking advantage of relationships between the random quantity that we want to study and other more common random

variables. The examples also demonstrate how to use techniques, such as the inverse CDF method ([?]) and the acceptance-rejection method ([?]), to generate random values. When the simulation involves more real-world complexity than we can easily study analytically, an MC study can be an effective approach.

Simulation is also an excellent topic for strengthening programming skills. In addition to exploring the MC method and simulation, we also re-enforce the programming concepts introduced in Chapter 5. For the chapter examples, we map a mathematical description of a stochastic process into code and write functions to implement the simulation. In this process, we demonstrate how to use EDA of simulation results to debug our code, and we compare implementations of RNGs in terms of efficiency.

For our final example, we design and implement a simulation of a complex dynamic process ([?]). There, we focus on writing small functions in a flexible, reusable manner, and we validate the functions and combine them to create higher-level functions. We explore making the code faster in several different ways, including writing vectorized code. Finally, we carry out a simulation study to investigate the behavior of the dynamic model for different ranges of the inputs, and we consider different data structures for representing the simulation results.

Example 7-1 Magnetic Resonance Scans and the Rayleigh Distribution

Magnetic resonance (MR) imaging uses a magnetic field and pulses of radio wave energy to make pictures of structures inside the body. The data from an MR scan are typically represented as a two-dimensional array (x_j, y_j) of pixel measurements. Each pair consists of a signal (μ_1, μ_2) and noise (Z_1, Z_2) . The noise is modelled as two independent normal random variables, each with center 0 (i.e., there is no bias) and spread σ . From these MR scan data, a magnitude image is often constructed from the magnitude, $\sqrt{x_j^2 + y_j^2}$ at each pixel. What happens to the distribution of the noise in a magnitude image? It's no longer centered at 0 because the magnitude must be nonnegative. Indeed, the error in the magnitude is no longer normal, but is it close to normal? With a simulation study, we can gain understanding of the distribution of the magnitude image.

Example 7-2 Radon Levels and the Censored Log-Normal Distribution

Radon is a naturally occurring toxin that is produced by the normal decay of uranium in soil and rock. Without proper ventilation, indoor radon concentrations can reach unsafe levels. According to EPA guidelines, if radon levels exceed 4 picoCuries per liter (pCi/l) then remediation is recommended, and if these levels exceed 20 pCi/l this remediation should occur within a few months time. Radon measurements tend to follow a lognormal distribution, i.e., the log of the measurements follow a normal distribution. (Many naturally occurring quantities follow the log normal distribution.) However, the instruments used to measure radon are unable to detect radon levels below a threshold of 0.5 pCi/l. That is, concentrations below 0.5 pCi/l are reported as 0.5 pCi/l. How might this censoring affect the distribution of the measurements? How might it effect estimates of the mean and SD? We compare two approaches to generating random values from the censored log-normal distribution in Section 7.4.1 to address these questions.

Example 7-3 Close Examination of the Tail of a Distribution

In Q.7-2 (page 302), we are concerned about the upper tail of the distribution because it corresponds to high levels of radon. Often situations arise where we want to study the upper tail of a distribution in greater detail. When this is the case, we are confronted with the

problem of generating random values from only the tail of a probability distribution. We compare several approaches to this problem in [?].

Example 7-4 Performance of Estimators Under Contamination

The field of robust statistics concerns estimation in the presence of contamination. For example, we typically use the sample average to estimate the expected value (center) of a normal curve. However, when the data contain outliers (some unusually large and unexpected values), then the sample average is sensitive to these outliers and makes a poor estimate of the mean. Researchers have proposed alternatives to the simple average when the data may be contaminated. One such proposal, the Winsorized mean, censors large and small values, replacing them with a cut-off, and averages the resulting observations. It can be difficult to derive analytic results about the properties of a proposed estimator under contamination so we often resort to simulation studies to gain understanding. Since the estimator is expected to be less sensitive to outliers, we want to generate contaminated data and compare the proposal to the plain mean and to other proposals. A typical approach for creating contamination is through a mixture model where a large fraction of the time the data are generated from the true distribution and a small fraction of the time the data come from a different distribution that has atypically large values. In [?], we study the properties of the Winsorized mean and compare them to the plain and trimmed mean under a mixture model.

Example 7-5 Studying a Self-Organizing Dynamic System with a Phase Transition

The Biham-Middleton-Levine (BML) model [?] is a very simple 2-dimensional dynamic system for traffic. We can describe the model as follows. We start with an m -by- n grid of cells. Each cell can contain at most 1 car. We have two types of cars — red and blue. Each car can move 1 cell in a given time interval. The red cars move horizontally to the right; blue cars move vertically upward. When a car reaches an edge of the grid, its next potential position is to wrap around to the other side/edge of the grid. In other words, when a red car reaches the right-most cell of the grid, it next moves to the cell of the first column of the grid, and in the same row. This wrap-around motion gives the grid a torus-like connection between the edges. A car cannot move if the cell to which it would move is currently occupied. There are useful interactive demonstrations of the model at <http://www.jasondavies.com/bml/>. In [?], we simulate this simple mathematical model of a dynamic system. Through simulation and exploring the resulting data, researchers found this system to exhibit a phase transition and self-organizing behavior.

7.2 Monte Carlo Studies

Monte Carlo (MC) simulation is very simple. To study a random variable (or its probability distribution), we use a random number generator to produce a sample of independent random outcomes from this distribution. Then, we use the sample's summary statistics to approximate the expected properties of the distribution. For example, we use the average of the sample to approximate the expected value of the distribution, and we approximate a quantile of the distribution with the sample quantile of the observations. These empirical results (e.g., the sample quantiles, sample average, sample SD) do not exactly match the theoretical quantities (quantiles, expected value, SD of the probability distribution). How-

ever, with a large enough sample, they provide good approximations and can offer useful insights.

We use the MC method when it is difficult (or even impossible) to obtain analytically exact results about a stochastic quantity. This readily occurs with simple extensions of known probability distributions. For example, in Q.7-4 (page 303), we are interested in the behavior of an alternative to the sample average as an estimator for the expected value of a distribution. To study the estimator, we want to apply it to contaminated data. The contamination complicates the analysis of the theoretical properties of the proposed estimator, and we resort to a simulation study to examine the properties of this estimator.

In the next section, we provide an example of an MC simulation for a simple problem. We want to study the expected value for the noise in a magnitude image from a magnetic resonance scan.

Steps in the Monte Carlo Method

Suppose that we want to understand the properties of a random quantity. A typical example is a summary statistic of a sample, $T(X_1, X_2, \dots, X_n)$, where X_1, X_2, \dots, X_n are random variables with some known distribution. To do this, we employ the MC method as follows:

1. Generate a sample, x_1, x_2, \dots, x_n , and compute $T(x_1, x_2, \dots, x_n)$.
2. Repeat step 1, N times, yielding N sample statistics, T_1, T_2, \dots, T_N .
3. Examine the empirical distribution of the T_i s. For example, compute the sample average, sample SD, and histogram for insights into the expected value, SD, and distribution of $T(X_1, X_2, \dots, X_n)$.
4. If the distribution of the X_i depend on parameters, e.g., a normal distribution with center μ and spread σ , then we may want to repeat steps 1 through 3 for different values of the parameters to study the relationship between the distribution of T and μ and σ .

This is one example of a MC method applied to a summary statistic. We can also use the MC method to simple study the properties of independent random variables Y_1, Y_2, \dots, Y_n from some distribution, and we can study more complex dynamic processes, such as a model for traffic on a 2-dimensional grid.

7.2.1 Monte Carlo Study of Magnetic Resonance Scans

As described in Q.7-1 (page 302), we want to study the distribution of the noise in a magnitude image from a MR scan and understand its relationship to the signal. In our investigations we carry out 3 simple MC studies.

We begin with a study of the noise alone. Recall that the noise is modeled at each pixel as two independent normal random variables with expected value (center) 0 and standard deviation (SD or spread) σ . These errors are called white noise or Gaussian noise. In our first Monte Carlo study, we generate thousands of instances of this random noise and calculate the magnitude for each pair of normal errors. Since R is vectorized, we can do this simply with

```
N = 5000
z1 = rnorm(N)
```

```
z2 = rnorm(N)
x = sqrt(z1^2 + z2^2)
```

We have simulated 5,000 magnitudes from white noise, and we want to examine them as a way to understand the theoretical distribution. To do this, we can make a histogram of these simulated values with

```
hist(x, breaks = 40, freq = FALSE)
```

From Figure 7.1, we see that the distribution is skewed to the right with a peak at about 1.

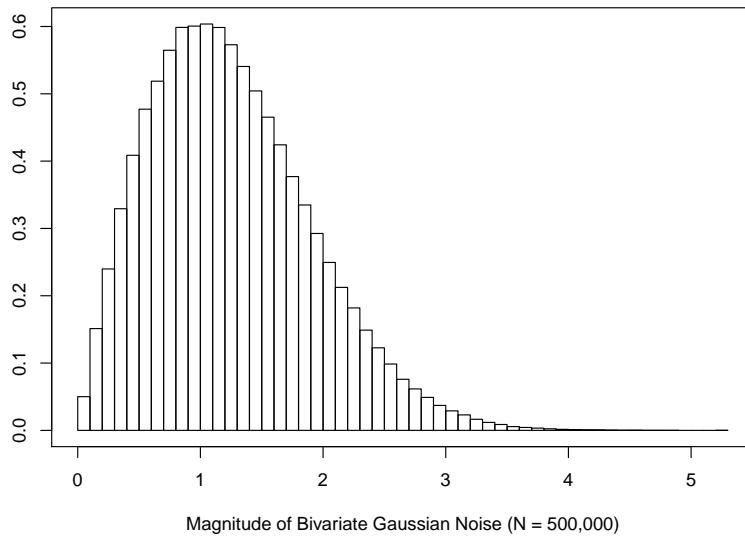


Figure 7.1: Histogram of Sample Magnitudes for Bivariate Gaussian Noise. *xx*

The histogram is an approximation to the distribution of the magnitude of bivariate normal random variables, and the average of the simulated values is an approximation to the expected value of this distribution, which is

```
mean(x)
[1] 1.256835
```

This is quite far from 0. When we do have a signal, the error introduces a bias in its measured magnitude. Before we study, the impact of this bias on various magnitudes, let's puzzle out where this mean comes from. We do not have a closed form solution for the expected value of our magnitude distribution; we have only a numeric approximation. We can guess that it is derived from constants related to the sum of squares of 2 normal random variables. Constants that come to mind are σ (which is 1 in this case), 2, and π because π is a constant in the normal density. None of these values match `mean(x)`. What about the square root of these constants? The square root of 2 and π are both too large and the square root of σ is too small. We can also consider ratios of these values, e.g. $1/\pi$, $1/2$ and $\pi/2$. These don't work either, but $\pi/2$ is about 1.57, which is closer. When we take the square root, i.e., $\sqrt{\pi/2}$, we get 1.253314. That's very close but not exactly the same as `mean(x)`.

Law of Large Numbers

Our simulated mean does not exactly equal the true mean because we are approximating the theoretical distribution with a sample of values from it. For this reason there will be some deviation between the empirical results and the theoretical value. The Law of Large Numbers is a theorem that tells us that the average of a large number of independent realizations of a random variable gets close to the expected value of the random variable as the sample size grows. The usefulness of the Monte Carlo method derives in part from this result.

What's the size of the variability of the sample average in our simulation study? We can run another MC simulation to help us quantify this variability. Before we do this, let's wrap our random number generator into a function. We call our function `rmag()`. This function has one required input—the sample size. We define `rmag()` with

```
rmag = function(n)
{
  z1 = rnorm(n)
  z2 = rnorm(n)
  sqrt(z1^2 + z2^2)
}
```

Now we can use `rmag()` in our second simulation study.

We want to investigate the variability in the average of N draws from the ‘magnitude’ distribution. We can do this by generating, say, 2000 averages, each of which is based on N samples from `rmag()`. With these 2000 sample averages, we can calculate their standard deviation, which is an approximation to the spread of the sample average. We do this with

```
avgs.many = replicate(2000, mean(rmag(N)))
sd(avgs.many)

[1] 0.009346984
```

This second MC simulation shows that our original sample average is quite close to $\sqrt{\pi}/2$. That is,

```
(mean(x) - sqrt(pi/2)) / sd(avgs.many)

[1] 0.3767043
```

We have found that the sample average is less than 1 standard deviation away from $\sqrt{\pi}/2$. Note that this does not prove that the expected value of the distribution is $\sqrt{\pi}/2$. That is the limitation of simulation studies. We don't observe exact theoretical values, only random outcomes that are close to the expected value. This leads us to another theoretical result, which demonstrates other beneficial properties of the MC method.

The Central Limit Theorem

The Central Limit Theorem (CLT) tells us that an average of a large number of independent random values from a distribution with expected value of μ and $SD = \sigma$ has an approximate normal distribution. The distribution of the sample averages is called the sampling distribution because it arises from the randomness in samples. The CLT also tells us that the expected value of the sampling distribution is μ and the SD of the sampling distribution is σ/\sqrt{N} . This SD of the sampling distribution is called the standard error (SE). We can compare the observed standard deviation of our 2000 sample averages, which we found to be 0.0093, to this theoretical value. However, we don't know the SD of the underlying magnitude distribution so we approximate it with the SD of the sample, i.e., `sd(x)`. We find

```
sd(x) / sqrt(N)
[1] 0.009174662
```

This value is very close to what we obtained by repeating our simulation 2000 times and calculating the SD of the 2000 sample averages.

We can confirm that the sampling distribution of the average is approximately normal with a normal-quantile plot of the 2000 sample averages. We do this with

```
qqnorm(avgs.many)
```

In Figure 7.2, we see a straight line, which confirms that the empirical distribution of the sample averages is roughly normal.

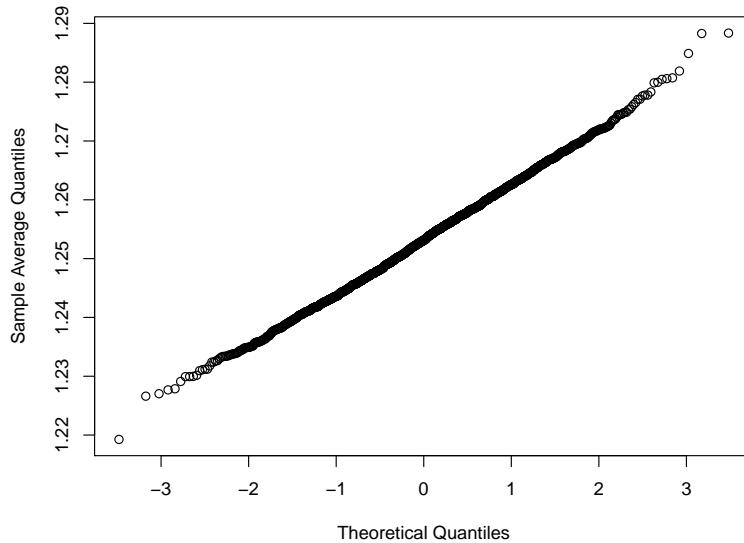


Figure 7.2: Normal Quantile Plot for the Sample Average of Magnitudes. xx

Closed-form Solutions

We have used the MC method to gain insight into the distribution of the random variable X , where $X = \sqrt{Z_1^2 + Z_2^2}$ and Z_1 and Z_2 are independent standard normal random variables. The simulation study is very useful when we do not have a closed form solution for a distribution or to its expected value (or some other statistic of interest). In fact, in his particular case, the distribution of X has been derived analytically using polar coordinates (see CITE). This distribution is called the Rayleigh distribution, and its density curve is

$$f(x) = \frac{1}{\sigma^2} e^{-\frac{x^2}{\sigma^2}},$$

for $x > 0$. There is no RNG in base R for the Rayleigh, but the VGAM package provides one. We can generate N observations from the Rayleigh with

```
library(VGAM)
x.ray = rrayleigh(N)
```

The average of this sample

```
mean(x.ray)
[1] 1.247846
```

As with our original version, the sample average is quite close to $\sqrt{\pi\sigma/2}$. Now that we know the distribution of the magnitude of the noise follows the Rayleigh, we can derive a closed form solution for its expected value. We find that it is indeed $\sqrt{\pi\sigma/2}$. (Note that even with a closed form representation of the distribution, we are not guaranteed to be able to find the expected value of tail probabilities analytically.)

Next, let's compare the efficiency of our RNG `rmag()` to the RNG provided in VGAM. To get a more accurate comparison, we generate 10 million values from each method with

```
M = 10000000
system.time({
  set.seed(a.seed)
  rmag(M)
})

user  system elapsed
2.098   0.037   2.192

system.time({
  set.seed(a.seed)
  rrayleigh(M)
})

user  system elapsed
0.599   0.009   0.630
```

Our approach takes about 4 times as long as `rrayleigh()`, but both methods are very fast. It takes about 0.01 seconds for `rrayleigh()` to generate 10 million Rayleigh values and about 0.04 for our `rmag()` function. However, even though both approaches are very fast, we should use the RNG provided in the package because it is designed by experts in random number generation and for this reason we expect that `rrayleigh()` is both more efficient and better tuned to providing numbers that more closely resemble a sample from the Rayleigh distribution.

Finally, we turn to the problem of determining the impact of Rayleigh errors (since they are not mean 0 and they are skewed right) on the distribution of magnitude measurements in the presence of a signal.

Recall that the model for the measurement at a pixel is that it consists of a signal (μ_1, μ_2) plus Gaussian errors (Z_1, Z_2) . For example, suppose $\mu_1 = 1$ and $\mu_2 = 0.5$, then the magnitude is

$$\sqrt{(1 + Z_1)^2 + (0.5 + Z_2)^2}.$$

We know the magnitude of the signal, with no noise present, is $\sqrt{1^2 + 0.5^2}$ or about 1.118. What does the magnitude look like for our sample?

We can update our function `rmag()` to accept signals as an input, e.g.,

```
rmag = function(n, s1, s2)
{
  z1 = rnorm(n)
  z2 = rnorm(n)
  sqrt((s1 + z1)^2 + (s2 + z2)^2)
}
```

The average measured magnitude for the signal based on N (5000) measurements is

```
x1.h = rmag(N, s1 = 1, s2 = 0.5)
mean(x1.h)

[1] 1.624873
```

and we find the standard error for the average to be roughly,

```
sd(x1.h) / sqrt(N)

[1] 0.01130063
```

This implies that the observed average is far from the signal magnitude; they are nearly 50 SEs away from each other. The bias is about 0.5, which is smaller than the expected value when there is no signal, but still considerable in size.

Let's take a more systematic effort at studying the size of the bias and carry out another MC simulation. This time, we create a set of possible signals and run our simulation for each pair. Then we can see how the average measurement deviates from the signal magnitude as we vary the signals. Let's begin with defining several pairs of signals with

```
mu1.2 = matrix(c(0.5, 0.5, 0.5, 1, 0.5, 2, 0.5, 4, 0.5, 8,
                 1, 1, 1, 2, 1, 4, 1, 8,
                 2, 2, 2, 4, 2, 8, 4, 4, 4, 8, 8, 8),
                 ncol = 2, byrow = TRUE)
head(mu1.2)

[,1] [,2]
[1,] 0.5  0.5
[2,] 0.5  1.0
[3,] 0.5  2.0
[4,] 0.5  4.0
[5,] 0.5  8.0
[6,] 1.0  1.0
```

For each of these combinations, we want to use the same N pairs of errors to make the comparisons more similar. We can do this simply, if our function `rmag()` accepts vector inputs. Do we need to rewrite our function?

There is no need to change our function's signature, but we need to make sure that our code works with vector inputs. Fortunately, we generate the errors first and then add the signal to the noise so we can apply the calculation of the magnitude over the signal pairs, e.g.,

```
rmag = function(n, s1, s2)
{
  z1 = rnorm(n)
  z2 = rnorm(n)
  mapply(function(mu1, mu2) {
    sqrt((mu1 + z1)^2 + (mu2 + z2)^2)
  }, mu1 = s1, mu2 = s2)
}
```

We have used `mapply()` in `rmag()` because we have two arguments in our function to calculate the magnitude.

Let's call our new version of the function, passing it our set of signals. We do this with

```
sampleMags = rmag(N, s1 = mu1.2[, 1], s2 = mu1.2[, 2])
```

The return value from this function is a matrix with `N` rows and a column for each signal combination. We confirm this with

```
dim(sampleMags)
```

```
[1] 5000 15
```

For each pair of signals, we find the sample average with

```
sampleMagMeans = apply(sampleMags, 2, mean)
```

and we want to compare these to the signal magnitudes which we calculate with

```
signalMags = apply(mu1.2, 1, function(x) sqrt(x[1]^2 + x[2]^2))
```

To compare the sample averages against the desired magnitudes, we can plot their differences against the magnitude with no noise. We also add error bars that are 2SEs from the differences so that we can assess the size of these deviations. We use the `plotCI()` function in the `plotrix` to do this with

```
ses = apply(sampleMags, 2, sd) / sqrt(N)
upper = (sampleMagMeans - signalMags) + 2 * ses
lower = (sampleMagMeans - signalMags) - 2 * ses

require(plotrix)
plotCI(x = signalMags, y = sampleMagMeans - signalMags,
       ui = upper, li = lower,
       xlab = "Signal Magnitude",
       ylab = "Difference in Simulated and True Signal Magnitude")
```

From Figure 7.3, we see that the bias decreases as the signal increases. We might expect the magnitude to decrease like

$$\sqrt{x^2 + 2 * \sigma^2}.$$

In this case, the differences would fall along the curve

```
curve(sqrt(x^2 + 2) - x, add = TRUE)
curve(sqrt(x^2 + 1) - x, add = TRUE)
```

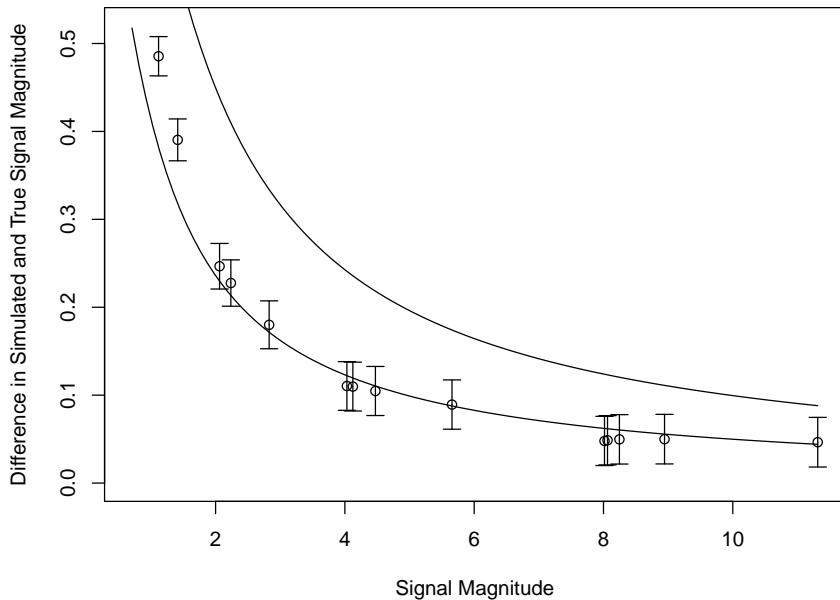
We also places $\sqrt{x^2 + \sigma^2}$ on the plot, and it does a better approximation. It's not correct, but appears to be a good approximation for large signal in comparison to the noise.

Lastly, let's examine the distributions of these magnitudes. Many of the magnitudes are close to each other so we choose a few that cover the range of magnitudes observed, e.g.,

```
idx = c(1, 2, 3, 11, 13, 12, 16)
signals[idx]
```

```
[1] 1.4142 2.2361 4.1231 5.6569 8.0623 8.9443 11.3138
```

We make normal quantile plots for these signal magnitudes.

Figure 7.3: Bias in the Sample Average of Magnitudes. *xx*

```

plot(0, ylim = c(0, 15), xlim = c(-4, 4), type = "n",
xlab = "Theoretical Normal Quantiles",
ylab = "Observed Signal Quantiles")
apply(samples[ ,idx], 2,
function(samp) {
  qs = qqnorm(samp, plot.it = FALSE)
  points(x = qs[["x"]], y = qs[["y"]], pch = 19, cex = 0.3)
})

ys = apply(samples[ , idx], 2, max)
text(x = 4, y = ys - 0.5, pos = 2,
      labels = round(signals[ idx ], digits = 1))

text(x = 4.2, y = 3.5, pos = 2, labels = "Signal\nMagnitude")

```

We see in Figure 7.4 that for a small signal to noise ratio (SNR) the distribution is skewed, but as the SNR ratio increases, the distribution appears close to normal.

Guidelines for Simple Simulation Studies

When we carry out a simulation study we keep the following in mind:

- When possible, we use the RNG functions provided in the software rather than develop our own because the standard functions tend to be more efficient and better at generating random looking samples.
- When a RNG is not available for our situation, developing a RNG that is based on transformations of random variables with built-in RNGs is often a good approach.

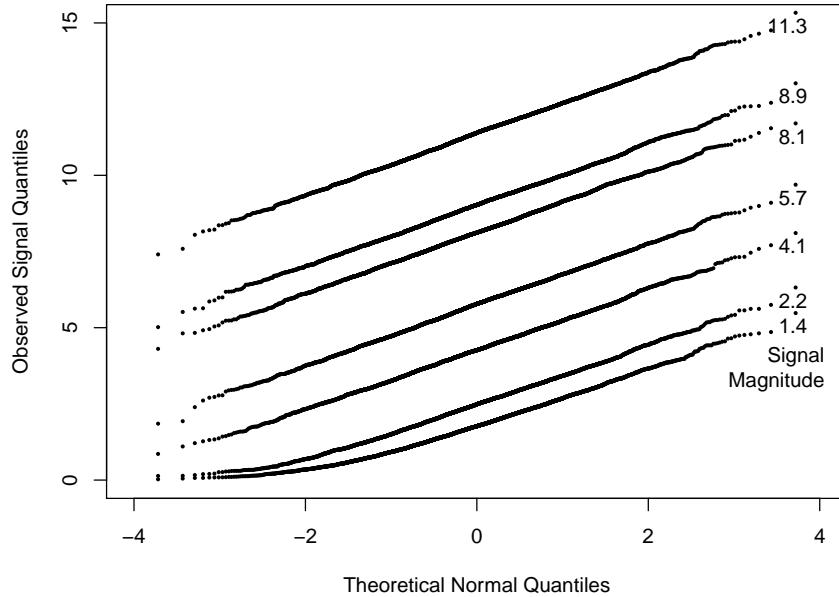


Figure 7.4: Bias in the Sample Average of Magnitudes. *xx*

- Simulation studies are not mathematical proofs, but they can give good guidance and intuition. They are especially useful when closed form solutions are difficult or impossible to obtain.
- The LLN tells us that a sample average based on a large sample of independent, identically distributed random values approximates the expected value of the theoretical distribution.
- The CLT tells us that the sampling distribution of the average is approximately normal with an SE of σ/\sqrt{N} , for large N .

PROGRAMMING SIDE BAR?

7.3 Random Number Generation

Random number generation is so important that it is an active field of research in statistics and computer science and statistical software provide random number generators for many common distributions.

The core RNG is the uniform random number generator. For the discrete uniform, if x_1, \dots, x_m are the possible outcomes, then each is equally likely, i.e., the chance of observing x_i is $1/m$ for $i = 1, \dots, m$. With the continuous uniform distribution on the range (a, b) , the chance of a random outcome falling in an interval of length l is $l/(b - a)$ for all intervals in (a, b) that have length l . Given the discrete nature of the computer, we can't truly generate values from the uniform continuous distribution, but we do not concern ourselves with this limitation here.

In *R*, we can generate values from many well-known, discrete distributions, such as the binomial, Poisson, geometric, hypergeometric, negative binomial, and multinomial with `rbinom()`, `rpois()`, `rgeom()`, `rhyper()`, `rnbinom()`, and `rmultinom()`, respectively. To generate values from the discrete uniform, we use the `sample()` function. RNGs for the standard continuous distributions, such as the uniform, normal, exponential, chi-squared, F, Beta, Cauchy, etc, are available with functions `runif()`, `rnorm()`, `rexp()`, etc.

The standard in *R* is to provide functions for calculating the density function (or probability mass function), percentiles, and quantiles, in addition to the random number generator for the distribution. The convention is to name these, e.g., `dnorm()`, `pnorm()`, `qnorm()`, and `rnorm()`, respectively.

7.3.1 The Cumulative Distribution Function

For completeness, we provide a brief background on probability distributions. A probability distribution is uniquely defined by its cumulative distribution function (CDF), which provides probabilities of the form, $\mathcal{P}(X \leq t)$. By convention the CDF is represented by F . For continuous distributions, the CDF is differentialve and its derivative, f is the density function. This implies the relationship between F and f :

$$F(t) = \int_{-\infty}^t f(x)dx.$$

In *R*, the `dxxx()` function is the density function f , the `pxxx()` function is the CDF F , and `qxxx()` is the quantile function. In the case where the CDF is invertible, `qxxx()` is the inverse CDF, F^{-1} .

For discrete probability distributions, the CDF is a step function. The location of the jumps correspond to the possible values and the size of the jumps to the chance of observing that value. In this case the `dxxx()` function is the probability mass function. Of course, distributions can be continuous for some ranges and discrete in others. One example appears in Section 7.4.1.

Expected Value and SD

The expected value of a random variable X that has probability distribution with CDF F and density function f is defined as

$$E(X) = \int_{-\infty}^{\infty} xf(x)dx,$$

and for discrete distributions, we have

$$E(X) = \sum_{i=1}^m x_i Pr(X = x_i).$$

The SD is also defined as

$$SD = \sqrt{E((X - E(X))^2)} = \sqrt{E(X^2) - (E(X))^2}.$$

7.3.2 Congruential Methods

In this chapter, we refer to the generators `runif()`, `rnorm()`, `rbinom()`, etc. as random number generators even though they technically are pseudo-random number generators. These

pseudo-random number generators rely on well tested code for generating values that appear as if they are random samples. In fact, these values are generated recursively from a deterministic algorithm.

One of these algorithms is the congruential method. The basic idea behind it goes as follows:

- Begin with an initial value, also called a seed, say x_0 , and constants b and m .
- Generate the first value in the sequence with

$$x_1 = bx_0 \mod m.$$

- Continue to recursively generate values, for $i = 2, \dots, n$ with

$$x_{i+1} = bx_i \mod m.$$

It is clear from this definition that when we start with the seed x_0 , we always get the same sequence of numbers. This is considered an advantage because we (and others) can reproduce a sequence of ‘random’ values when checking and sharing our work.

Let’s try this algorithm for the parameters:

```
b = 3
m = 64
seed = 17
```

Then the initial value in the sequence is

```
(b*seed) %% m
[1] 51
```

We can generate, say 200, values with a loop, e.g.,

```
x = vector(mode = "numeric", length = 200)
x[1] = (b*seed) %% m
for (i in 2:n) {
  x[i] = (b*x[i-1]) %% m
}
```

The first 20 values:

```
[1] 51 25 11 33 35 41 59 49 19 57 43 1 3 9 27 17 51 25 11 33
```

look far from random. The sequence repeats itself on the seventeenth value. This is always the case with the congruential method, and for any other recursive algorithm used in pseudo-random number generation. The length of the non-repeating sequence is called the period.

Fortunately, when the period is long and the sample size is smaller than the square-root of the period then the values look roughly normal, provided b and m are well chosen. If we take $b = 48271$ and $m = 2^{30} - 1$, then the sequence is much better behaved. How can we tell if the numbers look random? One simple way is to plot successive pairs of values. There should be no apparent pattern in the scatter of points. For our simple example, this is clearly not the case for the 200 numbers generated with our earlier choice for the multiplier and modulo, as seen in the plot on the left of [?]. There are only 16 distinct points because the sequence has a small period, and these points fall in stripes across the plotting region. On the other hand, the plot on the right shows no apparent pattern or repetition.

In practice, algorithms are subject to a battery of tests for independence before being adopted. The default RNG in R is the Mersenne-Twister CITE, and several others are also made available.

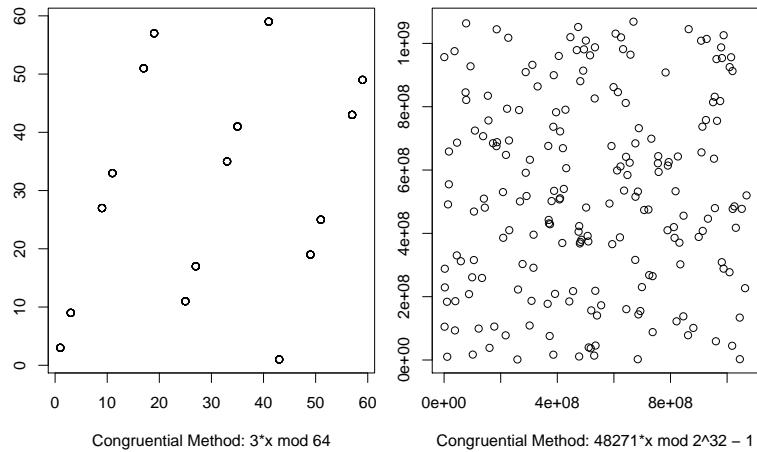


Figure 7.5: Scatter Plot of Values Generated by the Congruential Method. xx

Pseudo-Random Number Generator

A pseudo-random number generator uses a deterministic algorithm to generate a sequence of numbers. These sequences appear random in that they pass many tests for randomness.

An RNG algorithm are recursive in that it uses a state to generate a number and update the state for the next number, i.e., in pseudo-code `state = update(state)` `number = gen(state)`. An initial state is needed to start this process; typically it can be set with a ‘seed’.

The pseudo-RNG has several advantages. By setting the seed,

- The sequence can be replicated exactly, which can help debug code.
- The results from a simulation are reproducible.
- There is no need to store the sequence of numbers to document results.
- We can synchronize multiple simulations.

7.4 Inverse Probability Sampling

At times we want to generate random values from a probability distribution that is not supplied by the software environment. If the cumulative distribution function has an inverse in closed form, then there is a simple procedure for generating random values from the distribution. Recall that a CDF provides the tail probabilities,

$$F(t) = Pr(-\infty, t]$$

i.e. the chance that the random value is less than or equal to t . For the uniform distribution, the CDF is the identity on $(0, 1)$,

$$F_{\text{uniform}}(t) = \begin{cases} 0 & t \leq 0 \\ t & 0 < t < 1 \\ 1 & t \geq 1 \end{cases}$$

If we can generate random uniform values then we can use the inverse function (F^{-1}) of a CDF to create random values from the probability distribution with CDF F . The procedure is very simple:

- Generate a random value, U , from a standard uniform distribution, e.g., `U = runif(1)`.
- Compute $Y = F^{-1}(U)$.

This random variable Y has F for its CDF.

It is reasonably easy to verify that this is the case.

$$\begin{aligned} \Pr(Y \leq t) &= \Pr(F^{-1}(U) \leq t) \\ &= \Pr(F(F^{-1}(U)) \leq F(t)) \\ &= \Pr(U \leq F(t)) \\ &= F(t) \end{aligned}$$

In this derivation, we use the fact that F is monotonically non-decreasing and has an inverse (in the second equality).

A simpler or more intuitive way to think about why Y has the CDF F is the following. We are sampling a percentile at random when we pick a number U between 0 and 1. Then we ask for the corresponding quantile of the distribution with CDF F , i.e. the value t such that $F(t) = u$. This quantile is easily obtained by using the inverse of CDF so we are simply matching quantiles of the uniform distribution with the quantiles of F . Randomly sampling the quantiles in $(0, 1)$, we get random samples from $F(t)$.

In the next section, we have a case where there is no built-in random number generator, but we do have a function that well approximates the inverse CDF.

Guidelines for the Inverse CDF Method

The inverse CDF method is very simple. Suppose `iCDF()` is the inverse of the CDF of the probability distribution from which we want to generate N random values. Then, we generate these values with

```
x = iCDF(runif(N))
```

The inverse CDF method works well when

- A random number generator for the desired distribution is not available.
- We have a closed form solution to the inverse CDF (or good approximation to it).
- Evaluation of $F^{-1}(u)$ is not computationally expensive.

7.4.1 Simulating the Censored Log-Normal

As noted in Q.?? (page ??), we are interested in the behavior of the measurements of indoor radon concentrations. These scanning measurements are made by a device that collects air samples over a 3 to 7 day period. Radon concentration measurements tend to follow a log-normal distribution; that is, the log of the measurements have a normal distribution. Also, according to CITE, the error in the scanning process has a log normal distribution, where the log of the errors have a $\mathcal{N}(0, 0.2^2)$ distribution. Before we embark on a simulation study on the impact of censoring, let's get a better sense of the log normal distribution. Figure 7.6 displays the density curve of the normal, which we create with

```
curve(dnorm(x, mean = 0, sd = 0.2), from = -0.8, to = 0.8)
```

The center and right plots in that figure show a histogram of 5,000 observations from the corresponding log-normal distribution and a normal quantile plot for these observations. We generated the log-normal values from the normal distribution with

```
x5k = exp(rnorm(5000, sd = 0.2))
```

We use `x5k` to create the histogram and normal-quantile with

```
hist(exp(x5k), breaks = 40, freq = FALSE)
qqnorm(exp(x5k))
```

We see the log-normal is a right skewed distribution.

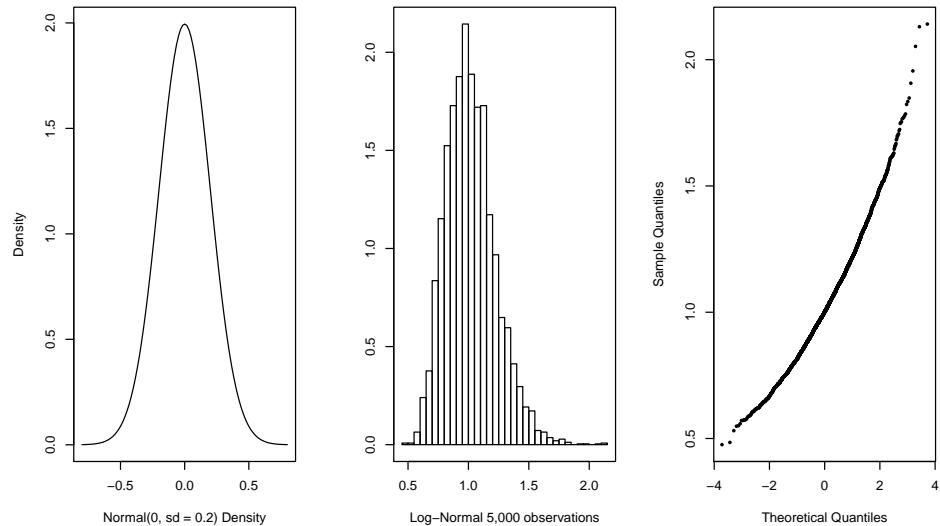


Figure 7.6: Relationship Between the Normal and Log-Normal Distributions. *xx*

We want to investigate the impact of censoring measurements below 0.5 pCi/l. Given the relationship between the normal and log-normal, we often work with the normal distribution for simplicity. *R* does not provide an RNG for censored log-normal/normal distributions, but we can easily simulate these values with a 2-step process: a) generate random values from the normal distribution; then b) censor the values that fall below the cutoff.

First we identify the necessary parameters for our simulation. In our example, we specified the mean and SD of the normal (which map to a mean and SD for a log-normal). These are:

```
mu = 0
sigma = 0.2
```

We also need to specify the cutoff. In our radon example, this is `cutPoint = log(0.5)`.

In practice, we observe a small number of observations, but for our study, we want to examine the impact of the truncation on the distribution and its mean so we generate a large number of values. Let's take this number to be, say 100 thousand, and let's make it a parameter too, e.g., `N = 100000`. Now, we can generate data from the normal, and replace the values that fall below `cutPoint` with

```
x = rnorm(N, mean = mu, sd = sigma)
x[x < cutPoint] = cutPoint
```

How many of these 100000 values were modified? We can determine this with

```
sum(x <= cutPoint)
[1] 24
```

This is not all that surprising because the chance of an observation falling below the cut off is

```
F_cut = pnorm(cutPoint, mean = mu, sd = sigma)
F_cut
[1] 0.000264
```

The impact of the impact of censoring seems limited in our specific example. We leave it to the exercises to carry out a more thorough investigation of the impact of censoring on the log-normal distribution and its mean.

An alternative to this approach takes a different set of two steps: a) generate the number of observations to censor; and then b) generate only the uncensored values. At times we observe only values above the cutoff and we don't know how many, if any fall below the cutoff. When this happens, we describe the distribution as truncated. In this case, we want to generate only the uncensored values in part b). The inverse CDF method works particularly well in this case. In the next section, we use the inverse CDF method to generate truncated data, and then return to the generating censored data using this alternative.

7.4.2 Simulating the Truncated Normal Distribution

As just described, the truncated distribution is similar to the censored distribution with the important difference that we observe values only from that portion of the distribution above a cutoff. A more general version of truncation provides both upper and lower cutoffs, say a and b with $a < b$, and we observe only values between a and b .

Our goal is to write a function, called `rtruncnorm()` that generates random values from a truncated normal distribution. What are the inputs to this function? We need to know: the number of samples, the mean and sd of the normal, and the cut off points. Which of these inputs are required and which have default values? Let's take `xxx` from the RNGs for the normal and uniform distributions, `rnorm()` and `runif()`, and require only the sample size, `n`. For the rest of the parameters, we take, `mean = 0` and `sd = 1`. And, we take the default to be no truncation so `a = -Inf`, and `b = Inf` so our function definition is

```
rtruncnorm = function(n, mean = 0, sd = 1, a = -Inf, b = Inf)
```

The inverse CDF method generates uniform random values and applies the quantile function to them, e.g.,

```
qnorm(runif(n), mean, sd)
```

How do we adapt this to the situation where we have a truncated distribution? Intuitively, we can generate random values uniformly from the interval (q_a, q_b) , where $q_a = F(a)$ and $q_b = F(b)$. Our function is then defined as

```
rtruncnorm =
function(n, mean = 0, sd = 1, a = -Inf, b = Inf)
{
  q.a = pnorm(a, mean, sd)
  q.b = pnorm(b, mean, sd)
  u = runif(n, min = q.a, max = q.b)
  return(qnorm(u, mean, sd))
}
```

Of course, we can use the approach from [?] and generate values from the normal and discard those that fall outside the desired range. The only problem with this approach is that we don't know how many normal values to generate because we don't know how many will be discarded. This means that we need to track the number that we generate and continue generating values until we have reached the desired `n`. We can begin by generating more than `n`. A reasonable number might be 10% more than the expected number required to produce `n` in the desired range, i.e., $1.1 * n / (q_b - q_a)$. In the function below, we take this approach.

```
rtruncnorm.2 =
function(n, mean = 0, sd = 1, a = -Inf, b = Inf)
{
  q.a = pnorm(a, mean, sd)
  q.b = pnorm(b, mean, sd)
  z = rnorm((1.1 * n / (q.b - q.a)), mean, sd)
  z = z[z >= a & z <= b]
  while (length(z) < n) {
    x = rnorm(ceiling(2 * (n - length(z)) / (q.b - q.a)),
               mean, sd)
    z = c(z, x[x >= a & x <= b])
  }
  return(z[1:n])
}
```

Let's compare the timings for these two functions. We take the default values for the mean and `sd`, and simply use a lower cutoff. We set up a range of lower values with

```
a.vec = seq(-2.5, 2.5, by = 0.25)
```

For each of these cutoffs, we generate 1 million values with

```
N = 1000000
```

```

st1 = sapply(a.vec, function(a) {
  set.seed(222222)
  system.time(rtruncnorm(N, a = a)) [2]
})

st2 = sapply(a.vec, function(a) {
  set.seed(222222)
  system.time(rtruncnorm.2(N, a = a)) [2]
})

```

We examine the ratio of the 2 system times with

```
summary(st2/st1)
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
3.6	4.5	9.0	175.1	31.0	2380.0

We see that the range of this ratio is tremendous; the inverse CDF method is about 4 times as fast for cutoffs more than 1 SD below average, 9 times as fast when we throw out about 1/2 the values, and more than 2000 times faster when the cutoff is above 2. A plot can better help us see how this ratio changes with the cutoff. We make our plot on a log scale to make the relationship easier to examine. If the times are the same then we should see values at 0. If the iCDF is consistently 4 to 5 times faster, then we see a horizontal line at about 1.5.

```
scatter.smooth(log(st2/st1) ~ a.vec, span = 1/3 )
```

The inverse CDF methods is about 5 times as fast as the other method for cut offs that correspond to discarding less than 50% of the values. After that, the relative speed of the normal approach increases exponentially.

Lastly, we return to our censored example and use the inverse CDF method to generate truncated random values in the 2 step process: a) generate the number of observations to censor; and then b) generate only the uncensored values. We use `rtruncnorm()` for the second step, and for the first, we can generate the number of censored values from the binomial. That is, the number of censored values has a binomial distribution with parameters `N` and chance of success `F_cut`.

```

truncNum = rbinom(1, size = N, prob = F_cut)
z = rtruncnorm(N - truncNum, mean = mu, sd = sigma, a = cutPoint)

```

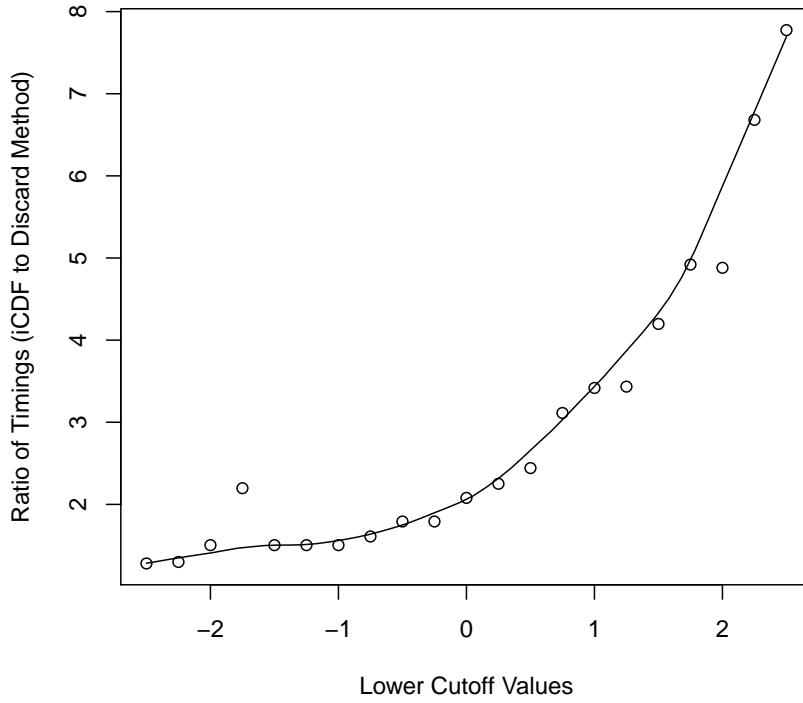
We can create the full sample with the truncated values with

```
sample(c(z, rep(cutPoint, truncNum)))
```

Note that we used `sample()` to randomize the order of the values.

7.5 Acceptance-Rejection Sampling

We don't always have the CDF available to us, or it may not always be invertible. If we have a continuous density function for our distribution and we can bound this density function above by a density function that we can sample from, then we can use the acceptance-rejection method for generating random values .

Figure 7.7: Distributions. *xx*

The acceptance-rejection method is an alternative to the inverse CDF method when we do not have a readily invertible CDF, but we have a density function. The idea is quite simple and intuitive:

- Sample uniformly at random from a two-dimensional region that encloses the density function.
- Keep (accept) only the points within our area of interest and drop (reject) those that fall outside this region.
- The x -coordinates of the accepted points form the sample.

This method depends on two basic notions. The first idea connects probability to area. That is, consider points in two-dimensions that are scattered randomly on a region, called \mathcal{A} , in the plane. A uniform scatter means that the probability of a point falling in a sub-region $\mathcal{B} \subset \mathcal{A}$ is the same for all subregions of the same area, and this chance is $\text{Area}(\mathcal{B})/\text{Area}(\mathcal{A})$.

We apply this notion to a special region: the two-dimensional region bounded by the x -axis and $y = f(x)$, i.e.,

$$\mathcal{A} = \{(x, y) : 0 \leq y \leq f(x)\},$$

And the subset that we look at is the region,

$$\mathcal{B}_t = \{(x, y) : 0 \leq y \leq f(x) \text{ and } x \leq t\}$$

See Figure XXX for an example. If a point is sampled in two-dimensions at random from \mathcal{A} , then the x -coordinate of the point follows the probability distribution with pdf f and

CDF F . To see why this is the case, we need only determine the CDF of the x -coordinate of the random-pair (X, Y) . We establish this with

$$\begin{aligned} \Pr(X \leq t) &= \Pr((X, Y) \in \mathcal{B}_t) \\ &= \text{Area}(\mathcal{B}_t)/\text{Area}(\mathcal{A}) \\ &= \int_{-\infty}^t f(x)dx \\ &= F(t) \end{aligned}$$

We see from this derivation that if we can generate data from the uniform distribution over \mathcal{A} , then we have a random sample from the distribution of interest. Unfortunately, it is typically the case that if we can generate samples uniformly from \mathcal{A} then we would also be able to generate samples via the inverse CDF method. The second basic property helps us out of this bind.

This notion is that when we take a random sample of n points from a larger region, say \mathcal{C} that contains \mathcal{A} , and discard all of those points that do not fall into \mathcal{A} , then the remaining points form a uniform random sample in \mathcal{A} . This means that if we want a random sample of 100 points from \mathcal{A} , we generate a point from \mathcal{C} , reject it if it is outside of \mathcal{A} , accept it if it is inside \mathcal{A} , and continue in this way until 100 points have been accepted.

It is important to choose the enveloping region to be as small as possible and still contain \mathcal{A} . This way we reject fewer observations and our generator is more efficient. In the next section, we apply this technique to generate random values from the tail of a normal distribution.

An Algorithm for the Acceptance-Rejection Method

The goal is to generate random values from a probability distribution with density f . Suppose we have a second probability density function, say $g()$, and we can generate random values from this distribution. Take c to be a constant such that for all x :

$$f(x) \leq cg(x).$$

Then we can generate a random value from the desired probability distribution as follows:

1. Generate a random value, X , from the distribution with density $g()$.
2. Generate a uniform random value, U from the interval $(0, cg(X))$.
3. Accept X if $U \leq f(X)$. Otherwise, return to Step 1.

Example 7-6 The Beta and Triangular Distributions

We provide a simple example with the $\text{Beta}(4, 3)$ distribution. With some trial and error, we can bound the density function for the Beta by a triangular density function with peak at 0.8, if we scale it by 1.6. We plot these two densities with

```
library(triangle)
curve(1.6 * dtriangle(x, c = 0.8), from = -0.05, to = 1.05,
      ylab = "y")
curve(dbeta(x, 4, 3), add = TRUE)
```

We see in XXX that the scaled triangular density sits above the Beta.

We follow the algorithm and generate `n` values from this triangular distribution with:

```
n = 10000
xVal = rtriangle(n, c = 0.8)
```

And, we generate `n` values from the uniform distribution with

```
yVal = 1.6 * dtriangle(xVal, c = 0.8) * runif(n)
```

The accepted values are

```
accept = yVal < dbeta(xVal, 4, 3)
```

Now the subset `yVals[accept]` follows the $Beta(4,3)$ distribution.

In the center plot of XXX, we plot the `xVal` and `yVal` pairs, and color code them to indicate which ones are accepted. We make this plot by adding points to the previous plot with

```
points(x = xVal, y = yVal,
       col = c("red", "grey")[accept + 1],
       pch = 19, cex = 0.1)
```

The right plot shows the histogram of accepted elements in `xVal`, overlaid with the Beta density curve. It confirms that the distribution is as expected.

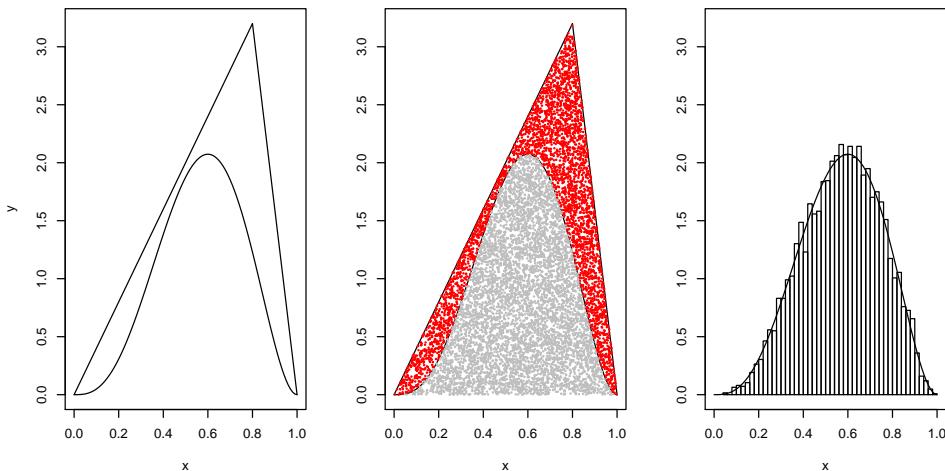


Figure 7.8: Acceptance Rejection Method. *xx*

7.5.1 Tail of a Distribution

In this section, we generate data from the tail of a normal distribution. This problem can be cast as a truncation problem so we can use the inverse CDF method and the `rtruncnorm()` function we developed in Section 7.4.1. Of course, we can't actually invert the CDF of the normal distribution and the percentile function `pnorm()` approximates the inverse CDF

when it calculates the percentiles. The accuracy of this approximation may be problematic in the extreme tails of the distribution. Since we do have a closed form representation of the normal density function, we can use the acceptance-rejection method (ARM), instead.

The ARM requires a density function that dominates the density function that we want to sample from. We can use the exponential curve because it decays like $\exp(-x)$ and the normal tails are $\exp(-x^2)$. However, we do need to normalize the exponential curve so that it sits above the normal tail. For a tail that begins at `shift`, we can do this with the scale factor

```
c = 1/sqrt(2*pi) * exp(-shift^2 / 2)
```

and by shifting the exponential by `shift`. In other words, we can bound the tail of the normal by `c * exp(-shift * (x - shift))`. We show this for `shift = 2.5` with

```
curve(dnorm(x), from = shift, to = shift + 2.5, lwd = 2)
curve(c * exp(-shift * (x - shift)),
      from = shift, to = shift + 2.5,
      add = TRUE, lty = 2, lwd = 2)
```

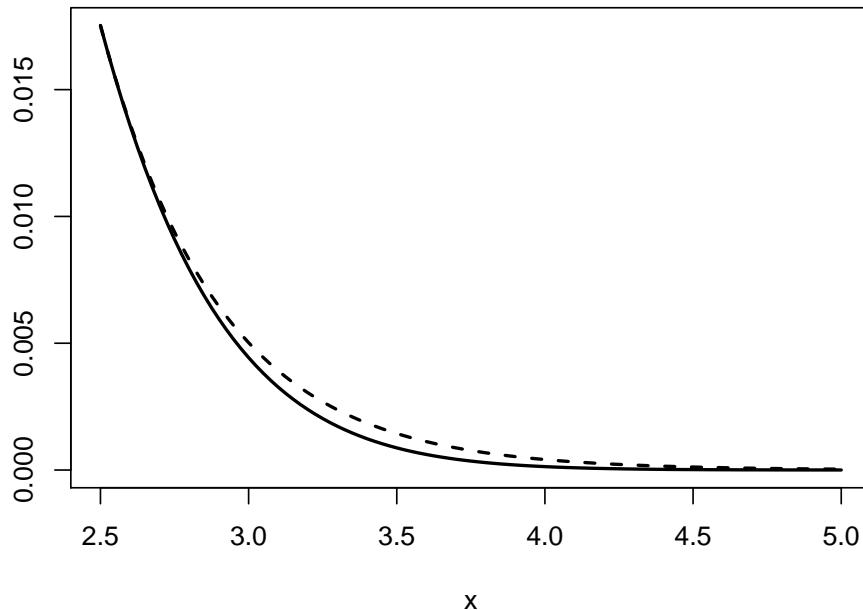


Figure 7.9: Acceptance Rejection Method. *xx*

As in Q.?? (page ??), the first step is to generate data according to the (shifted) exponential distribution, e.g.,

```
n = 2000
xVal = shift + rexp(n) / shift
```

Then we generate `n` values from the uniform distribution with

```
yVal = c * dexp(shift * (xVal - shift)) * runif(n)
```

We accept only those elements in `yVal` that fall below the normal curve, i.e.,

```
accept = yVal < dnorm(xVal)
```

We add these points to the plot, using different plotting symbols so that we can differentiate between the accepted and rejected.

```
points(x = xVal, y = yVal,
       col = c("red", "grey") [accept + 1], cex = 0.3)
```

These points look uniformly scattered in the region below the dominating function. The accepted points should look like a random sample from the upper tail of the normal distribution.

With this exponential, the proportion of values that are accepted is very high,

```
sum(accept) / n
[1] 0.8805
```

We can wrap this code up into a function with

```
rexpbound =
function(n, shift)
{
  c = 1/sqrt(2*pi) * exp(-shift^2 / 2)
  xVal = shift + rexp(n) / shift
  yVal = c * dexp(shift * (xVal - shift)) * runif(n)
  return(xVal[yVal < dnorm(xVal)])
}
```

If we want to generate a specific number of values, then we will need to call `rexpbound()` multiple times until we have enough samples. We can put this decision making code into another function to keep these two tasks separate. We do this with

```
rnormtail =
function(n, above = 2,
         factor = 1.1 / (above * exp(above^2 / 2) *
                           sqrt(2 * pi) * pnorm(-above)))
{
  Xs = rexpbound(n * factor, above)
  while (length(Xs) < n) {
    m = length(Xs)
    moreXs = rexpbound((n - m) * factor, above)
    Xs = c(Xs, moreXs)
  }
  return(Xs[1:n])
}
```

We leave it to the exercises to compare this technique to `rtruncnorm()` from [?].

7.6 Simulation Study of a Location Estimator for a Mixture Distribution

In Q.7-4 (page 303), our goal is to conduct a simulation study that compares the effectiveness of the Winsorized mean in estimating the expected value of a distribution with contaminated data. For our study, we use a mixture distribution to capture contamination; here a large fraction of the data are from the true distribution and on occasion, at random and without our knowledge, large values are observed from a second distribution.

We embark on this programming task by first identifying the steps and subtasks that we need to accomplish. As the project evolves, we will further refine our approach and rewrite our code and functions to make them more general. The currently identified tasks are:

Organizing a Simulation Study

- Develop an RNG for the contamination distribution.
- Test and refine our RNG before using it in the simulation study of the location estimator.
- Design the simulation study, which includes identifying the following
 - Location estimators to compare in the study and their parameter values, if any.
 - Distributions for generating contaminated data.
 - Sample sizes.
 - Number of replications for each combination of estimator, contamination distribution, and sample size.

This may require a few initial simulations to identify useful and feasible values for these quantities.

- Conduct the study
- Analyze the results

An Initial Mixture Distribution

We begin by writing code for a specific example before we generalize it to a function. For our example, let's take the true distribution to be the standard normal and the amount of contamination to be 5%. For the contamination, let's try a normal with mean 20 and SD 3. For a sample of 100, most of these values will come from the standard normal and a few come from the contaminated distribution. We can think of it as a 2-step process where, e.g., a marble is drawn from an urn with 95 white and 5 black marbles. If the marble drawn is white then we observe a random value from the standard normal, otherwise we observe a value from the $\mathcal{N}(20, 3^2)$ distribution. We can generate the draws from the urn with the Bernoulli distribution,

```
n = 100
prob = 0.95
```

```
whichComponent = rbinom(n, size = 1, prob = prob)
head(whichComponent, 20)

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

In the first 20 draws, the 6th and 11th are contaminated. We can create our mixture sample from `whichComponent` with

```
means = c(20, 0)
sds = c(3, 1)
x = sapply((1 + whichComponent),
            function(i) rnorm(1, means[i], sds[i]))
```

The calculation `n - sum(whichComponent)` indicates that 6 of the 100 values contaminate the sample, and the histogram,

```
hist(x, breaks = 30, main = "")
```

shows reason histogram looks reasonable. considering that

Encapsulate the Code in a Function

To wrap this code into a function, we first identify the inputs. In the code above, we parameterized the sample size (`n`), the chance the observation is from the contamination distribution (`prob`), and the means and SDs of the two normal distribution. It seems appropriate for all of these arguments to be required so our function signature is

```
rnormalmixture = function(n, p, mus, sds)
```

The code that we have written already can go straight into the body of the function without modification, e.g.,

```
rnormalmixture =
function(n, p, means, sds)
{
  whichComponent = 1 + rbinom(n, size = 1, prob = p)
  sapply(whichComponent,
        function(i) rnorm(1, means[i], sds[i]))
}
```

Actually, we modified `whichComponent` slightly by adding 1 to each element so that it can be used to index into `means` and `sds`.

Vectorization and Efficiency

One of the first things to consider when we write functions in *R* is whether or not we are taking advantage of the vectorized computations. Our initial attempt at `rnormalmixture()` generates each value one at a time in the `sapply()`. We can improve our code by generating all of the observations from each component distribution in one call to `rnorm()`. We simply need to know how many values to generate. Rather than generate `n` individual values to indicate which distribution to draw from, we can simply generate the number needed from each distribution, we can do this with `rbinom(1, size = n, prob)`. We update the function to use this concept

```
rnormalmixture.faster =
function(n, p, means, sds)
{
```

```

count = rbinom(1, size = n, p)
sample(c(rnorm(count, means[1], sds[1]),
         rnorm(n - count, means[2], sds[2])))
}

```

We can compare the efficiency of these two approaches with

```

set.seed(a.seed)
system.time(rnormalmixture(1000000, prob, means, sds))

user   system elapsed
5.052   0.344   5.589

set.seed(a.seed)
system.time(rnormalmixture.faster(1000000, 1 - prob, means, sds))

user   system elapsed
0.147   0.005   0.152

```

When we generate 1 million random values, the revised vectorized approach is about 70 times faster.

Test Cases

We tested our first function on a simple problem that generates 100 observations with 5% contamination, and we have not tested our revised function at all. It's a good idea to create a battery of tests that we can apply to check that our function works as expected. As we continue to revise and update our function we rerun these tests to confirm that the changes have not broken our code. It's a good idea to include in our test, cases that are at the boundary of possible values. For example, we can consider the case where the SD of the contamination is 0 so all contaminated values are the same, and we can consider the case where the mixture has percent is 0 so there are no values from the contaminant.

Let's try these test cases,

```

rnormalmixture.faster(6, 0.75, means, sds = c(0, 1))
[1] 20.000 -2.160 -1.077 20.000  0.160 -0.897

rnormalmixture.faster(6, 0, means, sds)
[1] -0.4941 -0.6827 -0.8346 -1.7186 -0.1701 -1.2826

rnormalmixture.faster(6, 1, means, sds)
[1] 17.8 15.6 25.8 16.9 19.3 15.5

```

It appears that our code is working properly

Adding Debugging Features

Another approach to help us debug our code is to add features to the function that provide additional information. For example, we can add names to the return vector that indicates which component the value is from. With our example, it is clear when we see a number over 15 that it is from the contaminant, but in other cases it may not be as clear. Another example is to allow the caller to specify the counts for each component. In order to do this, we need to raise creation of count into the function's signature. This way the caller can override it. Our revised function appears below:

```
rnormalmixture.faster =
function(n, p, means, sds,
        count = rbinom(1, size = n, p),
        addComponentNames = FALSE)
{
  ans = c(rnorm(count, means[1], sds[1]),
          rnorm(n - count, means[2], sds[2]))
  if(addComponentNames)
    names(ans) = rep(1:2, c(count, n-count))
  sample(ans)
}
```

Notice that the parameter `count` depends on other parameters in the function definition. With lazy evaluation, the call to `rbinom()` is evaluated when `count` is first used in the body of the function. We can use the test cases to confirm that our code continues to work as expected. Additionally, we can develop new test cases for checking if the features we have added work properly.

USE PKG FOR TESTS?

Design

We are now ready to design the simulation study. The estimator of interest is the Winsorized mean, and we want to compare it to the plain mean and possibly other estimators that are designed to reduce the impact of outliers. For simplicity, we include only the trimmed mean in the comparison group. Both the Winsorized and trim mean have a `trim` parameter to specify the amount of trimming. We choose these to be:

```
trim.amt = c(0, 0.02, 0.05, 0.08, 0.1)
```

Other parameters we need to choose include the amount of contamination, the location and spread of the contamination, the sample size, and the number of replications. The impact of the contamination can be observed with the original mean and SD (20 and 3, respectively) so we limit our contaminating distribution to this. We also keep the contamination to 5% and leave it to the exercises to continue the simulation study with other levels of contamination. Finally, we set the sample size and replication to

```
sample.size = c(20, 100, 200, 500)
reps = 1000
```

Next, we create a function to carry out one simulation for each of these values.

Simulation

The function `simStudy()` below carries out a simulation for one sample size and all of the trim amounts. We write this function so that it uses the same simulated mixture random variables for all of estimators. This reduces the variability in comparing the estimators and is more efficient. The function is defined as

```
simStudy = function(size1, trim, rep = 1000,
                     prob = 0.05, means = c(20, 0), sds = c(3, 1))
{
  # prob, means, and sds specify the mixture distribution
  # size1 is the sample size, trimAmt is a vector of trim values

  # generate the rep sets of samples
```

```

x = matrix(rnormalmixture.faster(n = size1 * rep,
                                 p = prob, means = means, sds = sds),
            nrow = rep)

# apply the mean-type function to each sample
require(psych)
meanPlain = apply(x, 1, mean)
meanTrim = sapply(trim, function(trim1)
  apply(x, 1, mean, trim = trim1))
meanW = sapply(trim, function(trim1)
  apply(x, 1, winsor.mean, trim = trim1))

return(cbind(matrix(meanPlain, ncol = 1), meanTrim, meanW))
}

```

BEFORE APPLY WITH FULL AMOUNTS – RUN A TEST CASE . We apply the function over the sample sizes with

```

set.seed(a.seed)
simResults = lapply(sample.size, simStudy, trim = trim.amt)

```

Now `simResults` is a list of 4 matrices, one for each sample size. Each matrix has 1000 rows, which correspond to the replications, and 11 columns, where the first column corresponds to the regular mean, the next 10 to the trimmed and Winsorized means (each trimmed at one of the 5 levels).

Analysis of Simulation Results

Now that we have conducted our simulation study, we have 36,000 means to compare. A graphical comparison often works well for it can be difficult to assimilate tables of summary statistics. For each estimator, we can make a violin plot of the 1000 averages. We can arrange these violin plots side-by-side to facilitate comparison between the trimmed mean and Winsorized mean at different levels of trimming. We make this plot with

```

par(mfrow = c(3, 1), mar = c(3, 3, 1, 1))

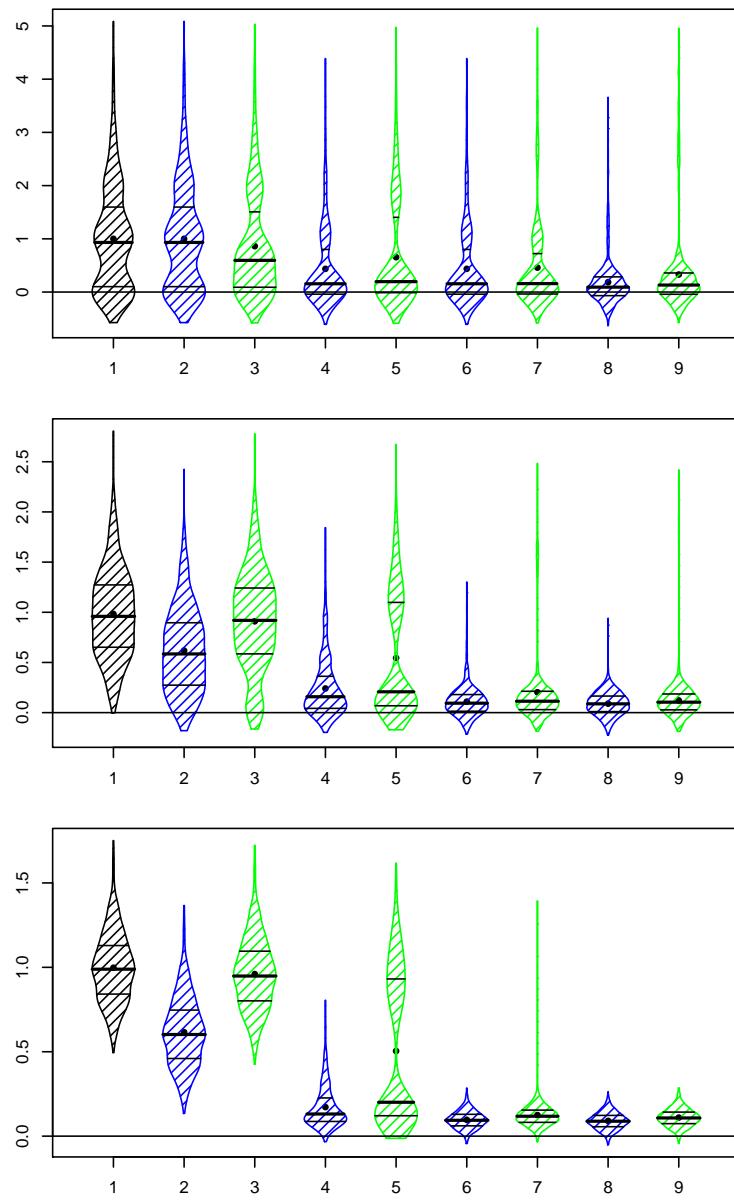
for(i in c(1, 2, 4)) {
  violinBy(x = simResults[[i]][,
    c(1, 3, 8, 4, 9, 5, 10, 6, 11)],
    main = "", col = c("black", rep(c("blue", "green"), 4)))
  abline(h = 0)
}

```

WHAT SEE? ADD RMSE COMPARISON WHAT SEE?

Generalize

We conclude this section by taking stock of our `rnormalmixture.faster()` function. It was very helpful in generating data for the contaminated model that is commonly used as in simulation studies in robust statistics. However, we can see that generating data from a mixture model where we mix more than 2 normals has the potential for being more useful. We return to our function and generalize it to any number of normal distributions. For example, we may want to generate values from a mixture of 3 normals, such as described by the following matrix:

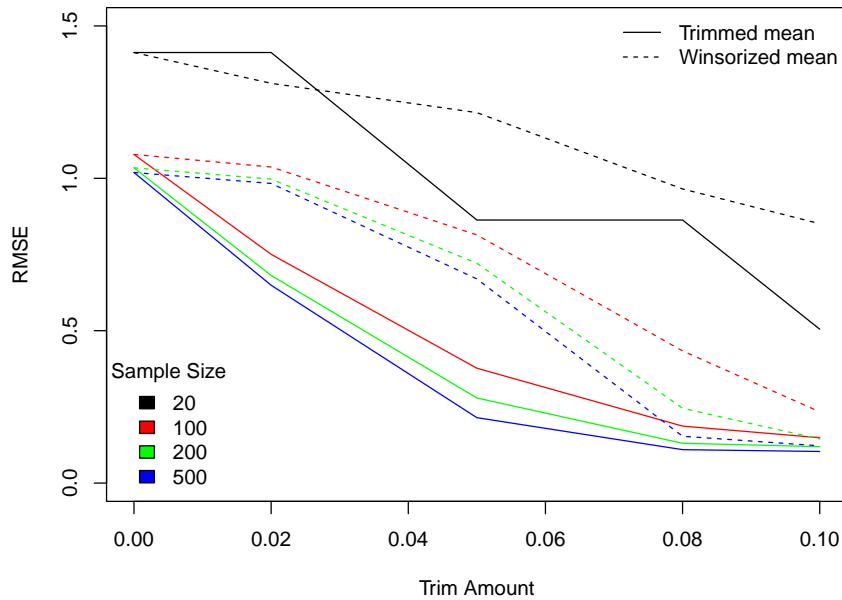
Figure 7.10: Violin Plots for Estimators. xx

```

mix = matrix(c(0, 1, 0.7, 10, 2, 0.1, -5, 1, 0.2),
             nrow = 3, ncol = 3, byrow = TRUE,
             dimnames = list(seq(1:3), c("Mean", "SD", "Prob")))
mix

```

	Mean	SD	Prob
1	0	1	0.7
2	10	2	0.1
3	-5	1	0.2

Figure 7.11: RMSE for Estimators. *xx*

Do we need to change the signature of our function? Recall it is

```
rnormalmixture.faster =
function(n, p, means, sds,
  count = rbinom(1, size = n, p),
  addComponentNames = FALSE)
```

We can keep the argument names as they are, but expect a vector for *p* that is the same length as *means* and *sds*. We do need to update the default value for *count* as we need to generate counts for more than 2 mixture distributions. The multinomial random number generator does this, ie., our new function signature is

```
rnormalmixture.faster =
function(n, p, means, sds,
  count = rmultinom(1, size = n, p),
  addComponentNames = FALSE)
```

The code in our original function depended on there being only 2 mixtures so we will need to rewrite our code to make it accommodate more than 2 mixture distributions. This often happens when we want to generalize our code. Our revised function is

```
normalmixture.general =
function(n, p, means, sds,
  counts = rmultinom(1, size = n, p),
  addComponentNames = FALSE)
{
  numMix = length(means)
  ans = unlist(sapply(1:numMix,
```

```

        function(i) {
            rnorm(counts[i], means[i], sds[i])
        })
    if(addComponentNames)
        names(ans) = rep(rownames(counts), counts)
    return(sample(ans))
}

```

We want to test our new, more general, RNG for normal mixtures. We can use `mix` with

```

set.seed(a.seed)
x = normalmixture.general(1000, p = mix[, 3], means = mix[, 1],
                           sds = mix[, 2], addComponentNames = TRUE)

```

The histogram in [?] xxxx. We can also check our code against the boundary cases described earlier, i.e., when we supply probabilities that are 0 and/or SDs that are 0 to the function. When we do, we find the return values are as expected.

Developing a Visualization for Debugging

Now that the mixtures can be more complex, we might want to consider other approaches to check our code. One possibility is to create a function that produces the density values for the mixture distribution, much as `dnorm()` provides the density values for the normal. The function `dnormalmixture()` below creates a weighted version of the `dnorm()` values for the desired density, e.g.,

```

dnormalmixture =
function(x, p, means, sds)
{
    rowSums(sapply(1:length(p),
                  function(i)
                      dnorm(x, means[i], sds[i]) * p[i]))
}

```

We can superpose the density curve on the histogram of `x` with

```

hist(x, prob = TRUE, ylim = c(0, 0.3), breaks = 40, main = "")
curve(dnormalmixture(x, p = mix[, 3], means = mix[, 1], sds = mix[, 2]),
      from = -10, to = 20, lwd = 2, add = TRUE)

```

We see in Figure 7.12 that the histogram follows the density curve quite closely.

Backward Compatibility of Code

One final remark: we generalized our RNG for a mixture of 2 normals to a mixture of any number of normals. In the original problem, we only needed to supply one probability in `p` because the chance of a random sample from the other, contaminating distribution is simply $1 - p$. In the revised function, we expect all of the probabilities to be provided so if there are only 2 distributions being mixed then we need to provide `p` as a vector of length 2. If we (or others) have code that uses the earlier version of `rnormalmixture()`, then we probably want to modify our general function so that it checks for this situation and modifies `p` to conform to the expected input. This way, the code that calls `rnormalmixture()` still works with the updated function and the transition from the old to the new version of the function is seamless. We discuss this issue more in the Guided Practice problems.

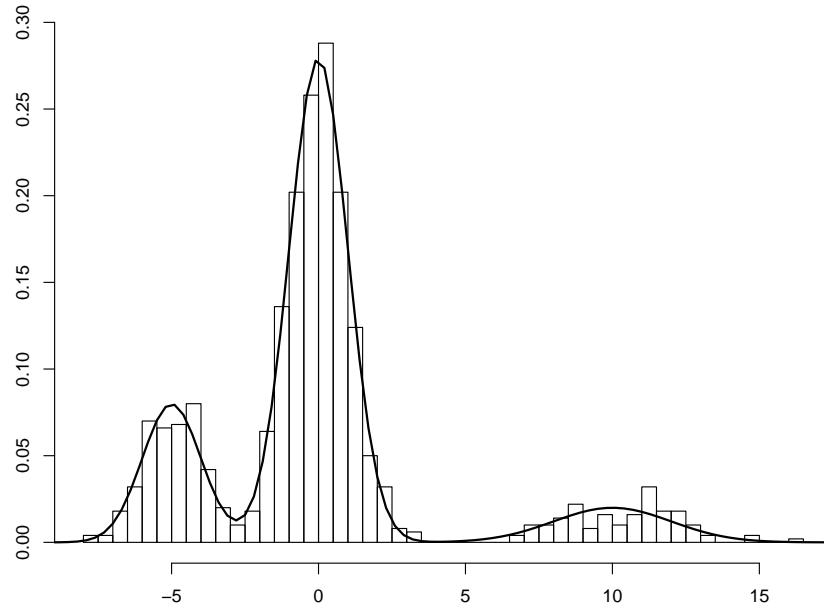


Figure 7.12: Histogram and Density for a Normal Mixture. xx

7.7 Simulating the Biham-Middleton-Levine Model for Traffic

Before we start coding or even identifying tasks to code, let's sketch a simple grid and move through a few steps of the process in order to make sure that we understand what's involved. We begin with cars on the grid at time 0 as in (a) of [?]. At time $t = 1$, all red cars move one cell to the right or to the first column if they are already at the right edge of the grid, unless that cell is currently occupied. At time $t = 2$, the blue cars move one cell up (or loop around to the bottom edge of the grid) if that cell is unoccupied. This continues with red cars moving at odd-numbered times and blue cars at even number times.

How do the cars get on the grid? In other words, how do we create the initial state of the system? We randomly place n_R and n_B cars on the grid. We have to make certain that each car is in its own cell and that 2 cars don't occupy the same cell. This is the only random part of the dynamic system. After the cars are randomly located in the initial grid, their moves are entirely deterministic.

Suppose we only had 2 cars on a 10-by-10 grid. Both are likely to be able to move at each time step as they probably do not occupy contiguous cells. Here, the cars move freely and their velocity is essentially 1 unit of distance for each unit of time. On the other hand, with 100 cars, no cell is vacant and no car can move. We have a complete traffic jam and the velocity is 0. Between these two extremes in the proportion of occupied cells, we can observe different behaviors ranging from global free-flowing traffic to localized deadlock to global deadlock. The behavior of this model depends critically on the number of cars on the grid.

Another clarifying point is that all cars of the same color move simultaneously. In other words, when it's the red cars turn to move, we determine which target cells are vacant, and then move only those cars that can move to one of these vacant cells. This approach does

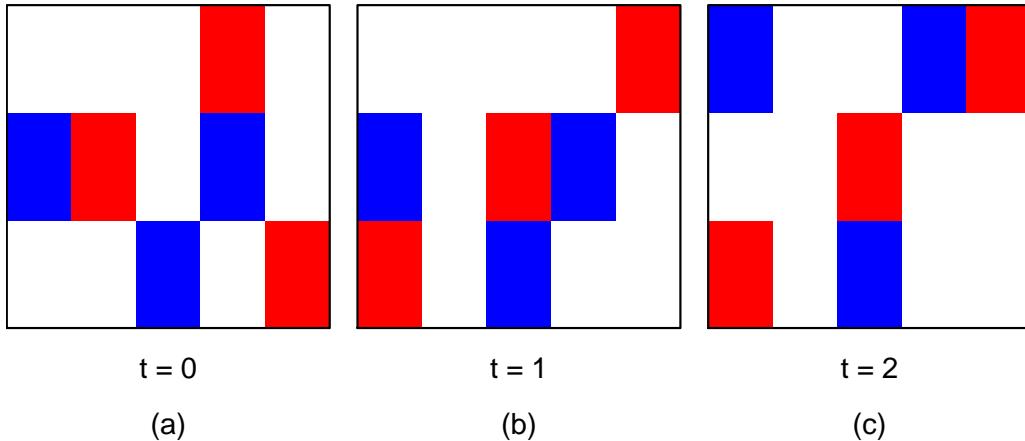


Figure 7.13: Movement on a Sample Grid. (a) shows the initial state of a 3-by-5 grid containing 3 red and 3 blue cars. At time $t = 1$, the red cars move horizontally. The red car on the bottom row “wraps around” to the first column on the same row. At time $t = 2$, the blue cars move up within the same column. The blue car in cell (1,3) is blocked by the red car above it that moved to that cell in at time $t = 1$. Accordingly, we obtain a sequence of grids indexed by time.

not allow a car to move into an adjacent cell if it is occupied by a car that can move to a vacant cell in that time period.

We now have a complete description of the BML model and can start to write R code to implement and simulate it. Let’s think about the different steps we have and also what inputs we need at each step.

1. We need to create the initial configuration of cars on the grid for time $t = 0$.
2. We want to view the grid at the start and also at different time steps in the process as it evolves. Visualizing the state of the grid helps us to understand the process, but is also essential to help debugging the code.
3. At a time t , we need to move the cars. At odd numbered time steps, we move the red cars, and at even numbered time steps, we move the blue cars.
4. To run the process, we need to iterate over a sequence of time steps.

As we develop functions for each of these steps, we also try to make them flexible, extensible, and efficient. In addition to visualizing the grid as it evolves, we also want to compute summary statistics from the grid such as the proportion of cars that move at each time step, and hence average velocity.

7.7.1 The Initial Traffic Configuration

To place the cars on the grid, we need to know the dimensions of the grid and the number of red and blue cars. We can think of the grid as being a square and use just a single dimension to describe it. However, this is unnecessarily restrictive. It costs us very little to allow the user of our function `createGrid()` to specify different lengths for the width and height of the grid. We can allow the caller to specify 1 length and use that for both dimensions,

but still have the option of specifying the 2 dimensions separately. Being able to vary the dimensions allows us to explore how the behavior of the process changes as we use, say, relatively prime dimensions. This does indeed turn out to be important.

Similar to the grid dimension, we can also specify the number of red and blue cars separately or specify a single value to be used for each. Rather than specifying the number of cars, it can be convenient to specify the proportion of the cells that should contain a car. As we change the dimensions, this makes it easy to keep the same density of occupied cells. Given this proportion and the dimensions of the grid, we can compute the total number of cars. We can divide this in two to get the number of red and blue cars. Accordingly, we have 3 different ways to specify the number of each type of car: a single number used for both types, a vector of length 2 with a number for each car color, and a proportion between 0 and 1 used to compute the proportion of cells occupied by a car of either type.

Let's put these different forms of inputs for the caller together to define the skeleton of a function to create our initial configuration:

```
createGrid =
function(dims = c(100, 100), numCars = .3)
{
  if(length(dims) == 1)
    dims = rep(dims, 2)  # a square grid

  if(length(numCars) == 1 && numCars < 1)
    numCars = rep(prod(dims) * numCars/2, 2)

  ...
}
```

This function ensures that `dims` and `numCars` have the correct length and interpretation.

We can now generate the actual locations of the cars. How should we do this? We know we need `numCars[1]` red cars and `numCars[2]` blue cars. For each car, we need its row and column numbers. We have to ensure that there is no existing car already at that location. We can generate the location one car at a time. We generate a possible location (i, j) and then check to see if the cell is already occupied. If the cell is occupied, we generate another possible location and iterate until we place that car. As we place more and more cars, fewer locations are available for the next cars. As a result, we spend more time/iterations finding an available location.

Rather than using a loop to place each car, let's think of different approaches that use a vectorized operation. Firstly, suppose we have a 2-column matrix of the locations (row and column indices) of all the cars without knowing the color of each car. We can randomly assign the red and blue labels to those locations/rows. This gives the same result probabilistically as if we allocate the colors before positioning them. We can generate the colors/labels for the $n_R + n_B$ cars with

```
carColors = sample( rep(c("red", "blue"), numCars) )
```

Note that we use `rep()` to create a vector with the specified number of red and blue elements and then we generate a permutation of this vector. This ensures we end up with the specified number of red and blue values. We can check `carColors` contains the correct number for each color with

```
table(carColors)
```

Once we know the colors of the cells with cars, we need to generate their locations. We might consider sampling the row indices and then the column indices and combine them together into a 2-column matrix, e.g.

```
N = sum(numCars)
rows = sample(1:dims[1], numCars, replace = TRUE)
cols = sample(1:dims[2], numCars, replace = TRUE)
```

Here `sum(numCars)` is the total number of cars to be placed. Unfortunately, this approach can end up with conflicts with 2 cars at the same location.

Instead, we can label each cell uniquely by its row and column numbers. We can then sample from these unique cell identifiers and decompose each identifier for a cell into its row and column. We are guaranteed no conflicts because we sample without replacement from the available cells. This allows us to do the sampling in one operation. We create the unique identifiers with

```
ids = outer(1:dims[1], 1:dims[2], paste, sep = ",")
```

This yields character strings of the form "1,1", "1,2", ..., "m,n". We can then sample these using

```
pos = sample(ids, sum(numCars))
```

and get the row and column values with

```
tmp = strsplit(pos, ",")
rows = as.integer(sapply(tmp, '[' , 1))
cols = as.integer(sapply(tmp, '[' , 2))
```

This works well, but we can make this much simpler.

Let's consider the idea of sampling indices 1, 2, 3, Suppose we represent our grid of cells as an *R* matrix. We can use these positions to directly index into elements of the matrix. This is because a matrix is merely a vector of values with a dimension attribute that specifies the number of rows and columns. We can create our empty grid with

```
m = matrix(0, dims[1], dims[2])
```

We can then sample the indices at which the cars will be located with

```
pos = sample(1:(dims[1]*dims[2]), sum(numCars))
```

Then

```
m[pos] = 1
```

sets only the elements corresponding to our sampled cells to 1. This is much more succinct. We have glossed over one detail in subsetting the matrix by the sampled indices. A matrix stores the values in column order and not row order. In our case it doesn't matter whether we use rows or columns when indexing as the cells we sample are uniformly distributed throughout the entire collection of cells in the matrix. The order doesn't matter.

Let's create our function for the initial configuration of cars using this final approach:



```

createGrid =
function(dims = c(100, 100), numCars = .3)
{
  if(length(dims) == 1)
    dims = rep(dims, 2)

  if(length(numCars) == 1 && numCars < 1)
    numCars = rep(prod(dims) * numCars/2, 2)

  grid = matrix("", dims[1], dims[2])

  pos = sample(1:prod(dims), sum(numCars))
  grid[pos] = sample(rep(c("red", "blue"), numCars))

  grid
}

```

We set the elements with a car to the corresponding color. Note that we explicitly returned `grid` at the end of the function. Sometimes people forget to do this and leave the last expression as the assignment `grid[pos] = sample(...)`.

7.7.1.1 Testing the Grid Creation Function

Before we move on to the next steps of moving the cars, let's verify that our function gives us sensible output. This is very important. If we get this wrong, the rest of our work will also be wrong. Any testing we do on subsequent code is wasted and this can consume a lot of time and also even lead to erroneous code, results, and confusion in the programmer's mind. In short, we need a solid foundation for each next step.

We need to think of ways to test the `createGrid()` function. We can print the results, but for anything but very small grids, the output will be overwhelming. Therefore, we need some meaningful summaries. We also need to test it with different inputs, e.g., non-square grids, different numbers of red and blue cars, and different densities.

A simple test is whether the function returns what we expect

```

g = createGrid()
class(g)

```

```
[1] "matrix"
```

```
dim(g)
```

```
[1] 100 100
```

These are what we expect.

We should also check that the number of red and blue cars is the same and account for 30% of the available cells:

```
table(g)
```

	blue	red
7000	1500	1500

The first element (7000) is the count of the empty cells. The name appears blank as this corresponds to the value "" in the grid. Do we expect the numbers of red and blue cars to be identical?

Rather than simply evaluating these expressions and visually verifying the results are as we expect, we can raise an error if they are not. For example,

```
stopifnot(dim(g) != c(100, 100))
stopifnot(all(table(g) %in% c( 7000L, blue = 1500L, red = 1500L)))
```

The latter test doesn't check whether the individual elements in `table(g)` are the same as those in the vector we expect. Instead, it merely checks that all the values are in the vector we expect; the counts can be in a different order corresponding to different colors. To test they are exactly as we expect, we have to compare the corresponding elements. `table()` returns a slightly more complex object than a simple vector of counts. To test for equality of the object, we have to get the dimension, dimension names, and class to be the same, e.g.,

```
stopifnot(identical(table(g),
                     structure( c(7000L, 1500L, 1500L),
                                dim = 3L, class = "table",
                                dimnames = list(g = c("", "blue", "red")))))
```

This is a much better test, but its added complexity runs the risk of introducing errors/bugs into the test itself.

Let's use our function to create a small grid with unequal dimensions:

```
createGrid(c(3, 5), .5)
```

This gives a warning

```
Warning in grid[pos] = sample(rep(c("red", "blue"), numCars)) :
  number of items to replace is not a multiple of replacement length
```

So good thing we checked! We use

```
options(error = recover, warn = 2)
```

to establish a debugging mechanism that allows us to explore the errors and warnings when and where they occur. Setting `warn = 2` causes warnings to be treated as errors. This allows us to stop at a warning and explore the current state of the computations where that warning occurs. We trigger the problem again by re-evaluating the same expression. (Given the computations involve randomness, it is possible that the same call will not generate a warning each time. However, in this case it will.)

```
createGrid(c(3, 5), .5)
```



```
Error in grid[pos] = sample(rep(c("red", "blue"), numCars)) :
  (converted from warning) number of items to replace is not
  a multiple of replacement length
```

Enter a frame number, or 0 to exit

```

1: createGrid(c(3, 5), 0.5)
2: #13: .signalSimpleWarning("number of items to replace is not
3: withRestarts({
   .Internal(.signalCondition(simpleWarning(msg, call
4: withOneRestart(expr, restarts[[1]])
5: doWithOneRestart(return(expr), restart)

```

Selection:

When the warning occurs, R converts it to an error and calls `stop()`, which is intercepted by our `error` handler, the `recover()` function. We are presented with the stack of current function calls, i.e., the call stack. These are the items 1 through 5. The error is in the body of the function `createGrid()`. So we can enter 1 at the `Selection:` prompt. This places us in the call frame for this function call. We can then examine (and even modify) the parameters and local variables that define the state of this call.

Within the debugging browser, when we look at the value of `numCars`, we see this is

```
[1] 3.75 3.75
```

and `pos` is something like

```
[1] 1 14 7 3 15 9 12
```

What is the right hand side of the assignment? It is the value of the expression

```
sample(rep(c("red", "blue"), numCars))
```

and is something like

```
[1] "red"  "red"  "blue" "blue" "blue" "red"
```

The values are random, but the number of elements is not and is 6 rather than 7, which is the length of `pos` (since `sum(numCars)` is 7.5). That is the disparity. The problem is the fractional values in `numCars`, i.e., 3.75 and how this affects the call to `rep()`. Instead of using `rep()`, we can use `rep_len()` and ensure that we get the same number of elements as in `pos`:

```
rep_len(c("red", "blue"), length(pos))
```

But this doesn't allow us to specify a different number of cars for the red and blue cars, i.e., a vector for `numCars`. Instead, we might use

```
sample(rep(c("red", "blue"), ceiling(numCars)))[seq(along = pos)]
```

which rounds the number of cars for each type to the next largest integer and then subsets the result to have the same length as `pos`.

We can make this change and redefine the function as

```

createGrid =
function(dims = c(100, 100), numCars = .3)
{
  if(length(dims) == 1)
    dims = rep(dims, 2)

  if(length(numCars) == 1 && numCars < 1)

```

```

numCars = ceiling(rep(prod(dims) * numCars/2, 2))

grid = matrix("", dims[1], dims[2])

pos = sample(1:prod(dims), sum(numCars))
grid[pos] = sample(rep(c("red", "blue"),
  numCars)) [seq(along = pos)]

grid
}

```

Now we need to rerun our tests that passed for the previous version and continue to add new tests. It is essential we do this. We have modified and added code. As a result, there is a good chance we have introduced a bug. Having the tests from earlier in separate files that we can `source()` into *R* and raise an error (via, e.g., `stopifnot()`) if there is an unanticipated result allows us to easily recheck our code and be somewhat confident about it before embarking on the next step.

Let's create a small grid. We may want to ensure we can get the exact values again for the cells and so set the random seed:

```

set.seed(1456)
createGrid(c(3, 5), .5)

[,1]   [,2]   [,3]   [,4]   [,5]
[1,] "blue"  ""     "red"  "red"  ""
[2,] ""       "red"  ""     "blue"  ""
[3,] "blue"  "red"  ""     "blue"  ""

```

We can specify the number of red and blue cars with

```

set.seed(1234)
a.grid = createGrid(c(3, 5), c(9, 3))
a.grid

[,1]   [,2]   [,3]   [,4]   [,5]
[1,] "red"  "red"  "red"  "blue" "red"
[2,] "red"  "red"  "red"  ""     ""
[3,] "blue" "red"  "blue" ""     "red"

```

Again, we need to develop tests for corner cases, e.g., the caller asking for 0 cars of either or both types, or more cars than can fit on the grid, etc.

7.7.2 Displaying the Grid

When we display a `BMLGrid` object in the *R* console, it is shown as a matrix, as we saw in the 3-by-5 example above. There are several aspects of this that make it harder than we like to quickly comprehend. Firstly, the row and column names are distracting, e.g., the `[1,]` and `[, 1]` on the side and above the actual cell values. We can display these as `1, 2, 3, ...`, without the surrounding `[,]`. More importantly, the rows appear in increasing order on the console. Specifically, row 1 appears above row 2, which appears above row 3, and so on. This is not how we think about the grid and can make it hard to reason about the movement of an individual car in the grid. When a blue car moves “up” from row 2 to row 3, it will actually move down as displayed on the console. Accordingly, we want the rows to appear in the opposite order. We can define our own function to display a BML grid object. We might implement it as

```
print.BMLGrid =
function(x, ...)
  print(structure(x[nrow(x):1,],
    dimnames = list(nrow(x):1, 1:ncol(x))))
```

This creates a new `matrix` object in the appropriate manner so that the rows appear in the “correct” order and the row and column names are set appropriately. It then uses R’s regular `print()` function to display this new representation of the original grid.

With this new `print.BMLGrid()` function, our grid `a.grid` will appear as

```
1      2      3      4      5
3 "blue" "red" "blue"   ""    "red"
2 "red"  "red" "red"   ""    ""
1 "red"  "red" "red" "blue" "red"
```

In addition to printing the grid on the console, we can visualize it using R’s graphics capabilities. This is useful so we can see that the locations of the cars and colors appear random and potentially identify any anomalies in our code. We should write a function to do this so that we don’t have to remember the details. We can draw the cells as colored circles on a scatter plot. However, it seems to make more sense to display them as rectangles that occupy the grid’s cell. We can use the `rect()` function to draw the rectangles. We need to create 4 vectors specifying the x and y locations for the two opposite corners of each cell, i.e., 4 n-length vectors where n is the total number of cars. We also have to create the initial coordinate system for drawing the rectangles by creating a new plot. This is quite simple but involves several steps. Instead, we let’s think about whether there is an existing high-level function in R or some R package that does what we want, or close to it, so that we can adapt it. Reusing functions is a good thing to do as they save us programming time but also are more likely to be correct and full-featured.

We can try to use the `image()` function to render the matrix for us. To use `image()`, we have to convert the string values in the matrix to numbers. We can do this by mapping the values "", "red", and "blue" to a set of numbers, say, 1, 2, and 3, respectively. We can do this with the `match()` function via

```
z = matrix(match(a.grid, c("", "red", "blue")),
           nrow(a.grid), ncol(a.grid))
```

The result corresponding to our grid `m` we created earlier is

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

We can then call the `image()` function as `image(z)`. This display in Figure 7.14 is somewhat difficult to interpret. Based on our matching, the colors should be white for 1, red for 2, and blue for 3. Therefore, our plot command is

```
image(z, col = c("white", "red", "blue"))
```

We have to carefully verify that the cells in the matrix correspond to those in the image. In fact, they do not. The problem is that `image()` essentially rotates the matrix by 90 degrees and displays that. Therefore, the rows and columns are exchanged in the plot. In order to obtain the same display as we have via `print.BMLGrid()` and as we would expect, we have to transpose the matrix before passing it to `image()`. We can use

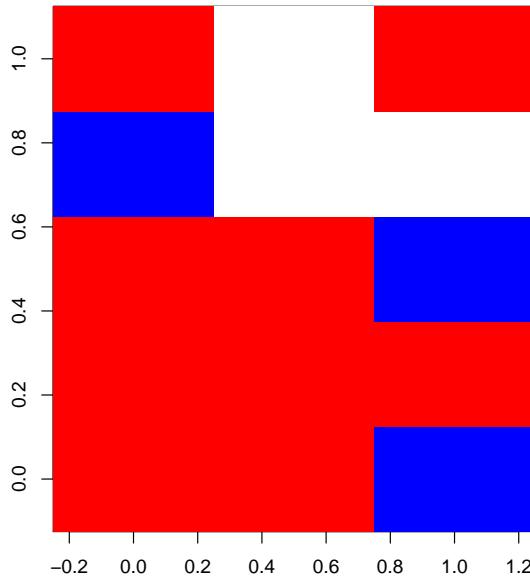


Figure 7.14: Simple Plot from `image()`. Using a simple call to `image()` to display the grid uses the wrong colors and also does not display the cells in the order we expect or want.

```
image(t(z), col = c("white", "red", "blue"))
```

To view a grid, we don't want to have to remember and perform these calculations each time. We should write a function to plot it for us that uses and encapsulates these computations. Let's call our function `plot.BMLGrid()` and define it as

```
plot.BMLGrid =
function(x, ...)
{
  z = matrix(match(x, c("", "red", "blue")), nrow(x), ncol(x))
  image(t(z), col = c("white", "red", "blue"),
        axes = FALSE, xlab = "", ylab = "", ...)
  box()
}
```

We remove the horizontal and vertical axes as they have no meaning in this context. Alternatively, we can add the row and column numbers for the grid. Note also that we changed our code to refer to `x`, the function's parameter, rather than `g`, which we had in our interactive expression we used to experiment with the `image()` function. This style of interactively refining a command and then copying it to the body of a function can often lead to bugs due to referring to variables not defined in the new function. Our function still works because `g` is available in the work space, but it ignores any grid we explicitly pass to it! It is good practice to use `codetools:::findGlobals(plot.BMLGrid, FALSE)` to check for and identify any non-local variables.

Importantly, our function passes any arguments for `image()` that we don't use directly in our function to the call to `image()` via the `...` mechanism. This allows the caller to customize the `image()` function with additional inputs. For example, she can specify a title for the plot with

```
plot.BMLGrid(a.grid, main = "A title", sub = "A sub title")
```

This is good practice to make our function more flexible for the callers with little additional effort.

Let's look at a larger grid:

```
g100by100 = createGrid(c(100, 100), .5)
plot.BMLGrid(g100by100)
```

We show this in the first panel of Figure 7.15.

We can also use many more red than blue cars to see if this characteristic appears in the display. We create a 10,000-cell display with half of the cells being red and 500 being blue. The remaining 4,500 cells are white/empty:

```
g100by100 = createGrid(c(100, 100), c(5e3, 500))
plot.BMLGrid(g100by100)
```

We can see this in the second panel of Figure 7.15.

Let's also create our own grid with no random values, but where we explicitly place the cars deterministically:

```
m = matrix("", 3, 4)
m[3, 1] = "blue"
m[1, 4] = "red"

m[row(m) == col(m) - 1] = "red"
m[col(m) == row(m) - 1] = "blue"
```

The final two expressions put red cars along the upper-off-diagonal and blue cars along the lower-off-diagonal. This is shown in the third panel of Figure 7.15.

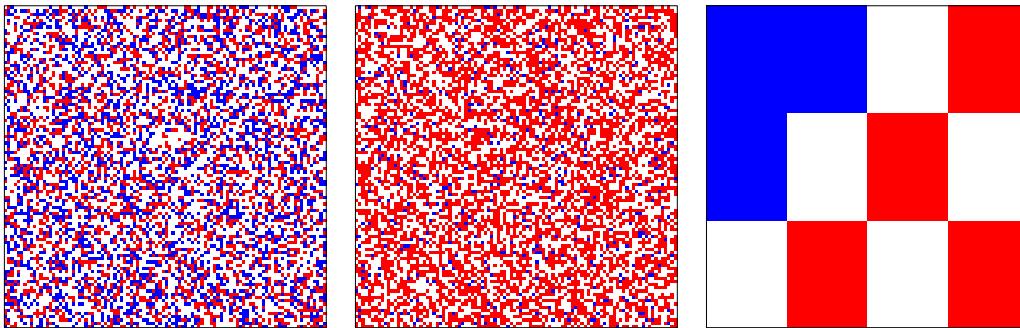


Figure 7.15: Sample Grid Displays. *The first panel shows a 100-by-100 grid with 50% of the cells occupied equally with red and blue cars. The second panel has the same dimensions but there are 5000 red cars and only 500 blue cars. The 3rd panel shows a small 3-by-4 grid where we placed the cars manually.*

Before we move on to the next topic, we note that we have now repeated our colors (white, red, and blue) in 3 different places: in `createGrid()` and in the calls to `match()` and `image()`. We are violating the DRY (Don't Repeat Yourself) principle. We have to ensure they are all the same and in the same order. It also makes it harder for us to change if we want to use different colors, e.g., for issues with color-blindness, displaying in different

media such as an overhead projector versus a monitor versus printing. Instead, we should define the color names as a vector and reuse this in each of these 3 locations. We can use a global variable and make this the default value for a parameter in each of these functions. In general, global variables are “bad;” however, here we are using this essentially as a global *constant*/immutable vector in a centralized location.

7.7.3 Moving the Cars

After all this work, we have now created our grid and functions to visualize it corresponding to the first 2 of our 4 steps (see [?]). The next task is to move the cars for a given time step. We have to move either the red cars right, or the blue cars up. We can write two separate functions, each handling the different colored cars. However, they are likely to share common operations and we would end up repeating code. To do this we would probably cut-and-paste code from one function to the other. If there is an error in the original function, it will be present in the second. We often correct it in one but not the other. Additionally, if we improve the code in one place, we have to make the same changes in the other place. Furthermore, it makes reading, understanding, and maintaining the code harder. We have lost the explicit connection between the two that is clear when two functions call a shared function, but not when we have the code repeated. The general idea is the DRY principle — Don’t Repeat Yourself. It is such an important concept in programming, we’ll say it again — Don’t Repeat Yourself!

Ideally, and ultimately, we would like to have a single function that we can call to move either set of cars, which color depending on the caller. Certainly, we would like to have a single function that identifies the common abstractions across the 2 colors, and perhaps specializes the actual motion for the different directions in other functions. It may be prudent not to be too ambitious at the very beginning. We may want to start by writing two separate functions to get things working. Then we could examine these and identify their common parts and combine them into one more general function. This is a good approach as we are striving for the general version ultimately, but making things simpler initially. Starting with the more general, abstract version may slow us down and be too difficult without having a working version for one type of car.

The most obvious way to move the cars is to process each car separately, determine whether the cell to which it would move is currently vacant, and if so update the location of that car. When a car moves, we have to clear the cell it currently occupies and set the color of the cell to which it moves. We can do all of this within a loop to process all of the cars of a given color. As usual, this is not ideal in *R* as it can be slow. Instead, we’d like a vectorized approach. However, let’s implement this loop approach here as a) we want a version that we know is correct and which we can use to check a more ambitious, vectorized version, and b) we’ll revisit this in another context later on in the chapter (when we implement a fast version in *C*). This approach treats the cars sequentially, rather than simultaneously. As a result, a car may not be able to move in this time step, but would if we changed the order in which we process the cars. This is fine for our implementation.

While the loop will be relatively easy to write, we immediately run into a problem with how we have represented our grid. We have the entire matrix but we do not know the location of the red or the blue cars. When we created the grid, we knew their locations. However, we then put this information into the matrix and discarded it. For each time step, we have to extract the information from the matrix about where the cars are currently located. We can do this with the `row()` and `col()` functions. We can get the locations of all the cars with

```
i = row(grid)[grid != ""]
```

```
j = col(grid)[grid != ""]
pos = cbind(i, j)
```

We can use `pos` to index our grid matrix to get the color associated with each of the cars:

```
colors = grid[pos]
```

Note how we are using a (2-column) matrix to subset a matrix. This takes a little time to get used to. Take a moment to experiment at the *R* console with matrix subsetting using some small, simple matrices or these grids of cars.

We can combine the row and column information into a matrix representing all of the car locations and using their color as the row names with

```
cars = structure(pos, dimnames = list(colors, c("i", "j")))
```

We now have the locations and we can loop over the relevant subset (blue or red) to move those cars. Why do we use a matrix rather than a 3 column data frame with color as one of the columns? There are good reasons for using data frames for data analysis. They do incur some overhead, but avoid others. In our situation, we are working with a grid that is a matrix, and it is natural to use a matrix for the car locations.

Our loop to move the cars of one color, say blue, can be implemented something like the following

```
w = which(rownames(cars) == "blue")
for(idx in w) {
    curPos = cars[idx, ]
    nextPos = c(if(cars[idx, 1] == nrow(grid))
                1L
               else
                cars[idx, 1] + 1L,
               cars[idx, 2])

    # check if nextPos is empty
    if(grid[nextPos[1], nextPos[2]] == "") {
        grid[nextPos[1], nextPos[2]] = "blue"
        grid[curPos[1], curPos[2]] = ""
    }
}
```

The code is reasonably straightforward. We determine the indices for the blue cars and loop over these. For each blue car index, we extract its current position and compute its would-be next position. We adjust for reaching the top edge of the grid so that we wrap around to the lowest row on the grid, if this occurred. Then we test if the would-be position is available/empty and if so, update the current contents of the grid.

To move the red cars, we would have very similar code. We'd replace “blue” with “red” and also how we compute the `nextPos`. This allows us to see how to write a single function, `moveCars()`, which can move either set of cars — red or blue. Our `moveCars()` function takes the current grid and returns the updated grid. Note that these 2 grids will be separate copies, not a modification of a shared grid object. This allows us to build up a sequence of grids and visualize the progress and easily compare them to validate our code. Our function also needs to know which color of car we are moving. We define it by combining the code from the different steps above and abstracting it to handle both red and blue moves.

Rather than writing one larger function that contains all of the code to carry out the initial configuration and moving from one grid layout to another, we create 2 helpers functions for this purpose. This way we separate the code for these two tasks so that we can test these functions individually and make our `moveCars()` function easier to read.

First we encapsulate the code to layout the initial grid with

```
getCarLocations =
function(grid)
{
  i = row(grid)[grid != ""]
  j = col(grid)[grid != ""]
  pos = cbind(i, j)
  colors = grid[pos]

  structure(pos, dimnames = list(colors, c("i", "j")))
}
```

Now we can test this code separately from moving the cars.

Next we create the function to calculate the next position, `nextPos`. This function needs the current position of the cars and whether we are moving horizontally or vertically. It also needs to know the dimension of the grid so that it can “wrap” cars around the edges, i.e., back to position 1 when they reach the edge of the grid. We represent the position of each car in the form `c(row, column)` where the row corresponds to the vertical position and the column corresponds to the horizontal position. We can define this function as

```
getNextPosition =
function(curPos, dims, horizontal = TRUE)
{
  if(horizontal)
    c(curPos[1],
      if(curPos[2] == dims[2])
        1L
      else
        curPos[2] + 1L)
  else
    c(if(curPos[1] == dims[1])
      1L
      else curPos[1] + 1L,
      curPos[2])
}
```

Our function `moveCars()` is

```
moveCars =
function(grid, color = "red")
{
  cars = getCarLocations(grid)

  w = which(rownames(cars) == color)
  for(idx in w) {
    curPos = cars[idx, ]
    nextPos = getNextPosition(curPos, dim(grid), color == "red")
```

```

    # check if nextPos is empty
    if(grid[ nextPos[1], nextPos[2] ] == "") {
        grid[nextPos[1], nextPos[2]] = color
        grid[curPos[1], curPos[2]] = ""
    }
}
grid
}

```

This function with its 2 helper functions, `getCarLocations()` and `getNextPosition()`, is more succinct and easier to read and follow than if we had not created these helper functions. With little mental effort and minor changes to the overall code, we have arrived at an improved solution.

We verify that when we re-factored the code to create these functions, we did not introduce references to parameters or variables that are not defined in these functions. We can use `findGlobals()` to verify this. Additionally, we need to write tests for `getCarLocations()` and `getNextPosition()` to verify they are working correctly for different inputs. Then we need to verify that the `moveCars()` function works correctly.

To do this, we use a small grid that we can inspect visually to test `getCarLocations()`. We use the grid shown below and assigned to the variable `grid`:

```

1      2      3      4      5
3 " "    " "    " "    " "    " "
2 "blue" "red" " "    "red"  "blue"
1 "red"  " "    "blue" "blue" " "

```

The output of `getCarLocations()` is

```
getCarLocations(grid)
```

```

i j
red 1 1
blue 2 1
red 2 2
blue 1 3
blue 1 4
red 2 4
blue 2 5

```

These appear to be correct.

To test the `getNextPosition()` function, we can try different locations and directions (i.e., colors).

```
getNextPosition(c(2, 3), dim = c(4, 5), horizontal = TRUE)
```

should give (2, 4), and

```
getNextPosition(c(2, 5), dim = c(4, 5), TRUE)
```

should wrap around and give (2, 1).

```
a = getNextPosition(c(2, 5), dim = c(4, 5), horizontal = FALSE)
```

returns (3, 5) as we are moving upwards. Moving that new position with

```
getNextPosition(a, dim = c(4, 5), horizontal = TRUE)
```

moves to (3, 1) due to the wrap around.

Next we need to test `moveCars()`:

```
grid1 = moveCars(grid)
```

This moves the red cars in the grid. We can compare the 2 grids (the before and after) visually with

```
grid
```

```
1      2      3      4      5
3 " "    " "    " "    " "    " "
2 "blue" "red" " "    "red"   "blue"
1 "red"  " "    "blue" "blue"  " "
```

```
grid1
```

```
1      2      3      4      5
3 " "    " "    " "    " "    " "
2 "blue" " "    "red"  "red"   "blue"
1 " "    "red" "blue" "blue"  " "
```

We can see that the red cars moved right, except the one in position (2, 4). That is blocked by a blue car in (2, 5).

Next we move the blue cars in `grid1` with `moveCars(grid1, "blue")`. We see that the new grid is

```
1      2      3      4      5
3 "blue" " "    " "    " "    "blue"
2 " "    " "    "red"  "red"   " "
1 " "    "red" "blue" "blue"  " "
```

The blue cars in columns 1, and 5 move up, and the blue cars in columns 3 and 4 are blocked.

If we spend time verifying these functions, we save ourselves time later on. Knowing these are correct allows us to use them to validate the results from other implementations, e.g., vectorized or compiled versions. Our time spent verifying these will be well spent and rewarded.

We note that in each call to `moveCars()`, we are recomputing the locations of the cars at each time step via `getCarLocations()`. However, in the previous time step we actually knew these locations so these computations are not really necessary. Instead, when we move a car on the grid, we could also update the location of that car in the associated matrix. This is duplicating information and, in a way, violating the DRY principle. However, here we are repeating data, not code. We would have to ensure that the two representations of the same information are synchronized at all times. We are also using more memory. However, it can remove the need for unnecessary, redundant computations. There is a trade-off.

7.7.4 Evaluating the Performance of the Code

Let's run our model through multiple time steps. We can write a function to do this so we can repeat this regularly for different configurations:

```
runBML =
function(grid = createGrid(...), numSteps = 100, ...)
{
  for(i in 1:numSteps) {
    grid = moveCars(grid, "red")
    grid = moveCars(grid, "blue")
  }

  grid
}
```

Technically, we iterate over twice the number of time steps as in each iteration we move both sets of cars. We allow the caller to specify the initial grid. We also use ... to allow the caller to specify inputs to `createGrid()`, which we call on their behalf if they don't provide the grid.

We can use this function to run a simple simulation using the defaults for creating the grid:

```
grid = createGrid()
grid.out = runBML(grid)
```

We can plot the initial and final configurations side-by-side with

```
par(mfrow = c(1, 2), mar = rep(1, 4), pty = 's')
plot.BMLGrid(grid, main = "Initial Grid")
plot.BMLGrid(grid.out, main = "After 100 iterations")
```

Note that we changed the margins for each plot and also used a square plotting region so that the aspect ratio of each grid is square. The display is shown in Figure 7.16.

Let's see how long it takes to run on a 100-by-100 grid, 50% filled with cars. We create this configuration once and reuse it in all our timings so that we are comparing code on the same initial state:

```
set.seed(1345)
grid100 = createGrid(c(100, 100), .5)
```

We time the code with

```
tm1 = system.time(runBML(grid100))
tm1

  user  system elapsed
 5.683   0.096   6.045
```

We create the grid separately from the timing so as to measure only the time to move the cars. The call to `runBML()` takes 6 seconds on a Macbook Pro laptop running OS X Mavericks, with a 2.6Ghz Intel Core i7 processor and 16GB of memory. Note that this will get slower as the number of cars increases, i.e., the density gets larger. This is because our loop in `moveCars()` will loop over more items.

Let's find out where the computations spend most of their time. We can use profiling for this via the `Rprof()` and `summaryRprof()` functions, e.g.,

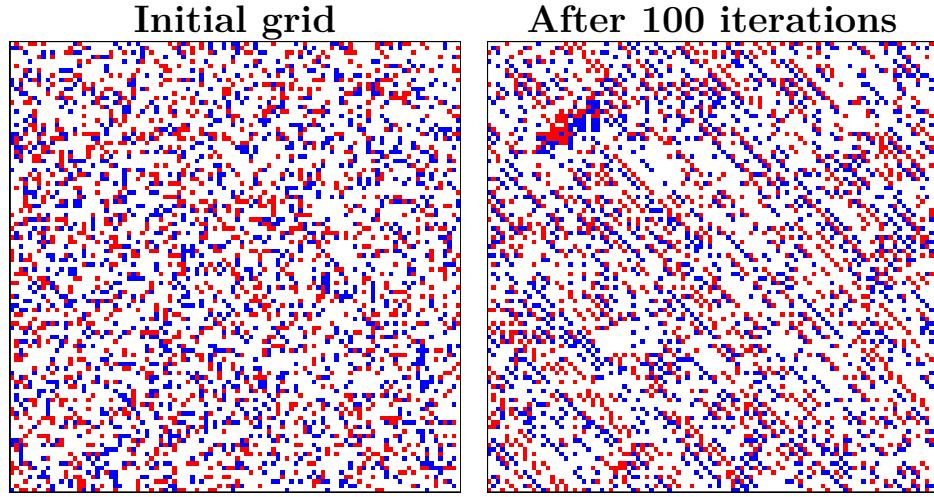


Figure 7.16: Sample Grid at Start and After 100 Iterations. *The left panel shows the initial grid with low density. The right panel shows the state of the grid after 100 iterations. Diagonal lines are already starting to emerge.*

```
Rprof("BML.prof")
grid.out = runBML(grid100)
Rprof(NULL)
head(summaryRprof("BML.prof")$by.self, 10)

      self.time self.pct total.time total.pct
"moveCars"        2.56    47.23      5.42   100.00
"getNextPosition" 1.44    26.57      2.56    47.23
"c"              0.66   12.18      0.66   12.18
"=="             0.36    6.64      0.36    6.64
"!="             0.10    1.85      0.10    1.85
"+"
```

	self.time	self.pct	total.time	total.pct
"moveCars"	2.56	47.23	5.42	100.00
"getNextPosition"	1.44	26.57	2.56	47.23
"c"	0.66	12.18	0.66	12.18
"=="	0.36	6.64	0.36	6.64
"!="	0.10	1.85	0.10	1.85
"+"	0.10	1.85	0.10	1.85
"dim"	0.08	1.48	0.08	1.48
"getCarLocations"	0.06	1.11	0.22	4.06
"cbind"	0.04	0.74	0.04	0.74
"structure"	0.02	0.37	0.02	0.37

As we expect, `moveCars()` takes a large proportion of the overall time. Also, calling `getNextPosition()` takes 26% of the time. It's surprising that `c()` takes 12% of the time.

We can make some simple improvements to `moveCars()`. We call `dim()` in each iteration of `moveCars()` in the call to `getNextPosition()`. We can move this call outside of the loop as the dimensions of the grid don't change. In general, this is a good thing to do in any computation — move invariants outside of the loop and compute them just once. Our new function definition is

```
moveCars =
function(grid, color = "red")
{
  cars = getCarLocations(grid)
```

```
w = which(names(cars) == color)
sz = dim(grid)
horiz = (color == "red")
for(idx in w) {
  curPos = cars[idx, ]
  nextPos = getNextPosition(curPos, sz, horiz)

  # check if nextPos is empty
  if(grid[nextPos[1], nextPos[2]] == "") {
    grid[nextPos[1], nextPos[2]] = color
    grid[curPos[1], curPos[2]] = ""
  }
}
grid
}
```

The small changes are highlighted.

We can compare the speed of our new function with our previous timing results in `tm1` via

```
tm2 = system.time(runBML(grid100))
tm1/tm2

user  system elapsed
1.11   1.22   1.12
```

These simple changes speed things up by about 12%.

We can go further than taking the call to `dim()` outside of the loop in `moveCars()`. If we specify the actual dimensions for the grid as a parameter for `moveCars()`, then it is computed only once in `runBML()` and passed in each call to `moveCars()`. We can use a default value for this parameter in `moveCars()` so that callers don't have to specify it. Is passing the dimension to `moveCars()` likely to significantly reduce the overall computation time?

Unfortunately, these changes are not likely to speed up the computations tremendously. This is because the calls to `dim()` account for 1% of the total time. To improve the performance significantly, we should focus on the first few functions in the output of `summaryRProf()`. How can we speed up `moveCars()`? The most obvious improvement we can make to our code is to remove the loops and attempt to vectorize the computations within `moveCars()`. This also involves vectorizing `getNextPosition()`. We cannot remove the loop in the `runBML()` function because the grid that serves as the input for iteration t is the output from the $t - 1^{st}$ iteration. The iterations depend on each other.

If we vectorize the computations, then we work with all the locations for, say, the red cars and compute all of the “next positions” in vectorized operations. Then, we find out which of these are empty and update just those in the grid. This removes the loop over the individual cars. Let's try to do this within the current structure of the code.

To develop the code, we create a simple configuration and examine the results as we step through our code. We create this simple test grid with

```
gs = createGrid(c(4, 7), .5)
print.BMLGrid(gs)

1      2      3      4      5      6      7
4 "blue"  ""    ""    ""    "red"  ""    "red"
```

```
3 ""      "blue"  ""      ""      "blue"  ""
2 "blue"  ""      "red"   "red"   ""      "blue"
1 "red"   ""      "blue"  "blue"  "red"   "red"
```

We can see this layout in the left panel of Figure 7.17.

We get the matrix of positions from `getCarLocations()` and extract the row and column values for the red cars with

```
pos = getCarLocations(gs)
red = rownames(pos) == "red"
rows = pos[red, 1]
cols = pos[red, 2]
```

The next positions can be computed for the red cars with

```
nextRows = rows
nextCols = ifelse(cols == ncol(gs), 1L, cols + 1L)
```

The row index doesn't change since the cars are moving horizontally. The `ifelse()` function is a vectorized version of an if-else statement. We can also compute the `nextCols` with

```
nextCols = cols + 1L
nextCols[nextCols > ncol(gs)] = 1L
```

Both approaches allows us to vectorize the computations for calculating `nextPos`. Next, we can determine if these cars can move in a vectorized fashion via

```
w = (gs[ cbind(nextRows, nextCols) ] == "")
```

This is similar to how we obtained the colors of the cars in `getCarLocations()` (see page 346).

We can confirm that our operations have successfully identified the red cars that can move in `gs` by comparing the following to `gs`:

```
pos[red, ]
```

```
i j
red 1 1
red 2 3
red 2 4
red 1 5
red 4 5
red 1 7
red 4 7
```

```
w
```

```
[1] TRUE FALSE TRUE TRUE TRUE FALSE FALSE
```

Indeed, we have correctly identified the red cars in `gs` that can move.

The variable `w` is a logical vector with as many elements as there are in both `nextRows` and `nextCols`, i.e., the number of red cars. We can use `w` to assign new updated values to the matrix and to set the old locations to `" "`:

```
gs[ cbind(nextRows[w], nextCols[w]) ] = "red"
gs[ cbind(rows[w], cols[w]) ] = ""
```

The updated grid is

1	2	3	4	5	6	7
4 "blue" "	" "	" "	" "	" "	"red"	"red"
3 "	"blue"	" "	" "	" "	"blue"	" "
2 "blue"	" "	"red"	" "	"red"	" "	"blue"
1 "	"red"	"blue"	"blue"	" "	"red"	"red"

Note that the car in column 7 moves to column 1 as it wraps around and the other columns are simple updates. This gives us the correct result as shown in the right panel of Figure 7.17.

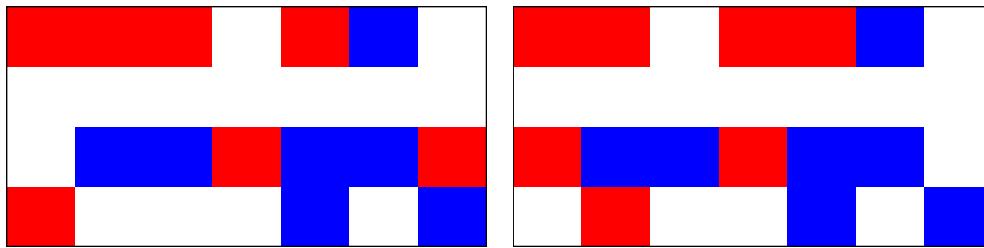


Figure 7.17: A Grid before and after Moving the Red Cars. *The left panel shows the initial 4-by-7 grid. The second panel shows the state of the grid after the red cars have moved.*

Now we have the way forward to rewrite our `moveCars()` function to use this vectorized approach:

```
moveCars =
function(grid, color = "red")
{
  cars = getCarLocations(grid)

  w = which(rownames(cars) == color)
  rows = cars[w, 1]
  cols = cars[w, 2]

  if(color == "red") {
    nextRows = rows
    nextCols = ifelse(cols == ncol(grid), 1L, cols + 1L)
  } else {
    nextRows = ifelse(rows == nrow(grid), 1L, rows + 1L)
    nextCols = cols
  }

  w = grid[ cbind(nextRows, nextCols) ] == ""
  grid[ cbind(nextRows, nextCols)[w, , drop = FALSE] ] = color
  grid[ cbind(rows, cols)[w, , drop = FALSE] ] = ""

  grid
}
```

Note the use of `drop = FALSE` when subsetting the matrices created with `cbind()`.

Again, we need to test our computations and function `moveCars()` thoroughly, including degenerate cases, e.g., where no car can move, where only one car can move, or with 1-by-1 grids. We can compare the output from this function with our previous version. These should yield the same results for arbitrary inputs. If they do agree for several different inputs, we can be confident our new function is correct. However, even if they agree, both could be incorrect. We can rename the function we defined earlier before we overwrite it, and then call them both with the same inputs. We can display the 2 grids or compare them in the *R* console or use `identical()` or `all.equal()` to test for equality of results from the two versions of the function. We'll stop here, but you should not until you are satisfied the code is correct.

Let's compare the timing for this new vectorized version of `moveCars()` to our previous timing:

```
tm_v = system.time(runBML(grid100))
tm2/tm_v

user   system elapsed
10.53    2.72   10.18
```

The new vectorized version is 10 times faster!

Let's profile our code to see if we can make more improvements:

```
Rprof("BML.prof")
grid.out = runBML(grid)
Rprof(NULL)
head(summaryRprof("BML.prof")$by.self, 10)

self.time self.pct total.time total.pct
"!="          0.18    39.13     0.18    39.13
"getCarLocations" 0.06   13.04     0.30   65.22
"cbind"        0.06   13.04     0.06   13.04
"ifelse"        0.06   13.04     0.06   13.04
"moveCars"      0.04    8.70     0.46  100.00
"=="           0.04    8.70     0.04    8.70
"structure"     0.02    4.35     0.02    4.35
```

What does this tell us? Calls to `ifelse()`, `cbind()`, and `getCarLocations()` are expensive. Moreover, it is in `getCarLocations()` that we call `!=`. We are finding that a lot of time is spent moving the results back and forth between the `grid` and the `pos` representations.

Do we need both representations? We found the grid not useful when we want to move the cars. On the other hand the grid is useful for visualizing the system. We visualize the grid only after we have run the process for many steps. If we keep the information about the cars in `pos` format, or as a simple vector of positions between 1 and `prod(dim)`, we may be able to improve our code a lot.

We no longer need `getCarLocations()`. As discovered earlier, we want to avoid computing the dimension of the grid and the number of cells in the grid with each move. Now that we are overhauling our functions, we also want to take this improvement into account. It often happens that as we program and learn more about the process, we rewrite our functions significantly. It's usually much faster for us the second time around.

We start with the function that creates the configuration. Our update does not return a matrix (`grid`) so let's give it a different name, such as `createConfig()`. With this new

structure, we no longer have the dimensions of the grid. We need the dimensions to determine when the car comes to the edge of the grid. We can add this information to the object as an attribute. Attributes store metadata about an object. Let's add the dimension of the grid as an attribute to our new object, and to avoid unnecessary computations, we also add the number of cells in the grid as an attribute. Our revised function is a pared down version of `createGrid()`, e.g.,

```
createConfig =
function(dims = c(100, 100), numCars = .3)
{
  if(length(dims) == 1)
    dims = rep(dims, 2)

  if(length(numCars) == 1 && numCars < 1)
    numCars = ceiling(rep(prod(dims) * numCars/2, 2))

  pos = sort(sample(1:prod(dims), sum(numCars)))
  names(pos) = sample(rep(c("red", "blue"), numCars))
  attr(pos, "grid") = dims
  attr(pos, "cells") = prod(dims)
  return(pos)
}
```

Now the return value is a named vector of integers, and this vector has 2 attributed, `grid` and `cells`. The length of the vector matches the number of cars in the system and the name of an element denotes the color of the car. In [?], we introduce object oriented programming and create a `grid` object to represent the traffic configuration and methods for printing and plotting the object.

Next we update `moveCars()`. We no longer need `getCarLocations()`. When we take a subset of a structure with attributes, we lose the attributes so we need to modify the function definition of `getNextPosition()` in order to supply this information. Our revised `moveCars()` appears as

```
moveCars =
function(locs, color = "red",
          dims = c(100, 100), total = prod(dims))
{
  w = which(names(locs) == color)

  newLocs = getNextPosition(locs[w], dims = attr(locs, "grid"),
                            total = attr(locs, "cells"),
                            horizontal = color == "red")

  noMove = newLocs %in% locs
  newLocs[noMove] = locs[w][noMove]
  locs[w] = newLocs

  locs
}
```

Now we tackle the revision of `getNextPosition()`. In addition to the locations of the red (or blue) cars and the logical that indicates whether the move is horizontal or vertical, we

also have arguments for the dimension of the grid and the number of cells in the grid. Our revised function uses computations similar to those in XXX, where we perform modular arithmetic to move the cars. The new function is:

```
getNextPosition =
function(curPos, dims, total, horizontal = TRUE)
{
  if (horizontal) {
    nextPos = curPos + dims[1]
    wrap = nextPos > total
    nextPos[wrap] = nextPos[wrap] - total
  } else {
    rows = (curPos - 1) %% dims[1] + 2L
    rows[rows > dims[1]] = 1L
    nextPos = dims[1] * ((curPos - 1) %/% dims[1]) + rows
  }

  nextPos
}
```

Note that we have not had to change the `runBML()` function. We implemented the functions it calls in such a way that they continue to take the same inputs, but perform their computations differently. The caller does not need to know about these details.

To time this revised version, we must first recreate the initial configuration so that it is in the format expected by `moveCars()`. This grid should match the original `grid100` because the code to generate the positions and color has not changed. We give it a different name to avoid confusion.

```
set.seed(1345)
grid100Config = createConfig(c(100, 100), .5)
tm_v2 = system.time(runBML(grid100Config))
tm_v / tm_v2

user  system elapsed
3.60    3.62   3.68
```

`tm1 / tm_v2`

```
user  system elapsed
42.1    12.0   42.0
```

This latest version of the code improves the performance over the previous vectorized approach by more than 3 fold, and it is more than 40 times faster than our original code. We have performed the timings, profiling, and performance improvements using grids with the same dimensions and densities. We also need to explore other dimensions and densities to ensure that the changes to the code improved matters for all grids. We continue to improve the speed of the simulation in [?], where we implement the algorithm for moving cars in C. We complete the simulation study at that time as well.

7.8 Summary

7.9 Exercises

Bibliography

Part III

Data Technologies

8

Text Manipulation and Regular Expressions

CONTENTS

8.1	Introduction	361
8.2	Basic Concepts in Pattern Matching	364
8.2.1	Matching Literals	364
8.2.2	Modifiers and Meta Characters	365
8.2.3	Equivalent Characters	366
8.3	Using Regular Expressions in R	370
8.3.1	Writing Our Own Literal Matcher	374
8.4	Alternatives, Grouping, and Backreferences	375
8.4.1	Grouping Characters into Sub-patterns	376
8.4.2	Alternative Patterns	376
8.4.3	Back referencing a Sub-pattern	376
8.5	Greedy Matching	377
8.5.1	Lazy Matching	378
8.6	Examples	378
8.7	Summary	383
8.7.1	Summary	384
8.7.2	Resources	384
8.8	Summary of Pattern Matching and String Manipulation Functions	384
8.9	Text Mining & Natural Language Processing	385
8.10	Guided Practice	386
8.10.1	Answers	386
8.11	Exercises	391
	Bibliography	392

8.1 Introduction

Much of the data we analyze are given to us as plain text. We input numbers in text files, download text files from Web and FTP servers, and save spreadsheets as comma-separated values in .csv files. In these cases, the data are easily interpreted by applications. However, there are many examples of more complex situations where the text must be processed to create the data values of interest. A simple example of this phenomenon is when numeric values are embedded into text, but not in a regular format, such as dates in a Web log. In this case, we must extract the elements of interest from the text content by identifying the patterns where the dates occur. A different sort of example occurs when text itself makes up the data, such as a speech, abstract, news story, email message, or tweet. Then we typically search for the presence of certain words or phrases in particular contexts or places, e.g., we might examine how often each word is used, the names of the author(s), the use of

punctuation, etc. Finally, documents and text are sometimes treated directly as data such as in search engines, databases, and so on.

Example 8-1 Extracting Variables: Weblogs

To analyze the requests made to a Web site, we extract the relevant information from each Weblog entry, such as the machine of the requester, the time and date of the request, the name of the file requested, the return status, and the number of bytes returned. Below are two lines from a Web log. (Note that each line in the Web log is broken across 5 lines of displayed text for formatting purposes.) The log is a text file where each request appears on a separate line, and although the text has a lot of structure, the information does not appear in a simple format such as in comma separated values, nor is it placed consistently in the same columns in the file.

```
169.237.46.168 -- [26/Jan/2004:10:47:58 -0800]
  "GET /stat141/Winter04 HTTP/1.1" 301 328
  "http://anson.ucdavis.edu/courses/"
  "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0;
   .NET CLR 1.1.4322)"
169.237.46.168 -- [26/Jan/2004:10:47:58 -0800]
  "GET /stat141/Winter04/ HTTP/1.1" 200 2585
  "http://anson.ucdavis.edu/courses/"
  "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0;
   .NET CLR 1.1.4322)"
```

We can extract the values of interest by searching for particular patterns. For example, the date and time are set between square brackets. ■

Example 8-2 Merging Data from Different Sources: Elections

To use demographic information to understand election results or to create a map of the results requires the census, election, and geographic information to be aligned or matched up. However, it is often the case that these data are from different sources and the fields to match on are not consistently recorded across the various sources. For example, the following are samples from geographic (top), census (middle) and election (bottom) data for counties in the United States.

```
"De Witt County", IL, 40169623, -88904690
"Lac qui Parle County", MN, 45000955, -96175301
"Lewis and Clark County", MT, 47113693, -112377040
"St John the Baptist Parish", LA, 30118238, -90501892

"St. John the Baptist Parish", "43,044", "52.6", "44.8", ...
"De Witt County", "16,798", "97.8", "0.5", ...
"Lac qui Parle County", "8,067", "98.8", "0.2", ...
"Lewis and Clark County", "55,716", "95.2", "0.2", ...

DeWitt      23        4,920     2,836      0
Lac Qui Parle    31        2,093     2,390      36
Lewis & Clark    54        16,432    12,655     386
St. John the Baptist    35        9,039    10,305     74
```

In order to merge these three sources, we can match the records by county name, which appears in each file. Unfortunately, the county names are not formatted in the same manner

across the three files. Notice that there is a period after 'St' in the census and election data but not in the geographic data; the election results differ from the other two sources in the use of & rather than "and" for Lewis and Clark County; the capitalization is not consistent across files (e.g., "Qui" rather than "qui" in Lac qui Parle County); and the use of "County" and "Parish" does not appear in the election data. If we can modify the text to make the county names consistent across the sources then our record matching will be less error prone.

Example 8-3 Deriving Variables from Text: Email

Spam filters attempt to differentiate legitimate electronic mail (email) from unsolicited bulk email (spam) by examining various characteristics of the email and using this information to predict whether or not a particular email message is spam. These characteristics can be thought of as variables that need to be derived from the email. For example, below are parts of the header from three messages. The bottom two headers are from spam while the top is not.

```
Date: Tue, 02 Jan 2007 12:17:45 -0800
From: Duncan Temple Lang <duncan@wald.ucdavis.edu>
To: Deborah Nolan <nolan@stat.Berkeley.EDU>
Subject: Re: 90 days
```

```
Date: Sat, 27 Jan 2007 16:28:48 +0800
From: remade SSE <glzmeqrxx99@embarqhsd.net>
To: depchairs03-04@uclink.berkeley.edu
Subject: [SPAM:XXXXXXXXXX]
```

```
Date: Thu, 03 Apr 2008 09:24:53 +0700
From: Faustino Britt <Faustino@sfera.umk.pl>
To: Brice Frederick <nolan@stat.Berkeley.EDU>
Subject: Fancy repl!c@ted watches
```

One potentially useful variable might be an indicator for whether or not the subject line of a message begins with "Re:". Another might be whether or not the username in the reply-to address ends with an underscore or digit. A more complex variable might be an indicator for whether or not the subject line contains a fake word, like '+rep1!c@ted' in the last email message above. To derive these variables, we need to search for particular patterns in the messages.

Example 8-4 Text Mining: State of the Union Addresses

We can mine the text of the State of the Union Addresses, looking for similarities between presidents' speeches. One approach to doing this would be to compare word frequencies across documents in the corpus of speeches. To do this, we build a word-vector for each speech that tallies the number of occurrences of each word used in the speech, e.g., there was one occurrence of the word "much", the word "debt" was used twice, and the words "nation", "national" or "nations" appeared five times in the December, 1790 inaugural speech given by George Washington (a snippet of the speech is shown below).

State of the Union Address
George Washington

December 8, 1790

Fellow-Citizens of the Senate and House of Representatives:

In meeting you again I feel much satisfaction in being able to repeat my congratulations on the favorable prospects which continue to distinguish our public affairs. The abundant fruits of another year have blessed our country with plenty and with the means of a flourishing commerce.

With word-vectors we can look for similarities between the distribution of words in the speeches. To create a word-vector, it is common to first stem words (e.g., reduce “national” and “nations” to “nation”) and possibly to remove stop words such as “and”, “the”, and “of”. We also typically remove punctuation. In other words, we need to process the text. Regular expressions can assist us. ■

Examples of Uses of Regular Expressions on Text Data

EXTRACT values that appear in non-standard formats.

CLEAN and **TRANSFORM** text into a uniform format and resolve inconsistencies.

CREATE variables from information found in text.

MINE text by treating documents directly as data.

8.2 Basic Concepts in Pattern Matching

Regular expressions provide the basic building blocks for specifying patterns that are to be matched in a piece of text. An expression allows us to match literal strings, a character from a particular set of characters or its complement (character sets), and one sub-pattern or another (alternation). We can also match by position such as at the beginning or end of a line, and we can create sub-patterns from individual characters by specifying the number of times the characters should be matched (quantifiers). Regular expressions offer a flexible way to describe patterns of characters to search for in a string.

8.2.1 Matching Literals

The core idea behind pattern matching is to search the target string one character at a time from left to right. When a character in the target string matches the first literal in the pattern, then we check to see if the next character matches the second literal in the pattern, and so on. If we do not find a complete match, then we resume checking for the first literal where we left off, i.e., the character in the string that is one past where we found the first literal in our pattern. The following simple example demonstrates the process.

Example 8-5 A Simple Literal Search

Suppose we want to search for the pattern ‘cat’ in the target string ‘The cad hid his coat. Scat!’. The pattern ‘cat’ indicates that we seek the literal c immediately

followed by the literal a, immediately followed by the literal t, anywhere in the target string. The regular expression matching engine begins looking for our pattern at the first character in the target string, i.e., at the T. Since this character does not match c, the first literal in our pattern, we move to the second character in the target and check it. The search continues, moving one character at a time through the target string looking for c. At the 5th character in the target, we find a c. We then check whether the c is immediately followed by a, the second literal in our pattern. This is the case so we examine the next character to see if it is t, and we find that we do not have a match because the string has a d in that location. Since the pattern has not been found, we resume the search for c at the 6th character in the target, i.e., one character past the last character that we checked to be c. The diagram below shows pictorially how the search proceeds.

cat	1	2	3	4	5	6	7	8	9	1	2	3	4	5	6	7	8	9	1	2	3	4	5	6	7		
	T	h	e	c	a	d	h	i	d	h	i	s	c	o	a	t	.	S	c	a	t	!					
Find c	x	x	x	x	x	✓																					
Followed by a							✓																				
Followed by t							x																				
Back up & resume				○																							
Find c	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	✓											
Followed by a																	x										
Back up & resume													○														
Find c													x	x	x	x	x	x	✓								
Followed by a																				✓							
Followed by t																					✓						
Match 24-26																											
Find c																										x	

Notice in the diagram that we find a match of our pattern in the target at positions 24 through 26. The match is embedded in the word Scat. That is, our pattern does not look for the word cat, just the literal pattern 'cat' and it is found within the word Scat. If we want to look for the word cat, then we can modify the pattern to indicate that we are searching for a word. We address how to do this in the next section. ■

8.2.2 Modifiers and Meta Characters

Regular expressions can be far richer than the example just described. For example, we can specify that a literal in the pattern is optional, i.e., we have a match either with or without that particular literal. We can indicate that the pattern must appear at the start or end of the string. We can allow a character in the string to match one of many possible characters. These modifications are specified with special characters, called *meta characters*; they include + * ? ^ \$. - | () [] {}. See Table 8.1 for brief descriptions of them. Regular expressions offer a rich set of meta characters for flexible pattern matching.

Example 8-6 A Simple Search Using an Anchor and Quantifier

Suppose that we want to check the subject line of an email message to see if it begins with "Re:". We can search for "Re:" using the pattern 'Re:', but if this pattern occurs anywhere in the string, it will be considered a match. For example if our subject line is the string

'It is all about Re:'

then the 'Re:' at the end of the string will match this pattern, which is not what we want. To restrict the pattern to match only at the start of the string, we use the regular expression: '^Re:'. Here the caret has a special meaning. It stands for the start of the string so a match must be at the beginning of the string. With this revision, we no longer have a match in the above string. Instead, a string such as

'Re: It's is all about Re:'

does contain a match in positions 1 through 3. The second occurrence of 'Re:' at the end of the string is still not considered a match.

If we wanted to expand our pattern matching to allow for any number of blanks before the Re: (including the possibility of no blanks), then we modify our pattern to: '^ *Re:'. Briefly, the pattern '^ *Re:' matches any string that begins with any number of blanks followed by 'Re:' so it yields a match for all three of the following strings:

```
'Re: yesterday'
' Re: yesterday'
'    Re: yesterday'
```

Notice that this pattern includes 6 characters, a caret, one blank, an asterisk, and the 3 literals 'Re:'. The caret and asterisk are meta characters. As before, the caret anchors the pattern to the start of the string. The '*' is a quantifier. It indicates that the previous literal can appear 0 or more times in the target string, e.g., in our pattern it matches 0 or more blanks. ■

Example 8-7 A Simple Search for a Word

If we wanted to revise our search (Q.8-5 (page 364)) so that it matches only the word cat, i.e., not cat appearing in some other word such as catastrophe, then we can use the pattern '<cat\>'. Here the '<' and '>' are meta characters that represent the start and end of a word, respectively. With this pattern, the word cat is found once starting at position 10 in the string,

```
'Feed the cat or there will be a catastrophe'
```

We will soon see that to use these meta characters in R , we need to use two backslashes, i.e., we write them as '\<' and '\>' because the backslash is a special character in R that is used for control characters. That is, we must indicate to R that the backslash is a meta character that is part of the regular expression. Again, Table 8.1 provides a list of some of the more commonly used modifiers. ■

8.2.3 Equivalent Characters

Some patterns present more of a challenge to express literally. For example, consider the problem of determining whether or not the username portion of an email address ends in a digit or digits. We don't care which particular digit occurs at the end of the username, only whether or not there is one of the literals 0, 1, ..., 9. Here we want to specify equivalent literals, i.e., any literal from the collection of digits is an acceptable match. Regular expressions allow us to succinctly express the concept of "match a digit", by explicitly enumerating the equivalent characters. We use the [] notation to specify a collection of characters that constitute a match; this collection is called a *character class*. To match any digit, we can use [0123456789]. Similarly to match a lower case letter, we can use

```
[abcdefghijklmnopqrstuvwxyz]
```

and to match a space or a TAB character, we use [\t]. The meta characters for expressing equivalent classes of characters are briefly described in Table 8.2.

Example 8-8 Simple Search with Equivalent Character

TABLE 8.1: Common Meta Characters

Character	Definition
?	Character or sub-pattern occurs zero or one time
+	Character or sub-pattern occurs one or more times
*	Character or sub-pattern occurs zero or more times
.	Any character
^	Anchor for the beginning of string
\$	Anchor for the end of string
\<	Beginning of a word
\>	End of a word
()	Group of literals, i.e., a sub-pattern
	Alternation, i.e., one sub-pattern or another
{ }	Quantifier: {n} means exactly n repeats of the literal or sub-pattern; {n,m} means n to m repeats; and {n,} means n or more repeats

Note to specify one of these meta characters as a literal in a pattern, it needs to be preceded with a backslash. And since the backslash has a special meaning in R, the meta character needs to be preceded with two backslashes in R. Meta characters pertaining to character classes appear in Table 8.2.

TABLE 8.2: Meta Characters for Character Classes

Character	Meaning
[]	Characters between square brackets are considered equivalent for a match.
-	Range within a character class, e.g., d-g stands for defg.
^	As the first character inside [], exclude these characters.

Characters which are considered equivalent in the search for a match. Some commonly used collections of characters are named; see Table 8.3 for examples.

Consider one of our earlier examples where we searched for the pattern cat in the string, 'The cad hid his coat. Scat!'. We can broaden our search to match cat or cad by specifying t and d as equivalent characters. We do this with the pattern 'ca[td]'. The regular expression pattern matching engine finds two matches in our string beginning at locations 5 and 24, as shown below.

<u>ca[td]</u>	1	2	3	4	5	6	7	8	9	1	2	3	4	5	6	7	8	9	1	2	3	4	5	6	7
	T	h	e	c	a	d	h	i	d	h	i	s	c	o	a	t	.	S	c	a	t	!			
Find c	X	X	X	X	✓																				
Followed by a						✓																			
Followed by t or d						✓																			
Match at 5-7																									
Find c							X	X	X	X	X	X	X	X	X	✓									
Followed by a																	✓								
Followed by t																		X							
Back up & resume																		◎							
Find c													X	X	X	X	X	✓							
Followed by a																		✓							
Followed by t or d																			✓						
Match 24-26																									
Find c																									X

Further, if we use the pattern, '`c [ao]+[td]`', then we search for a c followed by any positive number of a and o literals followed by a single t or d. Then the matching engine proceeds as shown in the diagram below.

<u>c[oa]+[td]</u>	1	2	3	4	5	7	8	9	1	2	3	4	5	6	7	8	9	1	2	3	4	5	6	7	
	T	h	e	c	a	d	h	i	d	h	i	s	c	o	a	t	.	S	c	a	t	!			
Find c	X	X	X	X	✓																				
At least 1 o or a						✓																			
t or d							✓																		
Match 5-7																									
Find c							X	X	X	X	X	X	X	X	X	✓									
At least 1 o or a																	✓	✓							
Followed by t																		✓							
Match 17-20																			✓						
Find c																			X	X	X	✓			
At least 1 o or a																				✓					
t or d																				✓					
Match 24-26																									
Find c																									X

Notice in the diagram that we now have three matches in our target string; in particular, we have picked up the four character match of `coat` that begins at the 17th character in the target string.

There are many collections of characters that are commonly used in pattern matching. For example, we often want to specify all the letters of the alphabet, lower or upper case or both. Also, we often want all the digits. In other cases, we want a subset of these sets. The character when used within the character class pattern (i.e., inside `[]`) typically identifies a range. We can specify the digits 0 through 9 more readily as `[0-9]` and the subset of the digits 3, 4, 5, 6 can be specified as `[3-6]`. Similarly, we can specify `[0-9A-F]` to match a hexadecimal digit. Also, we often see `[A-Za-z]` for all letters (upper and lower case) in the alphabet. The pattern expresses a higher level concept that is easier to read than explicitly enumerating the elements of the character set.

If we want to include the character `-` in our set of characters to match, then we must put the `-` at the beginning of the character set, otherwise it is interpreted as a range. For example, to match the basic arithmetic operators `+`, `-`, `*`, or `/`, we can use '`[-+*/]`', but not '`[+-*/]`'. Additionally, to match a digit with either a `+` or `-` in front of it, we can use `[-+][0-9]`. Notice that we have made a pattern from a sequence of two sub-patterns and each of these sub-patterns is made up using the primitive elements. The dash in the first sub-pattern is for the literal character and the dash in the second is a meta character that denotes a range.

We can also adapt this notation very slightly to indicate a match on the complement of the set of characters provided. That is, we place a caret `^` as the first character within `[]`

TABLE 8.3: Common Named Character Classes

Name	Collection of characters
[:alnum:]	All alphabetic and numeric
[:alpha:]	All alphabetic
[:lower:]	Lower case alphabetic characters
[:upper:]	Upper case alphabetic characters
[:digit:]	Digits 0123456789
[:punct:]	Punctuation characters
[:blank:]	Blank characters, i.e., space or tab
[:space:]	White space
[:cntrl:]	Control characters, e.g. new line
[:print:]	Printable characters
[:graph:]	Printable character except space

These are included in character classes as, e.g. `[[:alnum:]]`.

to indicate that the equivalent characters are the complement of the characters enumerated within, i.e., any character but the characters in `[]` is considered a match. For example, the expression '`[^a-z]`' matches any character other than a lower case letter.

Note that for some meta characters, the position of the meta character in the pattern determines whether or not it is treated as a meta character. For example, the `^` is a meta character in the above pattern '`[^a-z]`' that denotes the complement of the lower case letters; whereas, '`^Re:`' anchors the pattern to the beginning of the target string; and if the caret appears anywhere else in the pattern, then it loses its special meaning and simply is considered a literal caret (if properly escaped with backslashes).

Named Character Classes

Character classes are very convenient, and the range operator `(-)` succinctly specifies collections of characters to further simplify their use. The regular expression language also provides a collection of built-in character sets for commonly used collections. Each of these is identified by a short name. See Table 8.3 for a list of some of these named character sets.

We use these collections within the `[]`. For example, to specify the punctuation marks as equivalent characters, we use `[[:punct:]]`. Additional characters can be included in the overall set, such as `[[:digit:]A-F]` to specify that all digits and the capital letters A, B, C, D, E, and F are equivalent characters. Further, we can create an equivalence class from more than one named character class, e.g., `[[:punct:][:space:]]` matches any punctuation mark or white space.

Example 8-9 Searching for Digits in Email Addresses

The search for a digit or an underscore at the end of the username of an email address can now be easily performed with the pattern '`[[:digit:]_]@`'. This pattern finds matches in the following email addresses:

```
'depchairs03-04@uclink.berkeley.edu'
'John Nolan <nolan12345@hotmail.com>'
'Sarah Lang <sarah_@hotmail.com>'
```

Importantly, it does not match the following addresses:

```
'John the 2nd <nolan@hotmail.com>'
'Sarah_Lang <sarah@hotmail.com>'
```

The pattern '`[:digit:]_@`' matches any string that has a digit or underscore immediately followed by an '@'. Since the '@' separates the username from the domain part of an email address, this pattern will find usernames that end in a digit or underscore. We do note that if an '@' occurs somewhere else in the address, then a match will be incorrectly found, e.g., technically "very.@.weird"@strange.com is a valid email address. We do not concern ourselves with these rare cases here.

8.3 Using Regular Expressions in R

Several functions in R support regular expression matching. In this section, we demonstrate some of these functions' capabilities. Section 8.8 gives a brief summary of these functions.

Perhaps the simplest function is `sub()`, which we use to substitute one pattern for another in a string. Consider Example[ex:map] where the goal was to make files from three different sources have uniformly formatted county names. One problem we uncovered was that two of the files used a period in names such as "St. John the Baptist" while one file did not, e.g., "St John the Baptist". One way to correct this problem is to strip the period from "St." in two of the files so they match the third file. That is, we can substitute the occurrence of with that of in the files. We show how to do this in the next example with the use of `sub()`, and then we extend the example to use the function `gsub()` to eliminate all periods in the county name.

Both `gsub()` and `sub()` take three arguments: the regular expression (pattern) to match, another regular expression to use as the replacement for the matching substring, and the string(s) on which to do the matching and substitution. The "g" in `gsub()` refers to *global*, i.e., the function swaps the substitute pattern for all matches that are found in the string. The `sub()` function only replaces the first occurrence of the pattern.

Example 8-10 Removing Unwanted Periods from Strings

In this example, we expect there to be at most one occurrence of "St." in any county or parish name so the `sub()` function should work fine. We have created a small test set of county names as the character vector, i.e.,

```
countyNames
```

```
[1] "Dewitt County"           "Lac qui Parle County"
[3] "St. John the Baptist Parish" "Stone County"
```

We call `sub()` passing it the two patterns and as follows

```
sub("St.", "St", countyNames)

[1] "Dewitt County"           "Lac qui Parle County"
[3] "St John the Baptist Parish" "Stne County"
```

The 'St John the Baptist Parish' value is fixed, but a problem has arisen with Stone County. It has been changed to 'Stne County' because '.' is a meta character for any character. In other words, the pattern matching looks for three characters that begin 'St' and the third character can be any character so 'Sto' is a match and replaced with 'St'. To specify that we want to match only the literal '.', we add two backslashes to our pattern to qualify the period. That is,

```
sub("St\\.", "St", countyNames)
[1] "Dewitt County"           "Lac qui Parle County"
[3] "St John the Baptist Parish" "Stone County"
```

We have fixed the problem.

Alternatively, we may want to eliminate all periods from the county names, not just those that appear after St. We can do this with `gsub()` as follows:

```
gsub("\\.", "", countyNames)
[1] "Dewitt County"           "Lac qui Parle County"
[3] "St John the Baptist Parish" "Stone County"
```

The return value appears the same as for the previous call to `sub()`, because there is at most one period in the example strings and it appears after 'St'. We can make up another example string to further test our call to `gsub()`,

```
gsub("\\.", "", c(countyNames, 'St. Stephen Cty.'))
[1] "Dewitt County"           "Lac qui Parle County"
[3] "St John the Baptist Parish" "Stone County"
[5] "St Stephen Cty"
```

If we had called `sub()` rather than `gsub()` here, then the period at the end of 'St. Stephen Cty.' would not have been removed. ■

Another useful function is `grep()`. When supplied with a regular expression and a vector of strings, the function returns the indices of the elements of the vector where a match was found. The return value can be readily used to subset the character vector to get only the elements containing (or not containing) that pattern.

Example 8-11 Determining if a Literal Pattern Appears in a String

In an earlier example we created a regular expression to match subject lines in email that begin with , including the possibility of there being leading blanks in the string. The pattern we used was '^ *Re:'. Let's test this pattern on the following samples,

```
subjectLines
[1] "Re: a request"      "a request about Re:" "Re: Re:"
[4] "RE: too"
```

We pass our pattern and to `grep()`,

```
grep("^ *Re:", subjectLines)
[1] 1 3
```

Matches were found in the 1st and 3rd elements. The good news is that we didn't obtain a match for the second element because "Re:" does not appear at the start of the string. However, we do not match 'RE:' in the fourth element because the letter e is capitalized. If we want to find this capitalized version, then we can use the `ignore.case` parameter, i.e.,

```
grep("^ *Re:", subjectLines, ignore.case = TRUE)
```

```
[1] 1 3 4
```

Or, we can turn all the characters in `subjectLines` to lower case and appropriately modify our pattern with

```
grep("^\ *re:", tolower(subjectLines))
```

```
[1] 1 3 4
```

We can also use character classes to accomplish this task, e.g., '`^ *R[Ee]:`', if we want to localize the capitalization constraints.

The `regexpr()` function is a very useful tool for finding where in the string the match occurs. The function `regexpr()` returns more detailed information than `grep()`. It provides a) the position in the substring where the match begins, and also b) the length of the match. As an example, consider the literals '`one`' as our pattern, and our character vector is

```
oneStrings = c("a test", "a basic string", "one and one is 2",
              "and one that we want", "three two one")
```

Then the call to `regexpr()` determines where in the strings the pattern occurs,

```
regexpr("one", oneStrings)
```

```
[1] -1 -1  1  5 11
attr(", "match.length")
[1] -1 -1  3  3  3
```

The return value is an integer vector with an element for each of the elements in the supplied character vector. Each element in the return vector gives the position of the starting character of the match, if it exists, and -1 when no match occurs for that string. We see that the first two elements in do not match and the next three elements have matches starting in positions 1, 5, and 11, respectively. Also, the return vector has an attribute `match.length`, which indicates the length of the matching substring. With the length of the match in an attribute, we can treat the return value as a simple integer vector, and we can use the attribute information to extract the matched text from the string. We show how in the next example.

Example 8-12 Extracting Text from Between Brackets

For the Web log data of (Q.8-1 (page 362)), we want to extract the date and time from each log entry. That is, we want the text between the square brackets. Since square brackets do not appear elsewhere in the logs, a search for a left square bracket will bring us to the date information. Below is one line of text from the web log as a character string in *R*,

```
weblog
```

```
[1] "169.237.46.168 -- [26/Jan/2004:10:47:58 -0800]
\"GET /stat141/Winter04 HTTP/1.1\" 301 328
\"http://anson.ucdavis.edu/courses/\""
\"Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0;
.NET CLR 1.1.4322)\""
```

Notice that quotation marks appear with a backslash preceding them. The backslash acts as an escape character so the quotation mark within the character string does not end the character string.

The following call to `regexpr()` searches for the left square bracket followed by any number of characters followed by a right square bracket:

```
regexpr("\\[.*\\]", weblog)

1] 20
attr(, "match.length")
[1] 28
```

To search for a literal square bracket, we need to use the backslash twice, first to escape the use of backslash to indicate a control character in *R* and then to escape the use of [as a meta character in the regular expression language, i.e., so the square bracket is treated as a literal. The pattern '`\[.*\]`' contains two meta characters, the '.', which stands for any character, and the '*', which is a quantifier for any number of times (including zero times). Essentially, the pattern produces a match when it finds any string between [and].

The return value from our search is the integer 20, the position of the starting character of the match. In this case, it tells us that the left square bracket is the 20th character in the string. Also, the attribute `match.length` is 28, which indicates that the matching substring, from [to], is 28 characters long. To extract the date and time, we can pass these locations to the `substring()` function as follows:

```
index = regexpr("\\[.*\\]", weblog)
substring(weblog, index + 1,
          index + attr(index, "match.length") - 8)

[1] "26/Jan/2004:10:47:58"
```

Further string processing can be used to separate the date from the time. We leave this to Q.8-18 (page 379). ■

As with `sub()` and `gsub()`, the global version of `regexpr()` is `gregexpr()`. When we use this function, the return value provides all matches in each string. For this reason, the return value is a list, e.g., for the character vector we worked with previously,

```
regexpr("one", oneStrings)

[[1]]
[1] -1
attr(, "match.length")
[1] -1

[[2]]
[1] -1
attr(, "match.length")
[1] -1

[[3]]
[1] 1 9
attr(, "match.length")
```

```
[1] 3 3
[[4]]
[1] 5
attr(,"match.length")
[1] 3

[[5]]
[1] 11
attr(,"match.length")
[1] 3
```

We see that the locations of both occurrences of in the third element are returned.

8.3.1 Writing Our Own Literal Matcher

To reinforce the concept of matching literals in regular expressions and to show how useful simple string manipulation functions can be, we write our own function to match literals. This function, which we call `findPattern()`, has two arguments: `pattern`, a string which holds the sequence of literals to search for; and `strings`, a vector of target strings to search. The return value is a list the same length as `strings` where each element is a vector of locations in the corresponding string with the starting position of any matches.

To search for the pattern in a string, we can simply split the string into a vector of single characters and check for the first character in our pattern. We can use `strsplit()` to split the string. For example, for the vector of county names,

```
characters = strsplit(countyNames, "")
characters[[1]]

[1] "D" "e" "w" "i" "t" "t" " " "C" "o" "u" "n" "t" "y"
```

Then with `substring()` we can extract the first character of a pattern and compare it to each character in the string(s), e.g.,

```
firstChar = substring(pattern, 1, 1)
lapply(characters, function(oneString)
  which(firstChar == oneString))
```

Then at each location where we have a match of the first character in our pattern, we can determine whether or not the full sequence of literals appears there. Our function is as follows:

```
findPattern = function(pattern, strings) {

  firstChar = substring(pattern, 1, 1)
  numChars = nchar(pattern)
  splitStrings = strsplit(strings, "")

  firstMatches = lapply(splitStrings, function(letters)
    which(firstChar == letters))

  res = mapply(function(string, matches) {
```

```

    if (length(matches) == 0) return(integer(0))
    foundMatch = pattern == substring(string, matches,
                                      matches + numChars - 1)
    return(matches[foundMatch])
}, strings, firstMatches)
names(res) = NULL
return(res)
}

```

We try our function on a few examples with `countyNames`,

```
[1] "Dewitt County"           "Lac qui Parle County"
[3] "St. John the Baptist Parish" "Stone County"
```

Below we look for St.,

```
findPattern("St.", countyNames)
```

```

[[1]]
integer(0)

[[2]]
integer(0)

[[3]]
[1] 1

[[4]]
integer(0)
```

Also, let's try a search for the letter s, upper or lower case, with

```
findPattern("s", tolower(countyNames))
```

```

[[1]]
integer(0)

[[2]]
integer(0)

[[3]]
[1] 1 19 26

[[4]]
[1] 1
```

In [?], we add an additional argument to `findPattern()` to enable the case to be ignored in the matching.

8.4 Alternatives, Grouping, and Backreferences

The patterns that we have seen so far in this chapter use single literals and modifiers. However, we can also group characters together to make sub-patterns and, for example, apply quantifiers to them. We can provide alternative sub-patterns, and we can refer back to a sub-pattern that was found earlier in the pattern.

8.4.1 Grouping Characters into Sub-patterns

We use parentheses to group literals and meta characters and create sub-patterns within a pattern.

Example 8-13 Searching for Repeated Sub-patterns

Suppose we want to find repetitions of the pattern 'CT' in a string. We can use the pattern '(CT)+' to do this. For example,

```
gregexpr("(CT)+", "ACTCTGAGGCATCTCTCTAGC")
[[1]]
[1] 2 13
attr("match.length")
[1] 4 6
```

Notice that two occurrences of these repetitions were found in the string. ■

8.4.2 Alternative Patterns

We have seen that character classes consist of literals that are treated as equivalent in the matching process. We can also use the symbol | to create equivalent sub-patterns.

Example 8-14 Alternative Sub-patterns

Suppose we want to search for a day of the week, e.g., Sunday, Monday, etc. We can specify these alternatives with | as follows:

```
'Sunday|Monday|Tuesday|Wednesday|Thursday|Friday|Saturday'
```

Since all of the days end in 'day', we can express these alternatives more succinctly using parenthesis to create alternative sub-patterns within the pattern; that is,

```
'(Sun|Mon|Tues|Wednes|Thurs|Fri|Satur)day'
```

Here we have removed 'day' from the alternatives since they appear in all of them and at the end of the pattern. ■

Note that we can express a character class using alternation and parentheses. For example, rather than [0-9], we can use the construction: (0|1|2|3|4|5|6|7|8|9) for our pattern. This is tedious to write and becomes difficult to read as the number of characters to be matched becomes lengthy. We are not succinctly expressing the concept of "match a digit", but instead we are explicitly enumerating the characters. This makes maintaining and understanding the regular expression more difficult. Additionally, these single character alternations are not very efficient. They can slow down the speed with which the regular expression automata performs the matching.

8.4.3 Back referencing a Sub-pattern

With back referencing we can refer to a previously matched sub-pattern inside a regular expression. This can be especially helpful when, for example, we want to use a sub-pattern in a replacement pattern but the sub-pattern includes character classes or quantifiers so we don't know exactly what it looks like. Sub-patterns that appear within parentheses are automatically labeled sequentially in the pattern. The first sub-pattern labeled as `\1` (or `\\\1` in R), the next as `\2` (or `\\\2` in R), and so on. We can refer to a sub-pattern later in the pattern or in a replacement pattern by using this name.

Example 8-15 Looking for Duplicate Words

Suppose we want to find any words that have been accidentally typed twice in a string, such as

```
'Bring the the bowl and a\na cup'
```

In other words, we want to locate the double the and the double a. Notice that the double as appear on different lines, one at the end and the other at the beginning. Since we do not know what word may be repeated, we specify it using the equivalence class of letters with `'\\<[:alpha:]++\\>'` (we use the R specification with two backslashes). This pattern will match any word and we want to look for the same pattern immediately followed by a blank or new line. We use parentheses to create a sub-pattern from the word and refer to a blank, new line, or tab as follows:

```
'(\\<[:alpha:]++\\>)[:blank:]\\n\\t]\\1'
```

We test the pattern with the string, as shown in the figure below, and find two matches starting at positions 7 and 24 respectively. ADD FIGURE

8.5 Greedy Matching

Regular expression quantifiers are greedy, meaning they try to match as much as possible in a string. A quantifier such as `*` or `+` indicates that we want to match the literal or sub-pattern 0/1 or more times so the matching will include as many characters as possible. This can lead to unexpected results as shown in the following example.

Example 8-16 Removing HTML Tags from a string

Suppose we want to strip out the *HTML* tags from the following text,

```
'<h1>Hello!</h1><p>Click <a href="x23.com">here</a>'
```

An *HTML* tag begins with `<` and ends with `>`, and as seen in the sample string, it may contain many different characters in between. When we use the regular expression, `'<.*>'`, we get unexpected results. The pattern matches the entire string:

```
testString =
  '<h1>Hello!</h1><p>Click <a href="x23.com">here</a>'
gsub("<.*>", " ", testString)

[1] " "
```

The problem is greedy matching. The pattern '`<.*>`' searches for a pair of angle brackets with any characters between. Although, `<h1>` is a match, so is `<h1>Hello!</h1>` and `<h1>Hello!</h1><p>` and `<h1>Hello!</h1><p>Click ...</here`. All of these matches begin with `<` followed by *any* characters (including `>`) followed by `>`. The greedy match is the longest substring possible, which is not what we want. If we exclude the `>` from the set of equivalent characters, that should fix the problem.

```
gsub("<[^>]*>", " ", testString)

[1] " Hello! Click here "
```

That is the result we are after. Note that if we are working with *HTML* and want to extract text content from a document, there are more robust ways to do it than using regular expressions. See [?] for more information on this topic. ■

8.5.1 Lazy Matching

When greedy matching is problematic, we can often modify the regular expression slightly to remedy the problem, as in Q.8-16 (page 377). Alternatively, we can indicate that we want the matching to be lazy, i.e., we want the match to be as short as possible. We can specify that we want a quantifier such as `*` or `+` to be lazy, not greedy, by placing a `?` after it, e.g. `+?`. We revisit Example[ex:greedymatch] and show how to use lazy matching.

Example 8-17 Removing HTML Tags Using Lazy Matching

An alternative approach to stripping the *HTML* tags from the string,

```
'<h1>Hello!</h1><p>Click <a href="x23.com">here</a>'
```

is to use lazy matching. Recall from Q.8-16 (page 377) that we used the fact that tags begin with `<` and end with `>` and can contain many different characters in between to create our regular expression. However, when we tried the regular expression, '`<.*>`', we found that it matched too much because it allowed `>` to be one of the any characters. In that example, we remedied the problem by excluding `>` from the set of possible characters, with the pattern '`<[^>]*>`'. We take a different approach here by limiting the greediness of the matching. Instead, we place `?` after the quantifier `*` in '`<.*>`'; this indicates that we want the smallest match possible, e.g.,

```
gsub("<.*?>", " ", testString)

[1] " Hello! Click here "
```

This remedies the problem with greedy matching. ■

8.6 Examples

We now have the regular expression tools for addressing the text processing tasks from Section 8.1. We consider three of the four cases (web logs, election file merging, and deriving variables from email) and leave the fourth (mining the State of the Union addresses) to

the Exercises. In Q.8-18 (page 379), we split each entry into substrings that contain the various pieces of information (e.g., date, time, file name) desired. In Q.?? (page ??), we replace ampersands and eliminate periods from text to create a standard format for county names across the three files. When creating variables from the email messages in Example[ex:emailCtd], we search for text and patterns in the email headers and body and we, e.g., check the form of email addresses.

Pattern Matching Tasks

Regular expressions are typically used for one of the following types of tasks:

FIND text within a larger body of text.

VALIDATE text conforms to a desired format.

REPLACE text at matched positions.

SPLIT text into substrings.

Example 8-18 Extracting Variables: Weblogs, Ctd.

In the Web log analysis (Q.8-1 (page 362)), our ultimate goal is to transform a line in the Web log into a line of comma-separated values of the IP address, date, time, file, status, and bytes. We examined part of the problem – how to extract the date and time – in – Q.?? (page ??). Here, we tackle the original problem to transform a Web log entry:

```
193.188.97.151 - - [29/Dec/2003:06:36:18 -0600]
"GET /logo.html HTTP/1.1" 200 244 ...
```

into the comma separated values:

```
193.188.97.151, 29/Dec/2003, 06:36:18, /logo.html, 200, 244
```

which has extracted the IP address, date, time, file name, status, and number of bytes transferred.

Each of these pieces of information can be located in the file by carefully examining the structure of the file. The IP address comes first, and is always followed by two dashes. Then comes the date and time between square brackets, followed by either the command **GET** or **POST**, the file name, and HTTP or FTP all in quotation marks. Finally, the next two numbers in the log are the status and the number of bytes, respectively, and the rest of the entry can be ignored.

The notion of grouping literals into sub-patterns can be very useful here. As an example, consider the task of identifying the file name. As mentioned already, the file name appears between **GET** (or **POST**) and HTTP (or FTP), e.g.,

```
... "GET /stat141/Winter04 HTTP/1.1" 301 328 ...
```

A regular expression that extracts the file name can use this structure. That is, we can create a pattern that matches the entire weblog entry and that contains a sub-pattern of all literals between **GET** and HTTP as follows:

```
' .*"GET (.*) HTTP.*'
```

Then we can refer to the sub-pattern as '`\1`' and use `gsub()` to substitute the entire string for the sub-pattern, e.g.,

```
gsub('.*"GET (.*) HTTP.*', '\1', weblog[1])
[1] "/stat141/Winter04"
```

We have successfully identified the file name in the sample. However, when the **GET** is a **PUT** or when the **HTTP** is an **FTP**, then our regular expression will not locate the desired sub-pattern. Alternation comes to the rescue here. That is, we can search for **GET** or **PUT** by using the pattern **(GET|PUT)** and similarly we can replace **HTTP** in our regular expression with **(HTTP|FTP)**. Then we have

```
gsub('.*" (GET|PUT) (.*) (HTTP|FTP).*', '\2', weblog[1])
[1] "GET"
```

The function did not return the file name this time. What went wrong? Our regular expression has changed. Now it has three sub-patterns instead of one. The alternation **(GET|PUT)** is the first sub-pattern, and now the one that we want is the second sub-pattern,

```
gsub('.*" (GET|PUT) (.*) (HTTP|FTP).*', '\2', weblog[1])
[1] /stat141/Winter04
```

Now we have located the desired file name.

We can use this approach to identify all of the sub-patterns that we want in our output file. Consider the following regular expression,

```
pattern = '(.*).- - \[(.*):([0-9]{2}):([0-9]{2}):([0-9]{2})].*\]
  "(GET|POST) (.*) (HTTP|FTP) (/1.[01])?" ([0-9]+) (-|[0-9]+).*''
```

The first subgroup matches the IP address in the log entry, the second the date, the third the time, and so on. The subexpressions that we wish to keep are the first, second, third, fifth, eighth, and ninth. The substitution string can refer to these sub-patterns and build text that consists of these five values separated by commas: **'\1, \2, \3, \5, \8, \9'**. We use this substitution pattern in **gsub()** as follows:

```
gsub(pattern, '\1, \2, \3, \5, \8, \9', weblog[1])
[1] "169.237.46.168, 26/Jan/2004, 10:47:58, /stat141/Winter04, 301, 328"
```

The substitution string skips over the groups not needed for the output. (It is possible to avoid numbering these groups, if necessary.) ■

Example 8-19 Deriving Variables: Email, Ctd.

The task here is to create a set of features or variables for using to classify spam. Table 8.4 provides a list of variables that may prove useful in distinguishing spam.

We have already considered a few of these. For example, in Q.8-11 (page 371) we created a regular expression to determine whether or not a subject line begins with **'Re:'** and in Q.8-9 (page 369) we used a pattern to find whether or not an email address ends with a digit or underscore. Here we consider a couple more of the variables in Table 8.4 and leave the rest as exercises.

Let's consider the use of capitalization. The over use of capitalization in email is referred to as yelling, and from experience, we have seen that messages that yell a lot tend to be spam. There are several approaches to quantifying the amount of yelling in a message. We

TABLE 8.4: Variable Definition Table

Variable	Type	Definition
<code>underscore</code>	logical	TRUE if sender's address (part before the @) contains an underscore.
<code>numEnd</code>	logical	TRUE if sender's address ends in a number.
<code>subExcCt</code>	integer	Number of exclamation marks in the subject.
<code>subQuesCt</code>	integer	Number of question marks in the subject.
<code>numRec</code>	integer	Number of recipients of the message, including CCs.
<code>perCaps</code>	numeric	Percentage of capitals among all letters in the message body, excluding attachment.
<code>isRe</code>	logical	TRUE if Re : is present in the subject line of the header.
<code>subPunc</code>	logical	TRUE if words in the subject have punctuation or numbers embedded in them, e.g., w! se
<code>hour</code>	numeric	Hour of the day in the Date field.
<code>perHTML</code>	numeric	Percentage of characters in <i>HTML</i> tags in the message body in comparison to all characters.
<code>subBlanks</code>	numeric	Percentage of blanks in the subject.
<code>isYelling</code>	logical	TRUE if the subject is all capital letters.
<code>forwards</code>	integer	Number of response symbols in the message, e.g., >>> or indicates 3 rounds of communication.
<code>isOrigMsg</code>	logical	TRUE if the message body contains the phrase original message.
<code>isDear</code>	logical	TRUE if the message body begins with dear.
<code>isWrote</code>	logical	TRUE if the message contains the phrase wrote:
<code>avgWordLen</code>	numeric	The average length of the words in the message.
<code>numChars</code>	numeric	The number of characters in the message.

may, for example, look at the subject line in the header, and determine whether it is all capitals or not; we may wish to report the percentage of capital letters among all letters used in the body; or, we may count the number of lines in the body that are entirely capitalized. In the first case, it is natural to create a logical to indicate whether or not a subject is all capitals. Recall that the subject is an element of the header vector in each message element of emailStruct, and it is named Subject.

For example, meta characters make easy work of the task to derive a logical that indicates whether or not the subject line in an email is all capital letters. This phenomena in email is referred to as yelling. Here is a case when the complement of a character set is useful because we allow any character except lower case letters of the alphabet in the subject line of the email. But we face the problem of needing every character in the subject line to *not* be a lower case letter. We want to specify a pattern that consists of non-lower case letters from the beginning to end without knowing how long it is. Again, we can write a specialized function to do the work,

```
subjectLines

[1] " Re: 90 days"      "[SPAM:XXXXXXXXX]"
[3] " Fancy rep1!c@ted watches"

all(strsplit(subjectLines, "")[[1]] %in% LETTERS)

[1] FALSE
```

but regular expressions provide a clean, clear way to express this pattern via meta characters.

```
^[:lower:]*+$
```

To explain, the first character in this pattern, the `^` is the anchor for the beginning of the string, and the last character, `$` is the anchor to specify the end of a string. The asterisk denotes “any number of times” meaning that the character immediately preceding it may be repeated zero or more times. Put all together, the pattern finds a match when the string consists entirely of non-lower case letters from beginning to end. Note that the caret `^` appears twice in the pattern, and each occurrence has a different meaning. The first caret is the meta character for the beginning of line anchor, and the second caret, which is the first character inside the square brackets, represents the complement meta character that says any character that is not a lower case is a match.

```
grep("^[:lower:]*+$", subjectLines)

[1] 2
```

In this case, we could split the string into separate characters and check whether any of these are digits. Similarly, to find a fake word that contains punctuation in the middle of it, we could split the subject line into individual characters, search for punctuation or a digit, and if we find it, then look at the preceding character and the succeeding character to ascertain whether they are letters of the alphabet (upper or lower case). So any letter–punctuation or digit–letter combination is a match.

The search for a fake word that contains punctuation or a digit in the middle of it is handled by the following pattern

```
[:alpha:][:digit:][:punct:][:alpha:]
```

Paying careful attention to the square brackets in this pattern, we see that we are looking for three characters. The first can be any letter in the alphabet (upper or lower case), followed by a digit or punctuation mark, followed by another letter. Unfortunately this pattern matches the text string “it’s”, which we do not want. This problem can be resolved by providing the specific punctuation marks that are acceptable in the character class,

```
[:alpha:][[:digit:]]!@#$%^&*()?:,.[:alpha:]
```

or we could first remove any quotation marks from the search string and then use the original pattern.

```
s = c(subjectLines, "It's me")
s

[1] " Re: 90 days"      "[SPAM:XXXXXXXXX]"
[3] " Fancy rep1!c@ted watches" "It's me"

newString = gsub("'", "", s)
grep("[:alpha:][[:digit:][:punct:]][[:alpha:]]", newString)

[1] 2 3
```

Note that the search did not match the “Re:” because the colon is followed by a blank, nor does it match the fourth element because the ‘ has been removed from the string. It does find a match in the second element and the third element. To find exactly where the pattern was found in these strings, we can use the `regexpr()` function.

```
regexpr("[:alpha:][[:digit:][:punct:]][[:alpha:]]",
       newString)

[1] -1 5 13 -1
attr(,"match.length")
[1] -1 3 3 -1
```

The return value of 1 indicates that the pattern was not found in the first and fourth elements of `newString`. As for the second element, the return value of 5 indicates that the pattern was found beginning at the fifth character in the string, and the value of the attribute `match.length` for this element in the return vector indicates that the match is three characters long. The fifth through eighth characters in [SPAM:XXXXXXXXX] are M:X and so we have found the pattern we expected to find.

Notice that the match found in the third element of `newString` uncovers one more limitation in our pattern specification: the pattern was found in characters 13-15 in the string, i.e. c@t. That is, we did not find p1!c because it consists of four characters: a letter, followed by a digit, followed by a punctuation mark, followed by a letter. To search for the more general pattern of any number of digits or punctuation marks between letters, we must change the pattern as follows.

```
[:alpha:][[:digit:][:punct:]]+[:alpha:]
```

The plus sign between the second and third characters in the pattern indicates that the second character may appear one or more times.

8.7 Summary

In this chapter we introduced the basic concepts of regular expressions.

Literal String

Basic matching occurs one character at a time from left to right. Look for the first character in the pattern, when it is found in the string, see if the next character in the sting matches the second character in the pattern, and so on.

Character Sets

These are collections of equivalent characters, where a match could be any one of the characters specified in the character set. A character set is the collection of characters between [and]. Some of the most common collections are named, such as alpha for the letters of the alphabet.

Repetition

A match may be repeated a specific number of times, e.g. {m} for m times. Or a range of times, such as {m, } m or more times and {m, n} m through n times. In addition the meta characters *, + and ? denote zero or more, one or more, and zero or one, respectively. These quantifiers modify the character or group of characters that immediately precedes it.

Grouping

Parentheses can be used to form sub-patterns. Groups are useful for alternation, repetition, and referencing.

Alternation

Alternate patterns may be provided via the | symbol. For example this|that matches either this or that. Parentheses limit the alternation, e.g. th(is|at) has the same effect as the previous alternation.

References

A sub-pattern may be referred to later in the same pattern or in a substitution pattern. The reference is based on the position of the sub-pattern. The first or leftmost sub-pattern is referred to as \\1, the second as \\2, and so on.

8.7.1 Summary

8.7.2 Resources

There are many tutorials and examples on the Web that cover different uses and applications of regular expressions. There are also books on the subject that provide many more examples and a greater understanding of how regular expressions work. Most important of all, practice in creating regular expressions and testing them on data is essential to gaining both understanding and experience so that when you need to use regular expressions in handling data, they will be familiar.

<http://regexp.resource.googlepages.com/analyzer.html>

The regular expression language is a general language that has several implementations that are well-tested and efficient.

A good site <http://www.rexegg.com/>

Another site <http://www.regular-expressions.info/rlanguage.html>

8.8 Summary of Pattern Matching and String Manipulation Functions

This chapter introduced many of the functions available in *R* for working with regular expressions and strings. These are summarized in the table below.

grep() Search for the regular expression provided in a character vector of target string(s).

The return value is a vector of the indices of the target strings where a match is found. To ignore case with ASCII characters, set *ignore.case* to TRUE; to use Perl-style regular expressions, set *perl* to TRUE.

regexpr() Search for the first occurrence of a pattern in a character vector. The return value is a vector the same length as the vector of target strings and gives the starting position of the first match in each string. If no match is found in a string then 1 is returned. The return vector has an attribute, *match.length*, which provides the length of the matched text. Again, if no match is found, the attribute will be 1. To ignore case (for ASCII characters), set *ignore.case* to TRUE; to use Perl-style regular expressions, set *perl* to TRUE.

gregexpr() Search for all occurrence of a pattern in a character vector. The return value is a list of vectors with one vector for each element of the input character vector. Each vector in the list gives the starting position of all matches in the corresponding string in the input vector. As with **regexpr()**, a 1 is used for no match and the vector has an attribute *match.length*.

sub() Replace the first match of *pattern* in the target string(s) with the *replacement* pattern. The modified character vector is returned.

gsub() Replace all matches of *pattern* in the target string(s) with the *replacement* pattern. The modified character vector is returned.

substr() Retrieve the substring from the string(s) provided beginning in position *start* through position *stop*.

strsplit() Split each element of a character vector at the pattern provided in *split*. Multiple splits occur along the string if there is more than one match. If *split* has length 0 then the string is split into single characters. This function returns a list the same length as the supplied character vector. Each element of the list is a vector of substrings resulting from the split.

paste() Concatenate elements of the character vectors provided. If *sep* is provided, this character string is placed between the terms. If *collapse* is provided, the elements of the concatenated vector are collapse into one string with the string provided in *collapse* between elements.

nchar() Return the number of characters in the string provided.

nzchar() Return a logical vector where TRUE indicates the element of the supplied character vector is non-empty.

8.9 Text Mining & Natural Language Processing

Bag of Words; Stop Words; Stemming ; TF-IDF;
NLP and Parts of Speech; Taggers; WordNet

8.10 Guided Practice

The Internet Movie Database, commonly referred to by the acronym IMDb, is a website that aggregates a wide array of information regarding movies, documentaries, television shows, and all types of film media. One of its key features is that it allows users to rate movies. Using these ratings, IMDb maintains a list of the 250 most highly rated movies. The data file `top250` contains raw data taken from the html source file for this list, accessed on (insert DATE). The data in this file contains the rating, title of the movie, and the IMDb Rating, which is a weighted average of ratings given to the film by registered users of the site. The data is loaded using `load("top250.rda")`. The first six rows of the data

```
head(top250)

      Rank & Title IMDb Rating
1 1.\n      The Shawshank Redemption\n      (1994)      9.2
2 2.\n      The Godfather\n      (1972)      9.2
3 3.\n      The Godfather: Part II\n      (1974)      9.0
4 4.\n      The Dark Knight\n      (2008)      8.9
5 5.\n      Pulp Fiction\n      (1994)      8.9
6 6.\n      Schindler's List\n      (1993)      8.9
```

show that significant cleaning is needed to make the data usable. We would like to organize the data into a data frame consisting of three columns (rank, title, and year of release) with one row corresponding to each movie in the list.

We initialize this data frame and name it `top250Cleaned`, which has the same number of rows as `top250` and three columns named `rank`, `movieName`, and `year`:

```
top250Cleaned =
  matrix(nrow = nrow(top250), ncol = 3,
         dimnames = list(NULL, c("rank", "movieName", "year")))
top250Cleaned = data.frame(top250Cleaned)
```

In the following exercises, we fill in the empty columns in `top250Cleaned` by manipulating the text from the `Rank & Title` column in the `top250` data frame.

1. Extract the rank from the first column in `top250` (named `Rank & Title`). Ignore that the rank is implied by the ordering of the rows. Convert the extraction to a numeric vector for the `rank` column in `top250Cleaned`.
2. Extract the year from the first column in `top250` and create a numeric vector for the `year` column in `top250Cleaned`.
3. Extract the movie title from the first column in `top250` and create a character vector for the `movieName` column in `top250Cleaned`.

8.10.1 Answers

The exercises work with the strings contained in the first column of `top250` (named `Rank & Title`). In order to determine the general pattern of the strings, we closely examine the first six strings. The string begins with the rank, followed by a period, a newline, four whitespaces, the movie title, another newline, another four whitespaces, and the year enclosed in parentheses. There are many different ways to extract the information required by the exercises from these strings. The following answers provide several possible solutions.

1. One way to extract the rank is to extract the first three characters of the string and then remove everything that is not a number:

```
firstThree = substr(top250[,1], 1, 3)
head(firstThree)

[1] "1.\n" "2.\n" "3.\n" "4.\n" "5.\n" "6.\n"

rank1 = gsub("[^[:digit:]]", "", firstThree)
```

This works because the rank is a number between 1 and 250 and it is at the beginning of the string. We coerce our extraction to a numeric vector for the `top250Cleaned` data frame and check that we get the answer we expect:

```
rank1 = as.numeric(rank1)
range(rank1)

[1] 1 250
```

The extracted rank is between 1 and 250 – exactly as we expected! Sanity checks such as these are good practice when cleaning data, but they do not substitute for careful methodology. It is important that we understand the underlying format of the strings before using this method. Suppose that the first string was instead ‘1 The Shawshank Redemption (1994)’. This methodology would then produce the wrong output for the seventh string, ‘7 12 Angry Men (1957)’.

Another way to extract the rank is to use the fact that the string starts with the rank followed by a period and a newline, thus the string can be separated into substrings separated by the period and newline:

```
splitPattern = strsplit(top250[,1], ".\n", fixed = TRUE)
```

or

```
splitPattern = strsplit(top250[,1], "\\\.\\\\n")
```

This results in a list with a length of 250. We examine the first and last elements of the list:

```
splitPattern[[1]]

[1] "1"
[2] "    The Shawshank Redemption\n        (1994)"

splitPattern[[250]]
```

```
[1] "250"
[2] "    La Dolce Vita\n      (1960)"
```

Each element in the list is a character vector of length two: the first substring is the rank and the second substring is the rest of the string without the rank and period. To extract the rank, we extract the first element from each element in the list:

```
rank2 = sapply(splitPattern, function(x) x[1])
```

which returns a character vector of the ranks. Again, we coerce the string into a numeric vector for the `top250Cleaned` data frame and check our work:

```
rank2 = as.numeric(rank2)
range(rank2)
[1] 1 250
```

A more direct way is to extract the rank by specifying the regular expression pattern:

```
rank3 = gsub("[[:digit:]]{1,3}\\.\\.*", "\\\\1", top250[,1])
```

We explain the pattern in the first argument:

- (a) The pattern `[[:digit:]]{1,3}` matches any run of digits that are between one and three characters long.
- (b) The pattern `\.\.` is the period that follows the rank, which needs to be escaped with `\.`.
- (c) The pattern `.*` matches any run of characters, which is a catch-all for the movie name, its year, and whitespaces.

The run of digits is enclosed with parentheses so that it can be extracted with the second argument of `gsub()`: `"\\\\1"` says to extract the stuff inside the first set of parentheses from the first argument. We check our work again:

```
rank3 = as.numeric(rank3)
range(rank3)
[1] 1 250
```

Since all three methods produce the same result, we fill in the `rank` column of `top250Cleaned` with one of our extractions:

```
top250Cleaned$rank = rank1
```

2. We show three similar ways to extract the year from the string as we did in the previous exercise. Since the year is at the end of the string, one way to extract the year is to first count the number of characters with `nchar()` and then use this number to extract the year with `substr()`. Specifically, the last 5 characters contains the year and the closing parenthesis so the extraction must exclude the last character (the closing parenthesis) of the string. For example, the last five characters of the first string is '1994)', the release year of the top ranked movie, The Shawshank Redemption. Here is our method in action:

```
len = nchar(top250[,1])
year1 = substr(top250[,1], len - 4, len - 1)
```

The second and third arguments provide the starting and ending indices of the year. We check our work by first coercing the resulting vector to a numeric vector and then checking its summary statistics:

```
year1 = as.numeric(year1)
summary(year1)

  Min. 1st Qu. Median   Mean 3rd Qu.   Max.
 1921     1963    1990    1983    2003    2014
```

The output looks reasonable: all values are between 1921 and 2014. We emphasize again that this sanity check is important. This method would fail if there was a whitespace after the closing parenthesis, e.g., if the first string was ‘1.\n The Shawshank\nRedemption\n (1994)’ instead of ‘1.\n The Shawshank Redemption\n (1994)’. Though the extra space seems innocuous, this particular method to extract the year assumes regularity in the string being examined.

We show another method that assumes less regularity in the pattern. Knowing that the year is enclosed within a set of parentheses, we split the string using the parentheses, which should yield the year in its own string:

```
splitPattern = strsplit(top250[,1], "[\\(\\)]")
```

This results in a list with a length of 250. We examine the first element of the list:

```
splitPattern[[1]]
[1] "1.\n      The Shawshank Redemption\n      "
[2] "1994"
```

We notice that it is a character vector of length two: the second substring is the year and the first string is the rest of the string without the year enclosed in the parentheses.

```
year2a = sapply(splitPattern, function(x) x[2])
```

or

```
year2a = sapply(splitPattern, function(x) rev(x)[1])
```

Again, this assumes some regularity in the string. A safer way is to find the string that has exactly four digits. For example, to find the release year of *The Shawshank Redemption*, we use the `grep()` function as follows:

```
grep("^[:digit:]{4}$", splitPattern[[1]], value = TRUE)
[1] "1994"
```

The pattern “`^[:digit:]{4}$`” says the string must contain exactly four digits; no more, no less. We set the argument `value` to TRUE so that the `grep()` function returns the value of the element that matches the string rather than its index. To obtain the year of release for all movies, the same idea is applied to every element in the list:

```
year2b = sapply(splitPattern, function(x) grep("^[:digit:]{4}$", x, value = TRUE))
```

Since we know that the first extraction method passed sanity checks, let compare the outputs from the new extraction method against the output from the first method. First, we coerce the two vectors created by our new method:

```
year2a = as.numeric(year2a)
year2b = as.numeric(year2b)

[1] TRUE
```

and then check if they are equal to the output from the first method:

```
all.equal(year1, year2a)

[1] TRUE
```

```
all.equal(year1, year2b)

[1] TRUE
```

All methods are equivalent to each other.

A more direct way is to extract the year using our understanding that the year is enclosed in parentheses:

```
year3 = gsub(".*\\((([:digit:]])\{4})\\)", "\\\\1", top250[,1])
```

We explain the pattern in the first argument:

- (a) The pattern `.*` matches any run of characters, which is a catch-all for the rank, the movie name, its year, and whitespaces.
- (b) The pattern `\\\(` is the opening parenthesis, which needs to be escaped with `\\\`.
- (c) The pattern `(([:digit:]])\{4}` matches any run of digits that are exactly four long.
- (d) The pattern `\\\)` is the closing parenthesis.

The run of four digits is enclosed with parentheses so that it can be extracted with the second argument of `gsub()`: `"\\\\1"` says to extract the stuff inside the first set of parentheses from the first argument. Even though this seems very robust, one must always think of ways in which it could be broken. For example, there exists a film titled '(500) Days of Summer', although it is not in the IMDB top 250. This film comes very close to breaking this method of finding the year. For example, if it had been named '(5000) Days of Summer' instead, it would have! We check if this extraction matches up with the extraction methods from before:

```
year3 = as.numeric(year3)
all.equal(year1, year3)

[1] TRUE
```

They do match up.

Lastly, we fill in the `year` column of `top250Cleaned` with one of our extractions:

```
top250Cleaned$year = year1
```

3. Unlike the rank or year extraction, we can't simply count the number of characters in the string to extract the movie title. The other two methods do work, however. We begin by splitting the string. Since the movie title is enclosed by a newline followed by four white spaces, let's split on the white space pattern:

```
splitPattern = strsplit(top250[,1], "\\\n")
```

Let's examine the first element:

```
splitPattern[[1]]  
[1] "1."                                "The Shawshank Redemption"  
[3] "(1994)"
```

This gives a list of length 250, where each element in the list is a character vector of length three: the first string is the rank followed by a period, the second string is the movie title, and the third string is the year enclosed by parentheses. Since the movie title is the second element in the vector, the movie is extracted as

```
movie1 = sapply(splitPattern, function(x) x[2])
```

We check if the extraction looks good:

```
head(movie1)  
[1] "The Shawshank Redemption" "The Godfather"  
[3] "The Godfather: Part II"    "The Dark Knight"  
[5] "Pulp Fiction"           "Schindler's List"
```

It does!

Alternatively, we directly use our knowledge that the movie title is between two swathes of whitespaces:

```
movie2 = gsub(".*[:space:]{5}(.*)[:space:]{5}.*", "\\1", top250[,1])
```

We explain the pattern in the first argument:

- (a) The pattern `.*` matches any run of characters, which is a catch-all for the rank followed by the period.
- (b) The pattern `[[:space:]]{5}(.*)[:space:]{5}` is any run of characters, i.e., the movie name, sandwiched between two runs of five whitespaces.
- (c) The last pattern `.*` is a final catch-all for the year enclosed by parentheses.

This gives us the same output as `movie1`:

```
all.equal(movie1, movie2)  
[1] TRUE
```

Now, we can fill in the `movieName` column of `top250Cleaned` with one of our extractions:

```
top250Cleaned$movieName = movie1
```

8.11 Exercises

Paragraph about the State of the union addresses



Q.1 First question

Paragraph about derived variables for SPAM



Q.2 Write a function to handle an alternative approach to measure yelling: count the number of lines in the email message text that consist entirely of capital letters. Carefully consider the case where the message body is empty. How do you modify your code to report a percentage rather than a count? In considering this modification, be sure to make clear how you handle empty lines, lines with no alpha-characters, and messages with no text.

Q.3 Write alternative implementations for the `isRe()` function. For one alternative, instead of only checking whether the subject of a message begins `Re:`, look also for `Fwd~ : Re: .` For a second alternative, check for `Re:` anywhere in the subject, not just at the beginning of the string. Analyze the output from these 3 functions, including the original `isRe()` function in [?]. How many more messages have a return value of `TRUE` for these alternatives, and are they all ham? Which do you think is most useful in predicting spam?

Q.4 Choose several of the ideas listed in Table 8.4 for deriving features from the email and write functions for them. Be sure to check your code against what you expect. Try writing one of these functions in two different ways and compare the output from each. Use exploratory data analysis techniques to check that your code works as expected. Does the output of your function match the values in the corresponding columns of `emailDF`? If not, why do you think this might be the case? Does it appear that this derived variable will be useful in identifying spam or ham?

Bibliography

9

Relational Databases and SQL

CONTENTS

Bibliography	393
--------------------	-----

Bibliography

10

Shell Tools

CONTENTS

10.1	Digital Information	396
10.1.1	Bits and Bytes	396
10.1.2	Representing Numbers in Binary	396
10.2	Files and File Systems	396
10.2.1	Plain Text Files	396
10.2.2	Binary Files	396
10.2.3	File Systems	396
10.2.4	Permissions	396
10.3	Basics of a Shell Command	396
10.3.1	Syntax	396
10.3.2	Getting Help	396
10.4	Managing and Navigating a File System	396
10.4.1	Wildcards	396
10.5	Redirection	396
10.6	Managing Processes with the Shell	396
10.6.1	Running in Batch	396
10.7	Remote Login	397
10.7.1	Screen/tmux	397
10.8	Cases	397
10.9	Advanced Topics	397
10.9.1	grep	397
10.9.2	cut	397
10.10	Summary	397
10.11	Guided Practice	397
10.12	Exercises	397
	Bibliography	397

10.1 Digital Information

10.1.1 Bits and Bytes

10.1.2 Representing Numbers in Binary

10.2 Files and File Systems

10.2.1 Plain Text Files

10.2.2 Binary Files

The *file* command.

Shell

```
file plot.png
```

10.2.3 File Systems

10.2.4 Permissions

10.3 Basics of a Shell Command

10.3.1 Syntax

10.3.2 Getting Help

10.4 Managing and Navigating a File System

10.4.1 Wildcards

10.5 Redirection

10.6 Managing Processes with the Shell

job control, ps, kill, top.

10.6.1 Running in Batch

10.7 Remote Login

10.7.1 Screen/tmux

10.8 Cases

10.9 Advanced Topics

I'd move these up to the regular part of the chapter and integrate them into regular commands, redirection, etc.

10.9.1 grep

10.9.2 cut

10.10 Summary

10.11 Guided Practice

10.12 Exercises

Bibliography

11

XML

CONTENTS

11.1	Overview of <i>XML</i>	399
11.2	Hierarchical Structure	407
11.3	<i>XML</i> Namespaces and Additional <i>XML</i> Elements	410
11.4	The Document Object Model (<i>DOM</i>)	412
11.5	Accessing Nodes in the <i>DOM</i>	414
11.6	<i>XPath</i> and the <i>XML</i> Tree	421
11.7	<i>XPath</i> Syntax	425
11.7.1	The Axis	426
11.7.2	The Node Test	428
11.7.3	The Predicate	429
11.8	Summary of Functions to Read <i>HTML</i> , <i>XML</i> , and <i>JSON</i> into <i>R</i> Data Frames and Lists	430
	Bibliography	431

11.1 Overview of *XML*

The eXtensible Markup Language (*XML*) [16] provides a general approach for representing all types of information such as data sets containing numerical and categorical variables; spreadsheets; visual graphical displays such as *SVG*; descriptions of user interfaces; social network structures; text documents such as word processing documents and slide displays; RSS (Rich Site Summary or alternatively Real Simple Syndication) feeds; data sent to and from Web services; settings and preferences on computers; *XML* databases; and more. *XML* is so generic, it can be used to represent any data. Over the last fifteen years, *XML* has grown from a proposed simplification of *SGML* to a widely adopted and used technology in a multitude of areas, and today claims a plethora of many powerful real-world applications related to the management, organization, dissemination, and display of a broad array of data. As data scientists, we encounter *XML* on a daily basis in most aspects of data technologies.

This ubiquity and broad set of applications makes a compelling case for why anyone working with data needs to have some familiarity with *XML*. Although not rocket science, *XML* is much more than just the syntax or general format that is the *XML* specification. *XML* also includes the collection of inter-related specifications and technologies, and we will look at one of these in some detail: *XPath*, which is a powerful and flexible tool for locating content.

XML is not itself a language for representing data. Rather, it is a very general structure with which we can define any number of new formats to represent arbitrary data. *XML* provides the basic, common, and quite simple structure and syntax for all of these “dialects” or vocabularies. For example, the reader who has read or composed *HTML* (Hy-

perText Markup Language) will recognize the format of *XML* because *HTML* is a particular vocabulary of *XML*.

The basic unit in *XML* is the *element*, which we also refer to as a *node* when we talk about the hierarchical or treelike structure of the *XML* document. An element has a name, and may have attributes and child elements, each of which we describe below.

Element Names

Each element begins with the *start-tag*, e.g., `<title>` or `<article>`, and must close with a corresponding *end-tag*, e.g., `</title>` or `</article>`, respectively. That is, tags are paired and they delimit an element and its contents. In general, the start tag has the format `<tagname>` and the matching end-tag is identical except for the addition of the forward slash between the `<` and the tag name. In many respects, pairs of opening and ending tags are like parentheses, but with names that make it easier to identify the pairs when the elements are nested hierarchically.

Child Elements and Recursive Structure

XML elements can have content made up of other *XML* elements that are treated as child elements. It is this nested/recursive structure that allows us to represent different, complex data structures using *XML*. The *XML* content below (adapted from http://www.w3schools.com/XML/plant_catalog.xml)

```
<plant>
  <name>
    <common>Bloodroot</common>
    <botanical>Sanguinaria canadensis</botanical>
  </name>
  <zone>4</zone>
  <light>Mostly Shady</light>
  <price>$2.44</price>
</plant>
```

illustrates how the `<plant>` element contains four child elements: `<name>`, `<zone>`, `<light>`, and `<price>`. The `<zone>`, `<light>`, and `<price>` elements have content in the form of text elements, e.g., `Mostly Shady`. The `<name>` element also has child elements: `<common>` and `<botanical>`. Child elements are typically either regular *XML* elements, with a start and end tag, or simple text content. Text elements are simple *XML* elements that can have no children. We can also have other types of *XML* elements such as comments, processing instructions, etc. However, the data in an *XML* document are mostly in regular elements, text nodes, and attributes.

The `<para>` element in the first document in Section 11.1 illustrates how we can mix regular elements and text as child nodes.

```
<para>
This is text ...
and some <r/> code with <r:func name="table"/>
<r:code r:width="50"><![CDATA[
x <- (y > 1 & z < 0)
table(x)
]]>
<r:output>
FALSE  TRUE
 13    29
</r:output>
```

```
</r:code>
<para>
```

The text nodes, e.g., "This is text ... and some" and " code with", are interspersed with `<r>`, `<r:func>`, and `<r:code>` elements. The `<r:code>` element contains both text (within what is called a `<CDATA>` or "character data" section) and another element `<r:output>`.

This recursive structure of *XML* elements gives rise to trees, or hierarchies, of elements. Each *XML* document has a single root, or top-level, element. In our two example documents from Section 11.1, these are `<article>` and `<dataFrame>`.

Attributes

A regular *XML* element can have zero or more attributes, which are `name="value"` pairs. Attributes are specified in the start tag of an *XML* element. The attribute value must be contained within quotes, either a pair of single or double quotes. For example, the `<dataFrame>` tag from Section 11.1

```
<dataFrame numObs="10" year='2012'>
```

contains two attributes named `numObs` and `year`. There is much debate over whether to use attributes or put the values as children within an element. For example, we could have represented the `dataFrame` element in an equivalent form as

```
<dataFrame>
  <numObs>10</numObs>
  <year>2012</year>
  <source>
    ...
</dataFrame>
```

We typically use an *XML* attribute when describing metadata about the data within the element. For example, an attribute might specify the number of observations in the data frame; a printing option such as the number of digits; or whether or not to evaluate the code in an `<r:code>` node. The attributes keep this information separate from the content, e.g., whether or not to evaluate code is separate from the actual code within an `<r:code>` element. Note that attribute names must be unique within an element, i.e., we cannot have two attributes with the same name.¹ When the need to have duplicates arises, it often suggests using child elements rather than attributes.

There are some attributes that are special in *XML*. One of them is the `id` attribute. For example, the `<article>` element has an `id` attribute with value `"DataAnalysis"`. We use the `id` attribute to assign a unique identifier to an element (and its subtree). These unique identifiers or elements are very useful when extracting portions of a document or creating cross references in a document. The value *must* be unique across all `id` attribute values within the entire document and an *XML* parser will typically insist on this or produce a warning or error. The attribute `xml:lang` (or often simply `lang`) is another special attribute. This is used to identify the spoken or natural language in which the content is written, such as English or French.

Merging Start and End Tags for an Element

Basically, each *XML* element has a start tag and matching end tag. However, the end tag is unnecessary when the element has no child elements. In these cases, we can use the short version `<tagname/>`, i.e., ending the start tag with a `/` character. The `<r/>` element is

¹We can have two attributes with the same name if they belong to different *XML* namespaces.

an example of this. The start tag and end tag have been collapsed or contracted into one tag. There are several reasons why we may have an element with no content. In the case of `<r/>`, the tag identifies a particular concept—the *R* language—without the need for any additional qualification or parameterized content. Another reason is that the content may be specified in the tag's *attributes*. For example, `<r:func name="table"/>` provides the name of the *R* function `table()` via the `name` attribute on the tag. Also note that an empty element may be specified in its long form, e.g.,

```
<r:func name="runif"></r:func>
```

Collapsing the empty element into one tag is optional. Different pieces of software can produce the *XML* either way. It does not make any difference when we parse the *XML* content as the parser will create an *XML* element for us and we never need to look at the start or end tag.

Well-formed XML

XML is a specification of a general structure for describing content. It defines the basic and general syntax for specifying elements, attributes, and hierarchical content by nesting elements as children of other elements. *XML* also provides constructs such as entities and `<CDATA>` sections, comments, and processing instructions, but it does not define any element or attribute names. Instead, it is just an overarching framework or scaffolding that allows anyone to define an *XML* vocabulary made up of element and attribute names and to give meaning to those elements and the relationships between them. *XHTML* is one *XML* vocabulary, introducing elements such as `<html>`, `<body>`, `<table>`, `<a>`, and `` for describing Web pages. *SVG* is another vocabulary introducing elements such as `<circle>`, `<line>`, `<text>`, `<path>`, `<g>` (for group), `<animate>`, etc., for describing two-dimensional, interactive, and animated graphical displays. *KML* is a markup language for describing three-dimensional geographical information and displays. Each of these uses the same *XML* syntax and structure, but defines its own elements and attributes and their meanings. This is somewhat analogous to words, sentences, paragraphs, sections, and chapters. These hierarchical structures are common to many different languages, yet each of these languages uses different words. Furthermore, with the same basic structure, we can produce very different types of documents in any one of these languages.

When a document obeys the basic syntax rules of *XML*, it is said to be *well-formed*. The criteria are very general in nature, and do not pertain to a specific grammar, i.e., a specific set of allowable element and attribute names. The following few simple rules are an important subset of these syntactic requirements for a well-formed *XML* document; they cover the vast majority of cases that we encounter when we work with *XML* to access data. *XML* documents that are not well-formed produce potentially fatal errors or warnings when read.

Properties of Well-formed XML

Well-formed *XML* adheres to the following rules:

- One root element completely contains all other elements within the document, excluding the *XML* declaration and optional comments or processing instructions that may appear before the root node.
- Elements must nest properly, i.e., be opened and closed in the same order.
- Element names or tags are case sensitive.

- Tags must close, e.g., `<title>My article</title>`, or be self-closing, e.g., ``.
- Attributes appear in start-tags of elements, and never in end-tags.
- Attribute values have a `name="value"` format and the value must be quoted either with matching single or double quotes, but not mixed.
- Attribute names cannot be repeated within a given element (except if they are within different namespaces).
- No blank space is allowed between the `<` character and the tag name. Extra space is allowed before the ending `>` in the opening and closing tag. The blank space after the element name is to separate the tag from the first attribute, if it is present.
- Element and attribute names must begin with an alphabetic character or an underscore `_`; subsequent characters may include digits, hyphens, and periods. No space, colon, or the triple `"xml"` may appear in a tag name.
- The colon character is used for namespace prefixes. The namespace must be defined in the node or one of its ancestors.
- Within attribute values, the special characters `&` and `<` must be specified as entities, e.g., `company="AT&T"`.

There are additional rules that must be adhered to in order to be well-formed, but those found here cover the vast majority of the cases.

Most of the remaining syntax rules cover circumstances that many of us are not very likely to encounter. They will not be discussed here. For more details on the rules for well-formed XML see [7] and [14].

An XML document may be well-formed but “not valid”. Well-formed means only that the document conforms with the basic XML structure. It implies nothing about the validity of the content or meaning of the document. It may contain elements or attributes that make no sense or have no definition within the vocabulary. Similarly, it may have elements that are in the wrong location or incorrect order. XML schema and DTDs (Document Type Definitions) are used to specify the validity rules for a particular XML vocabulary. These will not be further discussed in this chapter. We provide a few examples to highlight the important features of XML, e.g., that it is self-describing, separates content from form, and is easily machine generated.

Example 11-1 A Climate Science Modelling Language (CSML) Document

The Climate Science Modelling Language (CSML <http://ndg.nerc.ac.uk/csm1>) was developed by the British Atmospheric Data Centre and British Oceanographic Data Centre through the UK’s Natural Environment Research Council’s (NERC) DataGrid project [9]. Rather than starting from scratch to create their vocabulary for climate science modeling, NERC built on an existing grammar, Geographic Markup Language (GML), which already had many of the needed XML elements and features for CSML. This is an example of the *extensibility* represented by the “X” in XML.

The following snippet of CSML data contains daily rainfall measurements at a specified location for each day in the month of January. These measurements (5 3 10 1 2 ...) are:

are the text content of the `<gml:QuantityList>` element, which is an element within `<PointSeriesFeature>`.²

```

<gml:featureMember>
  <PointSeriesFeature gml:id="feat02">
    <gml:description>
      January timeseries of raingauge measurements
    </gml:description>
    <PointSeriesDomain>
      <domainReference>
        <Trajectory srsName="urn:EPSG:geographicCRS:4979">
          <locations>0.1 1.5 25</locations>
          <times frame="#RefSys01"> 1 2 3 4 5 6 7 8 9 10 11
            12 13 14 15 16 17 18 19 20 21 22 23
            24 2 26 27 28 29 30 31
          </times>
        </Trajectory>
      </domainReference>
    </PointSeriesDomain>
    <gml:rangeSet>
      <gml:QuantityList uom="udunits.xml#mm">
        5 3 10 1 2 8 10 2 5 10 20 21 12 3 5 19 12
        23 32 10 8 8 2 0 0 1 5 6 10 17 20
      </gml:QuantityList>
    </gml:rangeSet>
    <parameter xlink:href="#rainfall"></parameter>
  </PointSeriesFeature>
</gml:featureMember>

```

Notice that the tag name, `<gml:QuantityList>`, begins with the prefix `gml:` while `<PointSeriesFeature>` has no prefix. The prefix distinguishes GML element identifiers from CSML element identifiers. In creating CSML, NERC began with the element names and structure of GML and added to it tags needed for climate science. We can simply add new element and attribute names to an XML vocabulary without using a namespace. However, if we use the same name for a different concept, we have a conflict. We use a namespace to avoid this conflict.

The creators of CSML decided to clearly separate the additional CSML elements by using a separate namespace for them. The sample XML suggests that this situation is reversed, i.e., that the GML elements have a namespace prefix and the CSML elements do not. In fact, both sets of elements have a namespace, but the CSML elements use the default namespace (defined earlier in the document). The majority of the elements come from the CSML vocabulary so we use that as the default namespace to reduce the number of prefixes. We can equivalently use explicit namespace prefixes for both sets of elements, or use the GML namespace as the default and so qualify `<PointSeriesFeature>`, `<PointSeriesDomain>`, etc. We can use any prefix, and are not required to use the name of the vocabulary, e.g., "gml". The namespace is defined by specifying a URI and a document-local prefix.

We leave this example with one final observation. The CSML snippet shows several layers of nesting. We use indentation to make the nesting of the elements clear and emphasize the

²The values can be marked up individually, e.g., within `<value>` or `<double>` elements. However, since the values do not contain white space, we can separate them by spaces and recover them faithfully.

tree structure. This *hierarchical structure gives great flexibility in describing complex data structures*. We can represent linear, tree, and even graph structures easily with *XML*. This generality, flexibility, and expressiveness make *XML* useful.

Example 11-2 A Statistical Data and Metadata Exchange (SDMX) Exchange Rate Document

The European Central Bank (ECB) provides daily foreign exchange rates between the euro and the most common currencies [4]. These are provided in several file formats, including an *HTML* format for the iPhone and two *XML* formats. Both *XML* formats were developed in accordance with the Statistical Data and Metadata Exchange initiative [3]. These foreign exchange reference rates (eurofxref, for short) use both the SDMX-EDI (GESMES/TS, GEneric Statistical MEssage for Time Series) format (<http://sdmx.org/>) and the ECB's extension vocabulary. The ECB uses this format to exchange data with its partners in the European System of Central Banks. According to them, this format "was a key element in the statistical preparations for Monetary Union and has proved both efficient and effective in meeting the ESCB's rapidly evolving statistical requirements" [5].

An *XML* snippet of exchange rates for four currencies on two days is shown below.³ Notice the extensive use of attributes in this vocabulary and little use of text elements for representing data. The attributes *time*, *currency*, and *rate* contain, respectively, the date, name of the currency, and exchange rate to buy one euro in this currency.

```
<Envelope>
  <subject>Reference rates</subject>
  <Sender>
    <name>European Central Bank</name>
  </Sender>
  <Cube>
    <Cube time="2008-04-21">
      <Cube currency="USD" rate="1.5898"/>
      <Cube currency="JPY" rate="164.43"/>
      <Cube currency="BGN" rate="1.9558"/>
      <Cube currency="CZK" rate="25.091"/>
    </Cube>
    <Cube time="2008-04-17">
      <Cube currency="USD" rate="1.5872"/>
      <Cube currency="JPY" rate="162.74"/>
      <Cube currency="BGN" rate="1.9558"/>
      <Cube currency="CZK" rate="24.975"/>
    </Cube>
  </Cube>
</Envelope>
```

This document appears quite different from the *XML* in Q.11-1 (page 403). Here the snippet shows three levels of tags with the same name, i.e., *<Cube>*, and each has no text content. All of the relevant information is contained in the attribute values of the *<Cube>* elements. There is one parent *<Cube>* that holds all of the others. The next layer of *<Cube>* elements pertain to the date; there is one *<Cube>* element for each day, where the *time* attribute identifies the specific date, e.g., "2008-04-17". Within each of these "time"

³ The gesmes prefix on the *<Envelope>*, *<subject>*, *<Sender>*, and *<name>* nodes has been omitted to focus on the structure of the document. These tags are part of the SDMX-EDI (also known as GESMES) grammar. The *<Cube>* tags belong to the eurofxref extension vocabulary.

cubes are four `<Cube>` elements, one for each currency (US dollar, Japanese yen, Bulgarian lev, and Czech koruna). These innermost `<Cube>`s provide the name of the currency in `currency` and the exchange rate for that currency in `rate`. The exchange rate is for the date found in the parent `<Cube>` in which the element is nested. The `<Cube>` element corresponds to a multidimensional data cube, which represent the dimensions in a generic way. The values are grouped by time, space/geography, and other variables as `<Cube>`s, with arbitrary metadata associated with each dimension.

The eurofxref grammar uses the SDMX cube model for data. That is, the data are viewed as an n-dimensional object where the value of each dimension is derived from a hierarchy. According to SDMX [10]: “The utility of such cube systems is that it is possible to ‘roll up’ or ‘drill down’ each of the hierarchy levels for each of the dimensions to specify the level of granularity required to give a ‘view’ of the data.”

This grammar offers an example where the attributes contain information used to differentiate one tag with the same name from another. The regular structure of the document is clear and can be *easily exchanged with other applications*. For comparison, an alternative format would be to use, e.g., `<date>` and `<rate>` tags where the `<rate>` has a currency attribute as follows:

```
<Cube>
  <date>
    2008-04-21
    <rate currency="USD">1.5898</rate>
    <rate currency="JPY">164.43</rate>
  </date>
</Cube>
```

Of course, there are many other possibilities for structuring this information. Provided it follows the syntactical requirements of *XML*, you can develop any vocabulary you want. However, the consumers must be able to understand and make sense of the content. ■

The strengths of *XML* are also the sources of its problems. Although *XML* is structured to make it easy for a computer to read, this structure leads to relatively verbose content. This can be overcome by compressing the *XML* content. Since *XML* is simple text and human-readable, we are tempted to edit it manually. However, many people feel that *XML* is awkward and cumbersome to work with directly. This means tools to visualize and manipulate *XML* content are highly desirable, i.e., software that reads and writes the *XML* and insulates us from the details. This is not a simple problem because *XML* is extremely general (another one of its features!). However, many communities have been developing tools for viewing their own types of data, and often these tools can be shared across communities and applications. Fortunately, some general tools for editing and reading *XML* have emerged and can be easily customized to various needs. Furthermore, because of the plethora of general *XML* tools and technologies, we can build customized tools to manipulate *XML* content. This allows us to avoid working with the *XML* directly, and also makes the process more robust and reproducible.

In comparison to *JSON*, another format for data exchange, there are facilities in most languages for reading *JSON* directly into data structures in that language. The *JSON* format only has the standard and common data structures such as logicals/booleans, real numbers, strings, and ordered and associative arrays. There is a natural and simple mapping from *JSON* to native data structures that needs no human interpretation. We can rearrange the data after reading them into these basic structures. In contrast, while most languages allow us to readily parse an *XML* document, there is another step in interpreting the *XML* data and mapping it to appropriate data structures in the language. Many *XML* documents

are quite simple and map directly to native data structures, e.g., lists and data frames. However, since we can define complex data structures in *XML*, it is natural that we have to explicitly map these to more complex native data structures in different languages. This is not really different from post-processing *JSON* data into different and more complex data structures. For *XML* however, we can programmatically read *XML* schemas, which describe classes of *XML* documents. These tell us about the data types of different *XML* elements, using the much richer set of data types available in *XML* such as different types of numbers, times, dates, tokens, and identifiers. We can also use the schema to generate code and data type definitions that can read and convert *XML* documents to complex data types in our language.

11.2 Hierarchical Structure

The conceptual model of an *XML* document as a tree can be very helpful when processing and navigating the document. In fact, one of the two standards for parsing *XML* documents is the Document Object Model, or DOM for short, which reads the *XML* content and returns a data structure that represents the document in a hierarchical form.

The XML Tree Structure and Terminology

The hierarchical nature of *XML* can be represented as a tree. In the tree, the lines connecting elements are called branches, and the elements are referred to as nodes. The names used to describe relationships between nodes are modeled after family relations.

root The sole node at the “top” of the tree.

branch A directed edge that links one node to another. Branches emanate from the root “downward”.

parent, child The branch between two nodes links a parent to a child node. The parent is one step higher in the hierarchy, i.e., closer to the root node. There are no branches between siblings in the tree.

ancestor, descendant Any node on the path up the tree from a node to the root is an ancestor of that node, and that node is a descendant of its parent and its parent’s ancestors.

sibling Nodes with the same parent.

leaf-node Nodes with no children, also known as *terminal* nodes. Text nodes do not have children and so are leaf nodes. Regular *XML* elements may not have children and so are also leaf nodes.

The *depth* of a node is the number of branches to traverse from the root to that node. The root node is at depth zero. The set of all nodes at a given depth is called a level of the tree.

The nesting syntax rules for *XML* (i.e., opening and closing tags must be balanced) means that elements in a document must be hierarchically structured. Each element can be thought of as a node in the hierarchy where branches emanate from the node to those

elements that it immediately contains. Figure 11.1 shows the tree representation of the XML document of Q.11-1 (page 403).

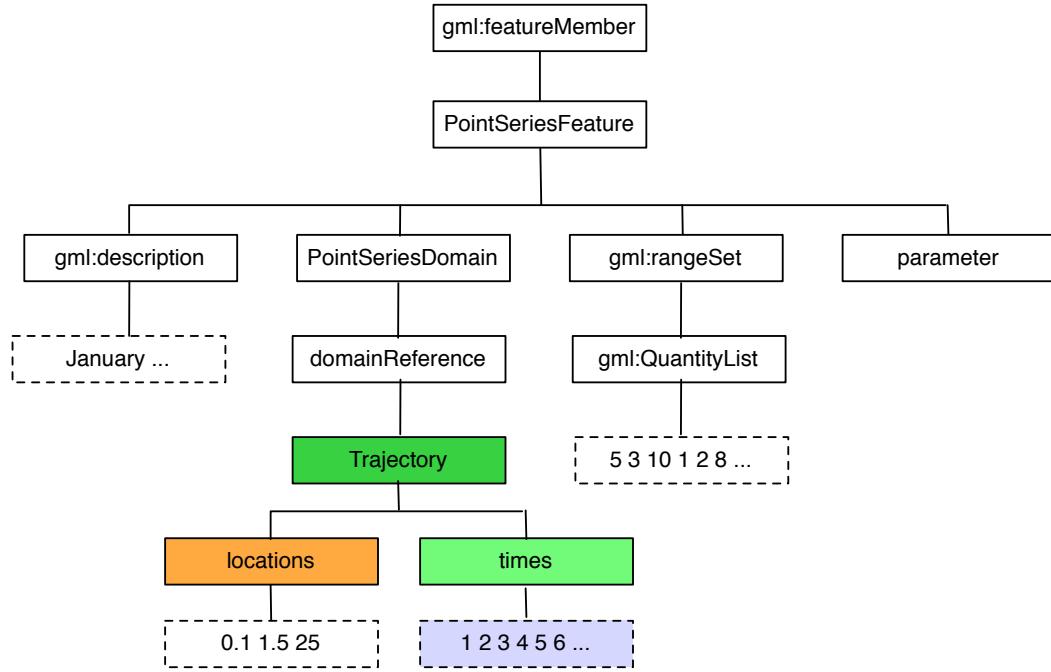


Figure 11.1: Example of a CSML Document Tree. *This tree provides a conceptual model for the organization of data found in the CSML document in Q.11-1 (page 403). Each element is represented by a node in the tree and the branches from one node to another show the connections between the nodes. The <times> element (shown in light green) is a child of the <Trajectory> element (dark green). That is, <times> is nested directly under <Trajectory>, and the branch between these nodes in the graph indicates this connection. Additionally, the <locations> node (in orange) is a sibling to <times>. Text nodes are part of the hierarchy and are displayed via nodes with dashed borders, e.g., the purple node is a text child of <times>.*

The *root* of this tree, also referred to as the *document node*, is `<gml:featureMember>`. As mentioned earlier, there is only one root per document. In this case, the document node has only one child, `<PointSeriesFeature>`, which has four children (grandchildren of the document node)—one each of `<gml:description>`, `<PointSeriesDomain>`, `<gml:rangeSet>`, and `<parameter>`.

We use family tree terminology to describe the relative position of nodes in the tree. For example, consider the `<times>` node shown in green in the figure. Its *parent* is `<Trajectory>` (shown in dark green), and reciprocally, `<times>` is a *child* of `<Trajectory>`. The `<locations>` node (shown in orange) is a *sibling* to `<times>`. Also, `<Trajectory>`, `<domainReference>`, `<PointSeriesDomain>`, `<PointSeriesFeature>`, and `<gml:featureMember>` are all *ancestors* of `<times>`.

With all trees, the document node is the ancestor of all other nodes, and all other nodes are said to be *descendants* of this node. Additionally, the character content of an element is placed in a “text” node in the tree. In our example tree, the text `1 2 3 4 5 6 ...` shown in purple is a child node of `<times>`. The *terminal* nodes in a tree at the end of the

branches, which have no children, are known as *leaf nodes*. By design, any text content will always be a leaf node because text cannot contain an element. Of course, an element with no content will be a terminal or leaf node, e.g., `<r/>` or `<Cube currency="USD" rate="1.5872"/>`.

The examples that we examined in the previous section take quite different approaches to organizing data. In the CSML example, the numeric data values are provided in “batches” in the text content (i.e., as a single text node with the values separated by white space, e.g., “5 3 10”), and metadata, such as the units of measurement, the names of the variables, etc., are provided through additional tags and attributes. Alternatively, the exchange rate data values are provided through element attribute values, with one datum per attribute. (See Figure 11.2.) The `<Cube>` element is used for multiple purposes, and it is through the attribute names that the type of information, or dimension of the data cube being provided, is ascertained. Note also that the eurofxref document is four nodes deep, in comparison to the CSML tree which is seven nodes deep. This is partly due to the use of attributes rather than child nodes to represent the data and to the additional `<PointSeriesFeature>` element under the root of the tree.

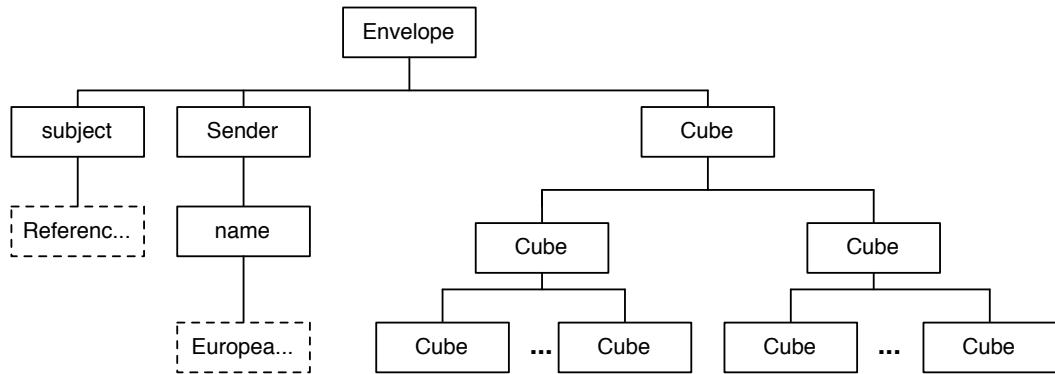


Figure 11.2: Example of an SDMX Document Tree. *The tree shown here is a representation of the hierarchy of the SDMX document in Q.11-2 (page 405). Each element is a node in the tree, with text content represented by a dashed border on the node. The exchange rates, dates, and type of currency are found in the attribute values of the nodes, which are not shown in the diagram. These omissions are made merely for simplicity in the presentation of the tree.*

Whether to use attributes or child nodes to represent data values is a matter of style. In general, attributes are often used to provide information about the data. There are no required rules about when to use attributes and when to use elements. However, if we have multiple data values, we cannot use the same attribute for each one separately. Instead, to use an attribute, we have to combine them into a single string, separated by some delimiter, e.g., “5,3,10”. Here it is often easier to separate child nodes, e.g.,

```

<value>5</value>
<value>3</value>
<value>10</value>
  
```

Similarly, if the data values are not simple primitive values, but instead have some complex structure, we cannot easily use attributes. A simple rule-of-thumb is that it is best to use attributes for information about the data, i.e., metadata, rather than the actual data values. For example, when we refer to an R function in our extended

DocBook vocabulary, we add the name of the package as an attribute, e.g., `<r:func pkg="XML">xmlParse</r:func>`. Of course, the distinction between metadata and data is not always clear.

11.3 XML Namespaces and Additional XML Elements

While the element or node is the primary component in *XML*, the language also includes several other, somewhat less common, but useful constructs. These are the *XML* declaration, processing instructions, comments, `<CDATA>` section delimiters, entity references, and document-type declarations. In this section, we briefly describe each of these.

Additional XML Markup

CDATA Character data that is treated by the *XML* parser as literal text and not *XML* content. This is used to “escape” content that happens to contain special *XML* characters such as `<` and `&`, and treat it as verbatim text. Markup between the delimiters `<! [CDATA [and]]>` is not processed, e.g.,

```
<! [CDATA[ x <- y > 10 & z < 20 ] ]>
```

Comment A block of text that is not considered part of the data itself, but informal information about the document. A comment appears between the delimiters `<!--` and `-->`, e.g.,

```
<!-- This is a comment. -->
```

XML Declaration Optional statement at the beginning of a document that identifies the content as an *XML* document and provides additional information about the character encoding and the version of *XML*, e.g.,

```
<?xml version = "1.0" encoding="UTF-8" ?>
```

Processing Instruction Optional hint or instructions to an application that might parse the *XML* document. It is a way for us to include directives to a target application so that it will do something as it processes the *XML* nodes, such as use a particular style sheet to render the document. A processing instruction has two parts—the name of the target application and the instruction. The target application is a simple string, while the instruction can be any sequence of text and is entirely application-specific. For example, we can set the width option in *R* with

```
<?R options(width = 140) ?>
```

Other applications reading the *XML* will ignore this. One can include the same information within regular *XML* elements, but processing instructions are a powerful and convenient way to specify application-specific information without adding to the *XML* vocabulary used for the data.

Document-type Declaration A declaration at the start of the document (before the root node) that identifies the “type” of the document, e.g.,

```
<!DOCTYPE html>
```

The type can either be one of the known types such as `html` or `xhtml`, or can also specify the location of an external *DTD* (Document Type Definition) document that describes the structure of a valid document of this type. Some or all of the *DTD* can also be inlined within the `<DOCTYPE>` node, and extensions and character entities can be added.

Applications often build on or extend other *XML* grammars, adding their own tag names to those of one or more well-established standards. For example, *RDocBook* extends the *DocBook* vocabulary with element names for *R* expressions, function and package names, code, etc. As another example, Q.11-1 (page 403) showed an *XML* document with GML and CSML vocabularies. This flexibility is one of the strengths of *XML*. Occasionally, two vocabularies use the same name to mean different things, and when this happens, *namespaces* [1] provide a mechanism to avoid conflicts. A namespace prefix qualifies an element name and connects it to a particular vocabulary (i.e., namespace). Q.11-2 (page 405) included element names from two vocabularies, the GESMES vocabulary and the eurofxref namespace. In that example, we ignored these namespaces for simplicity. However, the actual *XML* appears as:

```
<gesmes:Envelope xmlns:gesmes="http://www.gesmes.org/xml/2002-08-01"
  xmlns="http://www.ecb.int/vocabulary/2002-08-01/eurofxref">
  <gesmes:subject>Reference rates</gesmes:subject>
  <gesmes:Sender>
    <gesmes:name>European Central Bank</gesmes:name>
  </gesmes:Sender>
  <Cube>
    <Cube time="2008-04-21">
      <Cube currency="USD" rate="1.5898"/>
      <Cube currency="JPY" rate="164.43"/>
      <Cube currency="BGN" rate="1.9558"/>
      <Cube currency="CZK" rate="25.091"/>
    </Cube>
    <Cube time="2008-04-17">
      <Cube currency="USD" rate="1.5872"/>
      <Cube currency="JPY" rate="162.74"/>
      <Cube currency="BGN" rate="1.9558"/>
      <Cube currency="CZK" rate="24.975"/>
    </Cube>
  </Cube>
</gesmes:Envelope>
```

A namespace is associated with a prefix via a namespace definition, which appears in the `xmlns` “attribute” of an element. (Technically, `xmlns` is not an attribute, but the syntax for specifying it is the same as for attributes.) The `Envelope` element defines the two namespaces. The prefix for the GESMES namespace is `gesmes` and the `<Envelope>` element uses this prefix to indicate it belongs to that namespace, i.e., `<gesmes:Envelope>`. Additionally, any descendant of `<gesmes:Envelope>` that has a `gesmes` prefix is associated with the GESMES namespace. The `eurofxref` grammar is also defined in `<gesmes:Envelope>`,

but a prefix is not provided. In this case, the eurofxref namespace is treated as the default. That is, any child element of `<gesmes:Envelope>` that does not have a prefix belongs to the euroxref namespace. This convention can be very convenient because it saves us from having to add prefixes to node names.

In general, the syntax for adding a namespace definition to an element is:

```
xmlns:prefix="URI"
```

The URI in the namespace definition is used to identify the vocabulary; it is not used to look up information about the vocabulary. However in practice, companies often use it as a pointer to a Web page containing information about the grammar.

11.4 The Document Object Model (*DOM*)

The XML package provides several approaches for extracting data from *XML* documents. In this section, we focus on one approach called Document Object Model (*DOM*) parsing [7, 8]. This is perhaps the most natural approach for many users as the document is represented as a tree (see Section 11.2). We demonstrate how to extract information from a document by using the tree hierarchy to access the content of interest. We introduce this type of parsing first because it is the simplest and the cornerstone for several other parsing models.

Recall that with the tree, each node corresponds to an *XML* element and the hierarchy represents the relationships between elements. Our *DOM* parser returns a copy of the *XML* document in this hierarchical form, which makes it reasonably easy to operate on elements in the document to convert their contents or extract relevant information. The *DOM* also enables us to use *XPath* to locate nodes and attributes.

Let's get started with parsing the *XML* document into a tree-structure and demonstrating the correspondence between the various *XML* elements and their representation in *R*. The `xmlParse()` function in the XML package is a *DOM* parser, meaning it reads a document into a structure that represents the hierarchical structure of the document. While `xmlParse()` has 19 arguments, the only one that is required is `file`, which we can use for specifying the location of the *XML* document, be it a local file, *URL*, or a string containing the document itself. Simple calls of the form

```
doc = xmlParse("FargoDailyWeather.xml")
```

will yield a tree object in *R* for accessing the contents of the *XML* document. When we show/print such objects in *R*, they appear in the *R* console as the existing *XML* document, suitably indented, etc.

To demonstrate how to parse an *XML* document, we provide a simple example from the US Geological Survey (USGS) catalog of earthquakes that occurred in a one-week period [12]. The USGS provides real-time, worldwide earthquake data in a variety of formats, including two *XML* vocabularies, which are available at <http://earthquake.usgs.gov/eqcenter/catalogs/>. Below is a snippet of our *XML* file that shows the structure of the information for one event/quake within the document.

```
<?xml version="1.0" encoding="UTF-8"?>
<merge>
  <event id="00068404" network-code="ak"
        time-stamp="2008/09/16_22:17:31" version="2">
    <param name="year" value="2008"/>
```

```

<param name="month" value="09"/>
<param name="day" value="14"/>
<param name="hour" value="00"/>
<param name="minute" value="59"/>
<param name="second" value="04.0"/>
<param name="latitude" value="51.8106"/>
<param name="longitude" value="-175.9250"/>
<param name="depth" value="146.0"/>
<param name="magnitude" value="3.8"/>
<param name="num-stations" value="10"/>
<param name="num-phases" value="15"/>
<param name="dist-first-station" value="126.1"/>
<param name="azimuthal-gap" value="53"/>
<param name="magnitude-type" value="L"/>
<param name="magnitude-type-ext"
      value="Ml = local magnitude (synthetic Wood-Anderson)"/>
<param name="location-method" value="a"/>
<param name="location-method-ext"
      value="Auryn (Confirmed by human review)"/>
</event>
...

```

Notice that the root node of the document is `<merge>`, and information for each earthquake/episode is contained in an `<event>` child of `<merge>`. The details for an earthquake are in `<param>` nodes in the earthquake's `<event>` element. Each `<param>` has a `name` and `value` attribute, creating a structure that is parallel to the name–value pair type of format used in many plain text files.

We can read this *XML* document into *R* with

```
doc = xmlParse("merged_catalog.xml.gz")
```

The `xmlParse()` function uses the *XML* parsing library `libxml2` [13] to build an *XML* tree as a native/*C*-level data structure. This data structure uses *C* pointers (similar to references) to connect nodes as children, parents, and siblings. As such, it is easy to navigate from a node to its children, or to its parent or ancestor, even to the top-level node in the document. It is harder to do this with the *R* data structures. For example, traversal up the tree is not possible with the list of lists returned from `xmlToList()`. By leaving the document in a *C* data structure, we are able to use *XPath* to traverse the tree in any direction.

The `xmlParse()` function returns an object of class, `XMLInternalDocument`; it has a field for the name of the file containing the *XML*, which we access with `docName()`, and a pointer to the root node of the document, which we access with `xmlRoot()`, e.g.,

```
root = xmlRoot(doc)
```

The `xmlRoot()` function returns an object of class `XMLInternalElementNode`. This is a regular *XML* node and not specific to the root node, i.e., all *XML* nodes will appear in *R* with this class or a more specific class. An object of class `XMLInternalElementNode` has four fields: `name`, `attributes`, `children` and `value`, which we access with the methods `xmlName()`, `xmlAttrs()`, `xmlChildren()`, and `xmlValue()`, respectively. For example, we can confirm that the root element is named “merge” with

```
xmlName(root)
```

```
[1] "merge"
```

Additionally, we can calculate the number of children of `<merge>` by passing `root` to `xmlSize()`:

```
xmlSize(root)
```

```
[1] 1047
```

This shows that there are 1047 child nodes, which matches the length of the list returned from

```
length(xmlChildren(root))
```

The `length()` function is not overloaded to give `xmlSize()`; that is, `length(root)` returns 1.

In the next section, we demonstrate how to use these and other methods to navigate the tree and access specific content.

11.5 Accessing Nodes in the DOM

The XML package provides many generic functions that can extract element names, attribute values, child nodes, and text content from nodes in the tree. In addition, the `[[` and `]` operators allows us to treat an *XML* node as a list of its child nodes. Similar to subsetting lists, we can use `[` and `[[` to access child nodes by positions, names, a logical vector, or exclusion. When we subset by “name”, we use the node’s element name. For example, we can extract the first `<event>` node from `<merge>` in our USGS document with

```
event1 = root[["event"]]
event1

<event id="00068404" network-code="ak"
       time-stamp="2008/09/16_22:17:31" version="2">
  <param name="year" value="2008"/>
  <param name="month" value="09"/>
  <param name="day" value="14"/>
  ...
</event>
```

We can retrieve, say, the tenth child of the first `<event>` with

```
event1[[10]]

<param name="magnitude" value="3.8"/>
```

We see that this child holds the magnitude of the quake. As another example, we can get the first seven `<event>` nodes with

```
root[1:7]
```

Similarly, we can get all but the first seven children of the root node with

```
root[ -(1:7) ]
```

Finally, we note that when we subset by name using the single square bracket, e.g.,

```
evs = root["event"]
```

then we extract all `<event>` nodes from the `<merge>` node. A call to `length()` confirms this:

```
length(evs)
```

```
[1] 1047
```

This is different from regular *R* lists but more convenient than

```
root[ names(root) == "event" ]
```

The `evs` object is of class `XMLInternalNodeList`, which is essentially a list of `XMLInternalElementNode` objects. This means that we can apply the methods `xmlName()`, `xmlValue()`, etc. to the elements in `evs`, e.g., we find that the first `<event>` node has 18 children with

```
xmlSize(evs[[1]])
```

```
[1] 18
```

We consider the node as a named list where the names are the node names of the child elements of that node. To see the names of the children of `event1`, i.e., the first `<event>` node, we use

```
names(event1)
```

```
param    param    param    param    param    param    ...
"param" "param" "param" "param" "param" "param"
```

We obtain the name of the node itself with `xmlName()`, i.e.,

```
xmlName(event1)
```

```
[1] "event"
```

In the following example, we examine Kiva data and demonstrate how to use `[]` to extract the `<template_id>` from the images in the `<lender>` nodes.

Example 11-3 Retrieving Content from Subnodes in a Kiva Document

We can create, `lendersNode`, as a “list” of 1000 lenders with

```
doc = xmlParse("kiva_lender.xml")
lendersNode = xmlRoot(doc)[["lenders"]]
```

We can confirm that there are 1000 children in `<lenders>` with

```
xmlSize(lendersNode)
```

```
[1] 1000
```

Before getting the `<template_id>` from the `<image>` node in each `<lender>` node, we demonstrate some additional approaches to subsetting. Within a `<lender>` node, we can retrieve by name the `<name>`, `<occupation>`, and `<image>` nodes of the first lender with

```
lendersNode[[1]] [ c("name", "occupation", "image") ]  
  
$name  
<name>Matt</name>  
  
$occupation  
<occupation>Entrepreneur</occupation>  
  
$image  
<image>  
  <id>12829</id>  
  <template_id>1</template_id>  
</image>  
  
attr("class")  
[1] "XMLInternalNodeList" "XMLNodeList"
```

Again, `lendersNode[[1]]` gives the first `<lender>` node and

```
lendersNode[[1]] [ c("name", "occupation", "image") ]
```

returns a list of the three nodes named name, occupation, and image. If there are, e.g., multiple `<name>` nodes in that `<lender>` element, then all of them are returned.

Suppose we want the children of `lendersNode[[1]]` that have more than one child. In this example, that is only the `<image>` node. We can determine which child nodes satisfy this constraint with

```
w = sapply(xmlChildren(lendersNode[[1]]), xmlSize) > 1
```

We can use this logical vector to get the subset

```
lendersNode[[1]] [ w ]  
  
$image  
<image>  
  <id>12829</id>  
  <template_id>1</template_id>  
</image>  
  
attr("class")  
[1] "XMLInternalNodeList" "XMLNodeList"
```

Finally, we retrieve the content of the `<template_id>` subnodes within `<image>` with

```
template_id = sapply(xmlChildren(lendersNode),  
                     function(x)  
                       xmlValue(x[["image"]][["template_id"]]))
```

Looping over *all* child nodes of an XML node is very common and so we have provided a slightly simpler idiom than

```
sapply(xmlChildren(parentNode), function(node) ...)
```

The functions `xmlApply()` and `xmlSApply()` take the node and the function and do the looping for us. (The “`sapply`” version attempts to simplify the result; `xmlApply()` does not.) We can, for example, obtain the number of children in each of the child nodes of the first `<lender>` node (from Q.11-3 (page 415)) with

```
xmlSApply(lendersNode[[1]], xmlSize)
```

That is, `xmlSApply()` loops over the children of the node provided and applies the supplied function to each child node, i.e.,

```
xmlSApply(parentNode, function(childNode) ...)
```

This is marginally simpler. If you want to iterate over a subset of the child nodes then you have to use `xmlChildren()` and subsetting, and then use `sapply()` or `lapply()` to iterate over the list of nodes.

In the following example, we will demonstrate how to use `xmlSApply()` to extract the time-stamp and magnitude for each earthquake in the USGS document introduced in Section 11.4.

Example 11-4 Retrieving Attribute Values from a USGS Earthquake Document

In Section 11.4, we saw that the data for each earthquake appears in an `<event>` node. The time of the quake is available in the `time-stamp` attribute on the `<event>` element. The magnitude appears in a `<param>` child of `<event>`. The `<event>` has several `<param>` children, and these are differentiated by their `name` attributes. That is, the magnitude of the quake is found in the `<param>` node that has a value of "magnitude" for its `name` attribute. The actual measurement for the magnitude is in the `value` attribute for this `<param>` node.

We first retrieve the time-stamp for each `<event>` node. We can use the `xmlGetAttr()` function to do this. This function takes as input a node and the name of an attribute, and it returns the value of the attribute on that node. We use `xmlSApply()` to apply `xmlGetAttr()` to each child of the root node, i.e., to each `<event>` node, with

```
timestamps = xmlSApply(root, xmlGetAttr, "time-stamp")
timestamps[1047]

           event
"2008/09/21_00:46:57 "
```

This apply function iterates over the input node’s children, invoking `xmlGetAttr()` and returning the vector of time-stamps. We can pass additional arguments for each function call via the `...` argument of `xmlSApply()` as we can for `lapply()` and `sapply()`. Here we passed "time-stamp" for the value of the `name` argument to `xmlGetAttr()`.

Now that we have obtained the time-stamps, we turn to the task of getting the magnitudes of the quakes. We provide a few alternative approaches that increase in sophistication and in generality. To start, we notice that the magnitude of the first `<event>` appears in its tenth child :

```
root[[1]][[10]]
```

```
<param name="magnitude" value="3.8"/>
```

For our first approach, we extract the tenth element of each `<event>`. Then from each of these `<param>` nodes, we extract the content of its `value` attribute:

```
child10 = xmlSApply(root, "[[", 10)
mags = as.numeric(sapply(child10, xmlGetAttr, "value"))
head(mags)

event event event event event
 3.8   1.9   1.1   1.2   0.6   1.3
```

The `xmlGetAttr()` function accepts a `converter` parameter, which is useful for coercing individual values. However, in this case, it is more efficient to get the attribute values of all the nodes and then convert this character vector to numeric.

An alternative approach is to combine the extraction of the children and the application of the `xmlGetAttr()` function in one step with

```
mags = as.numeric(xmlSApply(root,
                           function(node)
                             xmlGetAttr(node[[10]], "value")))
```

Of course, we have assumed in both of these approaches that the tenth child of each `<event>` node will always hold the magnitude. When we check the number of children in the first few events,

```
xmlSApply(root, xmlSize) [1:4]

event event event event
 18    18    18    22
```

we see that these nodes have different numbers of children. Relying on the tenth child of each `<event>` to contain the magnitude may be problematic. A more general approach that does not depend on a particular ordering of the `<param>` nodes is preferable.

A more robust approach is to get the value of the `name` attribute on each of the `<param>` nodes in an `<event>` node and determine which is the pertinent `<param>`. Then, we know that we are getting the magnitude value, rather than whatever might be in the tenth `<param>`. We can accomplish this using nested calls to `xmlSApply()` with

```
xmlSApply(root, function(evNode) {
  parNames = xmlSApply(evNode, xmlGetAttr, "name")
  i = which(parNames == 'magnitude')
  xmlGetAttr(evNode[[i]], "value")
})
```

In Section 11.6, we introduce *XPath* which we can use to locate the magnitudes more simply. Essentially, with an *XPath* expression, we can specify that we want the information in the `value` attribute of those `<param>` tags that have a `name` of "magnitude". For completeness and to demonstrate the potential of *XPath*, we provide the code for this approach. We use the `getNodeSet()` function, which takes a node and an *XPath* expression and uses this expression to navigate the DOM. We do this with

```
getNodeSet(root, "/merge/event/
  param[@name='magnitude']/@value")
```

The return value is a list of the desired attribute values, which we can unlist and convert to numeric.

This more elegant approach demonstrates the power of *XPath* expressions to locate nodes, attributes, and content. *XPath* provides a convenient fast way to find nodes with particular characteristics anywhere in the tree. Often all that is needed is to identify a standard characteristic, such as all nodes with a particular name or all nodes with a particular attribute value. Such types of criteria can be combined into rich expressions that identify subsets of the nodes. These expressions are available in the *XPath* language and are discussed in greater detail in Section 11.6. Often, we use *XPath* to obtain the nodes and then operate on them using the *R* functions we have explored in this section. ■

In the previous example, we saw how to access nodes in a *DOM* object with functions for going down the tree hierarchy, i.e., `[`, `[[` and `xmlChildren()`. In addition, we also have functions for traversing up the tree and across siblings. The `getSibling()` function retrieves a sibling of a node, either the sibling to the left (i.e., above) or to the right (below) of the node. Whereas, `xmlParent()` gives us access to the parent of a node. For example, suppose `firstMag` holds the `<param>` element with the magnitude information for the first quake in the catalog, i.e.,

```
firstMag = root[[1]][[10]]
```

Then, we can access that `<param>` node's sibling to the left with

```
getSibling(firstMag, after = FALSE)
<param name="depth" value="146.0"/>
```

The default value for the `after` parameter in `getSibling()` is TRUE, which returns the sibling node that follows the provided node. The `xmlParent()` function gives us access to the `<event>` node in which this `<param>` is located. For example, we retrieve the attributes on this parent with

```
xmlAttrs(xmlParent(firstMag))
```

	<code>id</code>	<code>network-code</code>	<code>time-stamp</code>	<code>version</code>
	"00068404"	"ak"	"2008/09/16_22:17:31"	"2"

Notice that the `XMLInternalElementNode` in `firstMag` has not lost access to the parent even though we assigned this node to a new variable. This is because the assignment has not made a copy of the node. As mentioned earlier, the parsed document consists of pointers to a *C*-level data structure. When we assigned `evs[[1]][[10]]` to `firstMag`, we did not get a copy of the `<param>` node. Instead, we have a reference to that point in the tree. Any changes made to `firstMag` will be reflected in the parsed document.

The `xmlChildren()`, `xmlParent()`, `getSibling()` and `xmlAncestors()` functions use these pointers to navigate to any node in the tree, e.g., we can traverse from `firstMag` up the tree to the root with

```
xmlName(xmlAncestors(firstMag)[[1]])
[1] "merge"
```

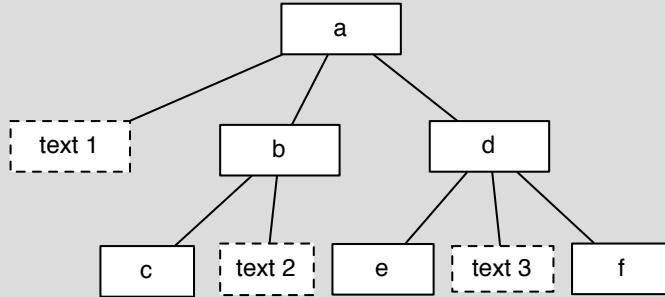
The `xmlAncestors()` function walks the chain of parents to the top of the document and returns a list of those ancestor nodes. The first will be the root node. Also, from a parent or sibling of a parent we can navigate down the tree, e.g.,

```
getSibling(xmlParent(firstMag))[[10]]
<param name="magnitude" value="1.9"/>
```

We have retrieved a “cousin” of `firstMag`. With the `XMLInternalDocument` object, it is easy to traverse from a node to its children, its parent, the top-level document containing the node, any ancestor, sibling, and so on.

The DOM Parser in R

The `xmlParse()` function in R by default returns a tree structure which can be treated conceptually as a list of lists, i.e., each node in the tree can be treated as a list of its children. Each node also has a name and attributes. The following functions in the XML package are available for accessing nodes and their content. We exemplify how to use them with the simple document shown below, where dashed boxes correspond to text content.



- Read the XML document into R with

```
doc = xmlParse("simpleEx.xml")
```

- Access the root node via

```
root = xmlRoot(doc)
```

- Operate on a node as if it is a list of its children, i.e., use `[` and `[[`] to access elements in the tree. For example, we access the `<e>` node in the document with either of the following:

```
node3_1 = root[[3]][[1]]
node3_1 = root[["d"]][["e"]]
```

- The XML package provides functions for determining information about a node. These include `xmlName()`, `xmlSize()`, `xmlAttrs()`, `xmlGetAttr()`, `xmlValue()`, `xmlNamespace()`, and `getDefaultNamespace()`, which provide, in order, the node’s name, number of children, attributes, a specific attribute, text content of the node and its descendants, namespace, and default namespace. For example,

```
xmlValue(root[["b"]])
```

```
"text 2"
```

and `xmlSize(node3_1)` returns 0.

- In addition to `[` and `[[`, other functions in XML enable us to work with a node's siblings, children, parent, and ancestors. These are `getSibling()`, `xmlChildren()`, `xmlParent()`, and `xmlAncestors()`, respectively. For example, we retrieve the parent of `node3_1` with

```
xmlParent(node3_1)
```

and the sibling following `node3_1` with

```
getSibling(node3_1)
```

With these functions, we can traverse the tree from one node to another anywhere in the tree. For example, from `node3_1` we access the first child of `<a>` with either of these expressions:

```
xmlParent(xmlParent(node3_1))[[1]]
getSibling(getSibling(xmlParent(node3_1), after = FALSE),
           after = FALSE)
```

- The tree object behaves differently from regular *R* objects. When we make the assignment, `node3_1 = root[[3]][[1]]`, we now have a reference to that point in the tree. Any operations on `node3_1` will be made to the tree as well. For example, `xmlParent(xmlParent(node3_1))` references the root of the parsed document, i.e., `root`.

11.6 XPath and the XML Tree

XPath is a language for querying and locating elements in an *XML* document. It operates on the hierarchy of a well-formed *XML* document to specify the desired chunks to obtain. *XPath* is *not* an *XML* vocabulary; it has a syntax that is similar to but more powerful than the way files are located in a hierarchy of directories in a computer file system. Anyone familiar with navigating file-system trees, either with command line utilities in *UNIX* such as `ls` for listing directories and `cd` for changing directory, or with a graphical user interface such as Mac OS X's Finder or Microsoft's Explorer, will find similarities to *XPath* expressions. *XPath*, however, is much more succinct and expressive.

XPath has many similarities to regular expressions. In both cases, we are identifying patterns to match data or content. There is a trade-off between matching too liberally/permissively and being overly specific. We often mix the pattern matching with subsequent *R* computations on the resulting matches. Like regular expressions, experience helps compose correct *XPath* expressions. However, *XPath* is a simpler language and has a simpler computational model to understand than regular expressions.

Let's consider the *XML* document from Q.11-2 (page 405) that contains the currency exchange rates relative to the euro. The basic structure of the document (with the namespaces removed for simplicity) is shown below.

```
<Envelope>
  <subject>Reference rates</subject>
  <Sender>
```

```

<name>European Central Bank</name>
</Sender>
<Cube>
  <Cube time="2008-04-21">
    <Cube currency="USD" rate="1.5898"/>
    <Cube currency="JPY" rate="164.43"/>
    <Cube currency="BGN" rate="1.9558"/>
    <Cube currency="CZK" rate="25.091"/>
  </Cube>
  <Cube time="2008-04-17">
    <Cube currency="USD" rate="1.5872"/>
    <Cube currency="JPY" rate="162.74"/>
    <Cube currency="BGN" rate="1.9558"/>
    <Cube currency="CZK" rate="24.975"/>
  </Cube>
</Cube>
</Envelope>

```

This snippet of an *XML* document is in the SDMX format. See Q.11-2 (page 405) for more details about this particular *XML* vocabulary. The exchange rates for each currency are in *<Cube>* nodes with the currency and that day's exchange rate given as attributes. The currencies are grouped together in a parent node for each day. This parent is also named *<Cube>* and it has a *time* attribute. To further confuse matters, the collection of daily data are organized as elements of yet another *<Cube>* node. The *<Cube>* is a general way to represent multidimensional data. Here we have three dimensions. There is the overarching *<Cube>* to indicate what it is being measured (exchange rates) and within this the different days and within day the different currency values.

The following *XPath* expression,

```
/Envelope/Sender/name
```

locates the *<name>* element near the top of the document. The document hierarchy shown in Figure 11.3 shows the realization of this *XPath* expression, i.e., the nodes that have been identified by the query. We can evaluate the query in *R* with

```
nm = getNodeSet(doc, "/Envelope/Sender/name")
```

An *XPath* expression defines a *location path* consisting of one or more *location steps*, each separated by a forward slash. In this expression, we start at the root (/) and look for a child element named *<Envelope>*. Having found that, we continue with the next step in the search, and from this position we look for a child node named *<Sender>*. Finally, we start from this *<Sender>* node and search for a child called *<name>*. Here the steps are very specific, but we will see that they can be much more general, e.g., any descendant at any level.

The *XPath* computational model is designed to identify *node-sets*, which are collections of nodes in the target tree that meet the criteria in the *XPath* expression. The result of our query in *R* is of class *XMLNodeSet* and is a list of references to those nodes which the *XPath* query matched. This is a set in the sense that there are no repeated elements, i.e., each node in the result is unique within this result. In this case, it contains a single *<name>* element from the document, but in general the expression can match more than one node. Indeed, in our example, there was only one matching node *at each location step*. However, there might be many, and *XPath* follows all matching nodes at each step. In this way, it is vectorized in its searching.

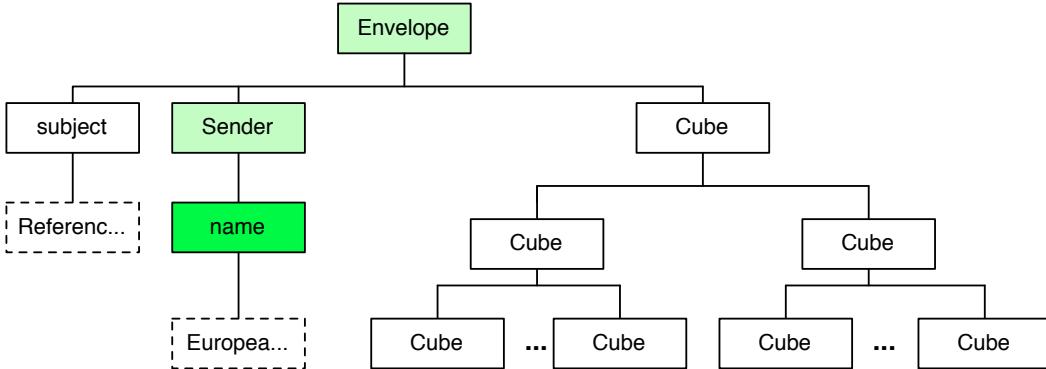


Figure 11.3: Simple *XPath* Expression Applied to a Tree. The shading in this diagram shows how the *XPath* expression, /Envelope/Sender/name locates the <name> node. The shaded nodes are location steps in the path to the matching node, which are progressively more brightly shaded as we move from one location step to the next and get more specific in the query.

1. The first location step identifies the root node, `<Envelope>`.
 2. The next step locates the `<Sender>` child of `<Envelope>`.
 3. The third step identifies `<Sender>`'s child called `<name>`.

For an example that matches multiple nodes, consider the *XPath* expression

/Envelope/Cube/Cube

This expression matches two nodes as shown in Figure 11.4, corresponding to the two days of data in our document. These matches are two sibling `<Cube>` nodes that are grandchildren of `<Envelope>`, and children of the topmost `<Cube>` node.

The notion of a node-set, i.e., where a node can occur just once in the set, may seem problematic. A node may match multiple conditions in a composite *XPath* expression, i.e., where we specify two or more node tests in the *XPath* query. In contrast, when we subset the same element multiple times in an *R* vector, we explicitly obtain multiple copies of the element. But the concept of a node-set with each element occurring at most once is precisely what makes *XPath* useful. We can find all nodes in a tree that match a particular query and then work with just those. We do not have to worry whether we have already processed that node earlier in the node-set since we know it is unique. We are also guaranteed that the nodes will appear in the node-set in the same order that they occur in the document, i.e., in document order. If we did two separate *XPath* queries, we would end up with two node-sets and they might contain some of the same nodes. If we were to process these two node sets, we might end up “double-counting” a node. Also, we would not know if we were processing the nodes in an appropriate order. The node-set is precisely what we want. For example, to extract all exchange rates for the Japanese yen from the SDMX data, we might look for all the `<Cube>` elements with a `currency` attribute value of JPY. The *XPath* expression

```
//Cube[@currency = "JPY"]
```

does just that. The expression //Cube is a shortcut for specifying the location step that indicates the <Cube> element may appear at any level in the document. It means: match

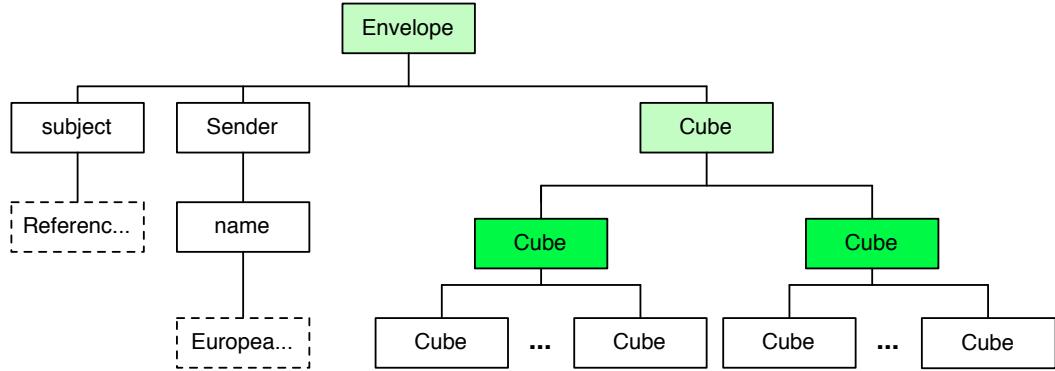


Figure 11.4: *XPath Expression Locating Multiple Nodes in a Tree*. The shading in the diagram shows how the *XPath* expression `/Envelope/Cube/Cube` locates two sibling `<Cube>` nodes. The lightly shaded nodes denote steps in the path toward the match of the two nodes that are brightly shaded in the diagram.

1. The first location step identifies the root node, `<Envelope>`.
2. The next step locates the `<Cube>` child of `<Envelope>`.
3. The third step identifies the two `<Cube>` children of the second-level `<Cube>` node matched from the second step.

all of the descendants, including the current node, by name. Having obtained the resultant matches, we apply a predicate to restrict the set. The square brackets are analogous to subsetting in *R* and provide a test on the nodes that have matched. That is, the expression within `[]` provides a condition that must be met by these matching `<Cube>` nodes, if they are to remain in the node-set. The expression `//Cube` matches all `<Cube>` nodes, but the condition `[@currency = "JPY"]` filters out those nodes that do not have a *currency* attribute or whose *currency* attribute does not have the value `"JPY"`. The `@` symbol is shorthand for attribute in *XPath*. Two nodes match our expression; these are the two nodes with an exchange rate for the yen, as shown in Figure 11.5.

We can also use *XPath* to find nodes based on their currency value. For example, we might want to locate the rates for the Czech koruna when the exchange rate was less than 25. We can find this with the *XPath* query

```
//Cube[@currency = 'CZK' and @rate < 25]
```

This combines two tests (currency value and rate) on each `<Cube>` node, filtering out those that do not match.

These four examples demonstrate the power of the *XPath* language for specifying content in an *XML* document. In addition to the matching capability in these simple expressions, *XPath* allows the specification of complex relationships in a document's hierarchy. The notion of the parent, ancestor and sibling to a node can be part of the expression, which means that the steps in the location path can travel up and to the side in the hierarchy, i.e., not just down the tree. For example, if we want to find the dates when the Czech koruna dropped below 25, then we need the *time* attribute on the parent `<Cube>` nodes that we have located. We can get this by backing up one level in the hierarchy with the following *XPath* expression

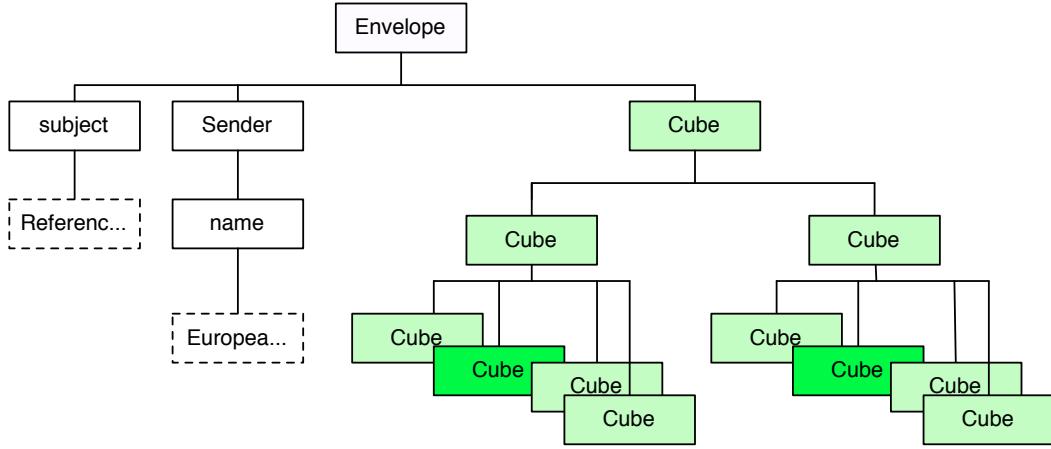


Figure 11.5: XPath Predicate Filters a Nodeset. The XPath expression, `//Cube`, matches all `Cube` nodes anywhere in the document. The `//` is shorthand in XPath for all descendant nodes from this point down, including this “self” node. The addition of the predicate `[@currency="JPY"]` filters the set of matches to those elements that have a currency attribute with a value of “JPY”. In this case, two nodes meet this condition. The shaded nodes highlight all the `<Cube>` elements in the document, and the two darkest satisfy the predicate.

```
//Cube[@currency = 'CZK' and @rate < 25]/../@time
```

XPath also includes functions that we can use to perform computations on a node. For example, the `starts-with()` function allows us to compare the beginning of a node’s content or name with a string, and `last()` determines if the node is the last matching element in the current node-set. We will explore these functions after we examine the formal details of the XPath syntax and computational model.

11.7 XPath Syntax

The examples of the previous section have informally introduced XPath expressions, but to put the full power of the language to use we need a more formal description of its syntax and computational model. An XPath expression is made up of one or more *location steps*, separated by a `/` character. An individual location step has three distinct parts, the *axis*, *node-test*, and an optional *predicate*, which we specify in the following format:

```
axis::node-test [predicate]
```

(The shorthand forms we have used in earlier examples in this chapter deviate from this syntax, but all XPath steps can be written in this form.) The axis is separated from the node-test by two colons, (e.g., `::`). This allows us to use a namespace prefix in the node-test, e.g., `descendant::r:code`. The predicate, if present, is specified within square brackets, e.g., `child::Cube[@currency = 'JPY']`.

We specify an entire location path by separating the location steps with the forward slash (`/`), e.g.,

```
/child::Envelope/child::Cube
```

The *XPath* engine breaks the path into location steps, ignoring extraneous white space, and then evaluates each step in turn. At the end of each step, there is a set of matching nodes. Each of these are used as the current location when evaluating the next location step. The axis indicates which direction to search and how. It might be to look at the child nodes, or alternatively all descendants, or the siblings to the right, all relative to the current starting point for this step. Given the direction to search, *XPath* applies the node-test. This typically matches the name of a node to the specified name, e.g., Cube. However, the node-test can also test the type of a node such as `text()`, `comment()`, or the generic `node()`. The (optional) conditions in the predicate further restrict the matching nodes.

The examples from the beginning of this chapter appear to have a slightly different syntax than that just described because those expressions had no `::` separating the axis from the node-test. This is because we used the shortcuts that *XPath* provides to specify some common axes. Instead of `/Envelope`, we can write `/child::Envelope` to mean the child is the axis. Similarly, `//Cube` is equivalent to

```
/descendant-or-self::Cube
```

Instead of `@currency = "JPY"`, we can write

```
attribute::currency = "JPY"
```

Clearly, the shortcuts are more convenient. In particular, the default axis (`child::`) can be dropped entirely from the location step. While these shortcuts are very handy, it is important to think in terms of axes to fully understand what an *XPath* expression means and how it will be evaluated.

XPath is very simple, and a good understanding of axes, node-tests, and predicates allows us to do very powerful processing both efficiently and succinctly. Next, we describe these three parts of the location step in more detail.

11.7.1 The Axis

The axis provides both the direction to look for nodes (from the current context) and also how to look along that direction. Directions are expressed using the family relationship terminology from the document hierarchy, e.g., child, parent, sibling, ancestor, descendant. The **child** axis looks down the tree one level (from the current location or context) to the immediate child nodes. The **parent** and **ancestor** axes look up the tree from the current location. The **parent** axis looks up one level to the immediate parent node while the **ancestor** looks at the parent node, the parent's parent, and so on up to the root node. The **ancestor-or-self** axis considers all the ancestors as well as the current node (the current node is the **self** axis). Similarly, **descendant** and **descendant-or-self** work on descendant elements in the tree. The axes **preceding-sibling** and **following-sibling** look along the same level of the current context at all of the preceding siblings (to the left) and the following siblings (to the right), respectively.

Axis Shortcuts and Abbreviations

Some axes are very common so there are shortcuts for these that make the *XPath* expression more succinct and clearer to read. These shortcuts can be used in a location step within the location path. The most common axis is **child**. This is the default axis and can be omitted from any *XPath* step.

Example 11-5 Simplifying *XPath* Axes to Locate SDMX Nodes

For example, the following expression:

```
/child::Envelope/child::Cube/child::Cube/child::Cube
```

consists of four location steps, each one proceeding down one level to the children of each of the nodes resulting from the context of the previous step. Written in this form, the *XPath* expression is very explicit. However, it can also be written as

```
/Envelope/Cube/Cube/Cube
```

This is more convenient to write and, when one is familiar with the *XPath* syntax, easier to comprehend. It also reminds us of navigating a file system of directories. How does this *XPath* expression work? It begins at the top of the document (denoted by the first `/`). The first step looks in the direction of the children of the current context—the root of the document—and searches for a node named `Envelope`, which it finds. The current node-set is then this single `<Envelope>` node. In the second step, we look from the current context of the `<Envelope>` element in the direction of its children to locate all nodes named `Cube`. Again, there is only one `<Cube>` child of `<Envelope>` and this in turn is the context or node-set for the third step. At this step, since the topmost `<Cube>` has two children, both are found. and multiple nodes satisfy this node-test. Then, in the final location step, the context is evaluated for each of the nodes from the previous step. For each of these two nodes, we look in the direction of the node's children and find many child `<Cube>` elements. The resulting node-set will be all eight great-grandchild `<Cube>` elements of `<Envelope>`, i.e., those with the `currency` attribute. This is the union of the two sets of four `<Cube>` elements coming from evaluating the final step on each of the two nodes that were located in the third location step. (See Figure 11.3 for a diagram of the hierarchy). ■

Another commonly used abbreviation is the double forward slash `//` which is shorthand for the axis **descendant-or-self**. For example, the expression `//Cube` starts at the root node and matches eleven nodes—the topmost `<Cube>` that is a child of the `<Envelope>` node, each of the `<Cube>` nodes with a `time` attribute, and then all eight of the `<Cube>` nodes with a `currency` attribute. The first of these nodes can be matched with an absolute path `/Envelope/Cube`. The advantage of the **descendant-or-self** axis is that we need not specify the exact and complete steps in the path so many nodes at different levels in the hierarchy can match the expression. In this case, nodes match at three different levels of the tree, e.g., eight of the matches are great-grandchildren of the root element. A disadvantage is that `//` requires *XPath* to traverse every descendant node. If the document is large, an exact path can be more efficient and also more specific. However, less specific and more inclusive expressions insulate us from knowing the exact details of the structure of the *XML*. This can be good if the documents change slightly across documents, but have the same basic content and structure. This trade-off between specificity and generality is quite similar to that when working with regular expressions.

Since `//` is an abbreviation for an axis, when we include it in a location step, we should still have an additional `/` character to separate the step from the other steps. This would mean we would have three `/`s in a row. *XPath* allows us to omit the separator `/` and abbreviate this to simply `//`, e.g., `/Envelope//Cube`.

Example 11-6 Using Parent and Attribute Axes to Locate Dates in an SDMX Document
Earlier we found the dates when the exchange rate for the Czech koruna dropped below 25, by filtering the exchange rate nodes to locate those for the koruna that had rates below 25, and then we moved from each of these nodes up to its parent to access the date. Our *XPath* expression was

```
//Cube[@currency = 'CZK' and @rate < 25]/../@time
```

TABLE 11.1: XPath Axes

Axis	Description
child default axis	Child elements of the context node. Since it is the default axis, it does not have to be specified, e.g., <code>child::Sender</code> is equivalent to <code>Sender</code> .
attribute Abbreviation: <code>@</code>	Locates attributes of the context node.
parent Abbreviation: <code>..</code>	The parent node of the context node. There is no need to specify the name of the parent node because there is only one parent, e.g., <code>..</code> is a shortcut for <code>parent::node()</code> .
self Abbreviation: <code>.</code>	The context node itself, e.g., <code>./text()</code> or <code>./</code> . The self axis is used in functions in predicates, e.g., the <code>name(.) = 'Send'</code> expression can be used to test the name of the context element.
descendant	Any child, child of a child, and so on of the context node.
descendant-or-self Abbreviation: <code>//</code>	Any child, child of a child, and so on of the context node, along with the node itself. While not technically a shortcut for an axis, <code>//</code> can be used as a shortcut for <code>/descendant-or-self::node() /</code> . Note also that when locating all descendants or self of the root, we also use the shortcut <code>//</code> rather than <code>///</code> .
ancestor	Any parent, parent of a parent, and so on of the context node.
ancestor-or-self	Same as ancestor but also includes the current node itself.
following-sibling	All elements that follow the context node and share the same parent.
preceding-sibling	All elements that precede the context node and share the same parent
namespace	Locates the namespace nodes.
following	Matches all nodes after the current node in terms of the document order.
preceding	Matches all nodes before the current node in terms of the document order.

This table lists the different XPath axes we can use in an XPath location step. These specify the direction to search in the tree and how to perform that search, e.g., look at the child nodes, or at all descendant nodes.

This expression uses several shortcuts—**descendant-or-self**, **attribute**, and **parent**. The fully qualified XPath expression would be

```
/descendant-or-self::Cube[attribute::currency = 'CZK'
                           and attribute::rate < 25]/
    parent::node()/attribute::time
```

The first location step locates all `<Cube>` nodes in the document and filters them to those that have an exchange rate for the koruna that is below 25. The second location step reverses up the tree to the parent of each node located in the first step. Then, the third location step looks along the attribute direction for the `time` attribute.

A list of axes and their abbreviations is provided in Table 11.1. ■

11.7.2 The Node Test

The node-test component in a location step identifies the name or the type of node to be matched. This is often just simply the name of the nodes in which we are interested, e.g., Cube, Sender, or molecule. As mentioned earlier, there may be times when we are using multiple vocabularies and the same name is used in two different contexts. When this occurs, we use namespaces in our *XML* document to clarify the provenance and meaning of a the node name, e.g., `<r:class>` and `<py:class>`. When a document has multiple namespaces, *XPath* requires that we specify the namespace in our node-test. For example, to find *R* `<code>` nodes in an *XML* document, we might need to distinguish these `<code>` nodes from other vocabularies that use `<code>` as a node name. We can use a namespace prefix in the node-test to find all *R* `<code>` nodes in `<example>` nodes as follows:

```
//example//r:code
```

or, more explicitly,

```
/descendant-or-self::example/descendant-or-self::r:code
```

Of course, we need to associate the namespace prefix "r" with the appropriate URI, i.e., `http://www.r-project.org`. We will discuss this issue further in [?], where we will see how to specify the namespace definitions when using *XPath* in *R*.

At times we want to match a node with any name, and not a specific fixed name. In this case we can use the asterisk (*) as a wildcard that matches all named elements. For example, in [?] we used the *XPath* expression

```
/*/molecule[./name/text() = 'frm-1']
```

to locate `<molecule>` nodes one-level down without having to know the name of the top node. The * symbol is really shorthand for the `node()` function that matches any regular node, i.e., it does not match text, comments, attributes, and processing instructions.

There are other circumstances where we want to match elements in the tree that are not regular nodes and do not have a tag name, e.g., a text node or a processing instruction node. There are *XPath* functions for specifying these. If we want to match text nodes we use the node-test `text()`. Similarly, we use `comment()` to match comments and `processing-instruction()` to match any processing instruction. If we want to match only processing instructions with a particular target, we pass the target name as a string in the call to `processing-instruction()`, e.g.,

```
//processing-instruction('R')
```

11.7.3 The Predicate

Predicates allow us to further restrict the node-set, but they are not always needed and so can be omitted. A predicate tests each of the candidate nodes matched by the node-test part of the location step. For each of these nodes, the expression in the predicate is evaluated in the context of that node. If the result of the comparison is true, then the node remains in the node-set; if not, it is discarded. That is, the predicate filters the node-set. This is similar to subsetting in *R* with a logical vector, i.e.,

```
nodeset[ sapply(nodeset, predicate) ]
```

It is very important to understand that the predicate is evaluated in *XPath* in the context of the node or *XML* element we are testing in the node-test, not in the parent node's context.

The syntax of the predicate is: [logical-condition]. These conditions can be unary or binary operations. For example, `Cube[@currency]` and `Cube[not(@currency)]` are examples of unary operations which test for the presence and absence of a `currency` attribute, respectively.

As another example, we can match `<section>` nodes that contain a `<figure>` element with

```
//section[ .//figure ]
```

This expression matches and returns the `<section>` nodes, not the `<figure>` nodes, and keeps only those `<section>` nodes that have a `<figure>` element as a descendant. Note that we use the shorthand `.` for the current node or `self`.

Binary operations are very common and have the form

```
[ expr1 booleanOperator expr2]
```

where `expr1` and `expr2` are compared via *XPath* boolean operators (i.e., `=`, `!=`, `>`, `>=`, `<`, and `<=`). We saw two examples of this earlier:

```
Cube[ @currency = 'JPY' ]
Cube[ @rate < 25 ]
```

In both examples, we test on the value of an attribute. This automatically fails if the node has no attribute with that name. In the case of comparing the `rate`, *XPath* coerces the value of the attribute from a string to a number for us. There are functions in *XPath* (e.g., `number()`) to do this explicitly if we need more control over how the conversion is done. Note that the test for equality is a single `=`, not two (`==`) as in *R* and other languages. This is because there is no assignment in *XPath*.

Predicates can appear in any location step, and indeed there can be multiple predicates in a single step.

Logical Operators

XPath provides logical operators for combining predicates. That is, predicates can be combined together into a compound predicate using one of the binary operators and or or. For instance, to match `<Cube>` nodes for the US dollar or Japanese yen, we can use

```
//Cube[@currency = "JPY" or @currency = "USD"]
```

Other boolean operators in *XPath* include: `not()`, `true()`, and `false()`. The `not()` operator is used to compute the opposite or negation of a condition. It is analogous to the `!` operator in *R*, and we can use it in an *XPath* expression such as

```
//graphic[ not(contains(@fileref, '.jpg')) ]
```

to find all `<graphic>` nodes that do not have a `fileref` attribute with the extension `jpg`. *XPath* has no direct equivalent of *R*'s `!=` operator.

It is important to note that `getNodeSet()` and related functions in *R* use *XPath* 1.0 via the `libxml2` C-level library. *XPath* 2.0 provides additional and richer functions than are available in *XPath* 1.0, but we cannot use those within `getNodeSet()`, etc. While it would be convenient to use these additional functions, we do not actually need them. Instead, we can perform simpler *XPath* queries and then apply our own predicates or transformations to the nodes that `getNodeSet()` returns. We have a much richer language and set of functions in *R* than is available in *XPath* 2.0. Therefore, we can combine *XPath* and *R* to perform the computations we need. Readers interested in more powerful facilities than *XPath* 1.0 might explore the *XQuery* language [2, 15]. The *RXQuery* package [11] is a prototype of integrating *R* and *Zorba* [6], an Open Source implementation of *XQuery*.

11.8 Summary of Functions to Read *HTML*, *XML*, and *JSON* into *R* Data Frames and Lists

Below are brief descriptions of the high-level functions in the XML package for parsing an *XML* document into an *R* list or data frame.

`readHTMLTable()` Return all the tables in the document as data frames. The *which* parameter allows us to specify those tables we want to extract from the document. The function's *colClasses* parameter allows us to specify the classes/types for the columns and a function will convert the content to numbers, percentages, factors, etc.

`xmlToDataFrame()` Loop over the child nodes of an *XML* node and create a data frame, where there is one row in the data frame for each child node and the variables correspond to the value of top-level *XML* nodes in each child. If the top-level nodes in each child contain *XML* nodes, then the value of the first node is retrieved.

`xmlParse()` Parse an *XML* document. This function can read a local file, a remote *URL*, *XML* content that is already in *R* as a string, or a connection.

`htmlParse()` Parse an *HTML* document. This function is a less restrictive version of the *XML* parser that tolerates *HTML* content that is typically not well-formed in the *XML* sense.

`xmlRoot()` Retrieve the root node of an *XML* document. This is usually given the parsed document, but we can also pass it any *XML* node in a document to get the root node.

`xmlChildren()` Get a list of all the child nodes of a given *XML* node.

`xmlAttrs()` Retrieve a named character vector of all the attributes of a given node.

`xmlGetAttr()` Retrieve the value of a single attribute of a given node, optionally converting it from a string to a different type. This function also allows provision of a default value to use if the attribute is not present in the node.

Bibliography

- [1] Tim Bray, Dave Hollander, Andrew Layman, Richard Tobin, and Henry Thompson. Namespaces in *XML* 1.0. Worldwide Web Consortium, 2009. <http://www.w3.org/TR/REC-xml-names/>.
- [2] Michael Brundage. *XQuery: The XML Query Language*. Addison Wesley, Boston, MA, 2004.
- [3] Economic Commission for Europe. Common open standards for the exchange and sharing of socio-economic data and metadata: The SDMX initiative. <http://sdmx.org/docs/2002/wp11.pdf>, 2002.
- [4] European Central Bank. Euro foreign exchange reference rates. <http://www.ecb.int/stats/exchange/eurofxref/html/index.en.html>, 2011.

- [5] European Central Bank. SDMX-ML and SDMX-EDI (GESMES/TS): The ECB statistical representation standards. <http://www.ecb.int/stats/services/sdmx/html/index.en.html>, 2011.
- [6] FLOWR Foundation. Zorba: The XQuery processor. <http://www.zorba-xquery.com>, 2012.
- [7] Elliotte Rusty Harold and W. Scott Means. *XML in a Nutshell*. O'Reilly Media, Inc., Sebastopol, CA, 2004.
- [8] David Hunter, Jeff Rafter, Joe Fawcett, Eric van der Vlist, Danny Ayers, Jon Duckett, Andrew Watt, and Linda McKinnon. *Beginning XML*. Wiley Publishing, Inc., Indianapolis, IN, fourth edition, 2007.
- [9] B. N. Lawrence, R. Lowry, P. Miller, H. Snaith, and A. Woolf. Information in environmental data grids. *Philosophical Transactions of the Royal Society A: Mathematical, Physical, and Engineering Sciences*, 367:1003–1014, 2009.
- [10] Statistical Data and Metadata Exchange Initiative. SDMX information model: UML conceptual design (version 2.0). http://www.sdmx.org/docs/2_0/SDMX_2_0SECTION_02_InformationModel.pdf, 2005.
- [11] Duncan Temple Lang. RXQuery: Bi-directional interface to an XQuery engine. <http://www.omegahat.org/RXQuery>, 2011. *R* package version 0.3-0.
- [12] USGS Earthquakes Hazards Program. Latest earthquakes: feeds and data. <http://earthquake.usgs.gov/earthquakes/catalogs/>, 2010.
- [13] Daniel Veillard. The XML C parser and toolkit of Gnome. <http://www.xmlsoft.org>, 2011.
- [14] W3Schools, Inc. XML tutorial. <http://www.w3schools.com/xml/default.asp>, 2011.
- [15] Priscilla Walmsley. *XQuery*. O'Reilly Media, Inc., Sebastopol, CA, 2007.
- [16] Worldwide Web Consortium. Extensible Markup Language (XML) 1.0. <http://www.w3.org/TR/REC-xml/>, 2008.

12

Web Technologies

CONTENTS

12.1	Introduction	433
12.2	Before You Scrape!	434
12.3	Scraping Data from <i>HTML</i> Tables	434
12.3.1	435
12.3.1.1	<i>HTML</i> Forms	437
12.3.2	Specifying the Column Types	440
12.4	Scraping Links from <i>HTML</i> Pages	441
12.5	Overview of <i>HTML</i> and <i>XML</i>	441
12.6	Scraping Arbitrary Content from <i>HTML</i> Pages with <i>XPath</i>	441
12.6.1	Finding Patterns in the <i>HTML</i> Content and using <i>XPath</i>	441
12.7	Scraping Data from <i>HTML</i> Forms	441
12.8	Dynamic <i>JavaScript</i> Content with Selenium	441
12.9	Basics of <i>HTTP</i> Queries	441
12.10	REST-based Web Services	441
12.10.1	<i>XML</i>	441
12.10.2	<i>JSON</i>	441
12.11	Summary	441
12.12	Guided Practice	441
12.13	Exercises	441
	Bibliography	441

12.1 Introduction

In the current Internet era, we can make great use of the many sources of “found” data. This allows us to use these data sets to ask our own questions, or to fuse one or more of them together with our own data to provide richer analyses. In some cases, we can download an entire dataset from a Web site as a single file, e.g., a CSV file or a **xlsx** spreadsheet. However, increasingly data of interest are directly displayed as part of Web pages or made available through application programming interfaces (APIs). To be able to make use of these data, we need ways to access the data. This involves a) making requests to the Web servers to access the content, and b) to extract the content from the Web server’s response. We need to know how to make *HTTP* requests, and parse and process *HTML*, *XML* and also *JSON* content. This chapter provides a reasonably comprehensive overview and introduction for readers to all of these topics.

In some cases, we will have to access “private” data and so will need to include authentication in our requests to the Web server. There are various different methods in effect

to provide restricted access to data and we will explore some of the more commonly used methods.

12.2 Before You Scrape!

When you start to scrape a Web site or query an API, you should always ask the following simple questions:

- When scraping *HTML* pages, can we download the data in a more convenient form such as a CSV document, an *R* data file, a database, an *XML* or *JSON* document, etc? These are typically faster and easier to download, easier to work with, and reduce the burden on the Web server.
- Is there already an existing *R* package (or *Python* module, ...) that provides more structured access to these data?
- Contact the owners of the data and the Web site and ask them if they are willing to share the data with you (and others) in a more direct form.
- Are you legally entitled to programmatically access the data from the Web site? This relates to the terms of service (ToS) for the site. If you are not entitled to do this, your account or IP address or IP network may be (temporarily) banned from accessing the site.
- Read the robots.txt file for the site and understand what URLs you are entitled to scrape.
- Will you place an excessive burden on the Web site by making so many queries? If so, wait a suitable time between requests.
- Perfect and thoroughly test your code with a small number of requests before making lots of request that may be incorrect.
- Enclose your requests within a `try()` or `tryCatch()` call so that an error in one request don't terminate the others and lose all the previous results.
- Cache the results of each request locally so that you don't have to repeat the request to the server if you need the results at a later date¹

The rapid growth in the availability of data on the Web gives us great opportunities to “fuse” two or more different data sources to gain insights. This is one of the very exciting aspects of data science. However, we should always think very carefully about the nature of “found data”. Were the observations sampled, or is this the entire population? What is the population? How were the data sampled? What are the explicit and implicit biases in the collection mechanism? What are the potential implications of these biases? Do the data really measure what we think they do? Can they help us to answer our real questions, or are they merely convenient but distracting?

¹Of course, if the results have changed, you will need to repeat the request.

12.3 Scraping Data from *HTML* Tables

We don't really need to know the structure of the *HTML* page other than the data we want appear as tables. These may not be `<table>` elements, but we can try.

12.3.1

The price of gas/petrol for cars and light trucks is important for most households and it also varies significantly during the year and by geographical location. It is interesting to see how much variation there is and also how it relates and lags behind changes in the price of oil. The Californian state government provides weekly data on the average retail price of *branded* and *unbranded* gas, along with details about what make up these costs, e.g., crude oil cost, refinery costs and profits, distribution and marketing costs and profits, taxes,

We can get the weekly data for the current year via the URL `http://www.energy.ca.gov/almanac/transportation_data/gasoline/margins/`. We can see a screenshot of the page in Figure 12.1 We see the data arranged by weeks in rows and columns for different dollar amount.

```
library(XML)
u = "http://www.energy.ca.gov/almanac/transportation_data/gasoline/margins/"
tbls = readHTMLTable(u, stringsAsFactors = FALSE)
```

By default, `readHTMLTable()` extracts all of the tables on the *HTML* page and returns them as a *list*. We can query how many there are on the page with

```
length(tbls)
```

In this case, there is only one table.

We access the table as the first and only element of the list:

```
tbl = tbls[[1]]
```

This should be a `data.frame` and we can check this. We can query the number of rows and columns with

```
dim(tbl)
```

```
[1] 38 19
```

We can look at the first few rows with

```
head(tbl)
```

	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12
1	<NA>	<NA>	<NA>	<NA>	<NA>	<NA>	<NA>	<NA>	<NA>	<NA>	<NA>	<NA>
2		Branded		Unbranded	<NA>	<NA>	<NA>	<NA>	<NA>	<NA>	<NA>	<NA>
3												
4	Aug 29		\$0.391	\$1.099	\$0.677	\$0.020	\$0.060	\$0.278	\$0.184	\$2.709		\$0.51
5	Aug 22		\$0.378	\$1.102	\$0.681	\$0.020	\$0.059	\$0.278	\$0.184	\$2.702		\$0.51
6	Aug 15		\$0.475	\$1.079	\$0.549	\$0.020	\$0.058	\$0.278	\$0.184	\$2.643		\$0.51

This isn't in perfect form with all the NAs, but we see the three weeks of values that we want. Looking at the `tail()` of the data frame, we see the data from the start of the year:

```
tail(tbl)
```

	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12	V13
33	Feb	08	\$0.636	\$0.724	\$0.567	\$0.020	\$0.055	\$0.300	\$0.184	\$2.486		\$0.741	\$0.724
34	Feb	01	\$0.572	\$0.757	\$0.662	\$0.020	\$0.056	\$0.300	\$0.184	\$2.551		\$0.664	\$0.757
35	Jan	25	\$0.568	\$0.673	\$0.840	\$0.020	\$0.058	\$0.300	\$0.184	\$2.643		\$0.789	\$0.673
36	Jan	18	\$0.596	\$0.664	\$0.888	\$0.020	\$0.060	\$0.300	\$0.184	\$2.712		\$0.852	\$0.664
37	Jan	11	\$0.532	\$0.709	\$1.034	\$0.020	\$0.063	\$0.300	\$0.184	\$2.842		\$0.698	\$0.709
38	Jan	04	\$0.315	\$0.856	\$1.137	\$0.020	\$0.063	\$0.300	\$0.184	\$2.875		\$0.368	\$0.856

From the display, we might guess that the second and eleventh column are empty. These are used to provide the visible separations between the columns. We can check these columns to see if they contain any values. These both have NA values in the header rows and empty values in the body.

where tanker trucks load their fuel from a distribution terminal's loading rack. The unbranded price is also based on OPIS pricing information.															
The distribution margin can be either positive or negative in value. A negative distribution margin implies that some gasoline is being sold at a loss. Similar to the refining margin, the distribution margin also includes the costs and profits of operating the retail gas station as well as various transportation and storage fees incurred once gasoline is moved from the bulk terminal to the retailer. Most branded franchisees purchase gasoline at a delivered price called the Dealer Tank Wagon price that is typically higher than the branded rack price. A retail-specific margin is not available at this time.															
Table Definitions															
Branded							Unbranded								
Distribution Cost, Marketing Costs and Profits	Crude Oil Cost	Refinery Cost and Profits	State Underground Storage Tank Fee	State and Local Sales Tax	State Excise Tax	Federal Excise Tax	Retail Prices	Distribution Cost, Marketing Costs and Profits	Crude Oil Cost	Refinery Cost and Profits	State Underground Storage Tank Fee	State and Local Sales Tax	State Excise Tax	Federal Excise Tax	Retail Prices
\$0.391	\$1.099	\$0.677	\$0.020	\$0.060	\$0.278	\$0.184	\$2.709	\$0.517	\$1.099	\$0.551	\$0.020	\$0.060	\$0.278	\$0.184	\$2.709
\$0.378	\$1.102	\$0.681	\$0.020	\$0.059	\$0.278	\$0.184	\$2.702	\$0.517	\$1.102	\$0.542	\$0.020	\$0.059	\$0.278	\$0.184	\$2.702
\$0.475	\$1.079	\$0.549	\$0.020	\$0.058	\$0.278	\$0.184	\$2.643	\$0.571	\$1.079	\$0.453	\$0.020	\$0.058	\$0.278	\$0.184	\$2.643
\$0.540	\$1.017	\$0.583	\$0.020	\$0.059	\$0.278	\$0.184	\$2.681	\$0.675	\$1.017	\$0.448	\$0.020	\$0.059	\$0.278	\$0.184	\$2.681
\$0.656	\$1.133	\$0.394	\$0.020	\$0.060	\$0.278	\$0.184	\$2.725	\$0.871	\$1.133	\$0.179	\$0.020	\$0.060	\$0.278	\$0.184	\$2.725
Aug 29															
Aug 22															
Aug 15															
Aug 08															
Aug 01															
Jul 25															
Jul 18															
Jul 11															
Jul 04															
Jun 27															
Jun 20															
Jun 13															
Jun 06															
May 30															

Figure 12.1: California Weekly Average Gas Prices. *The Web page shows the weekly average gas prices, US. state and federal taxes, costs and profit amounts.*

Convert the values in the first column to actual dates with the year, month and day.

Convert the values in the columns containing currency values from strings to numeric values.

We converted the currency columns using `substring()` or regular expressions. In other tables, we have values that are formatted with commas (,), e.g., 2,345,678. Another common format in tables is percentages, e.g., 15%. We can post-process the text after we get the data frame from `readHTMLTable()`. However, we can also specify the target classes of each column using the `colClasses` analogous to the same parameter in `read.table()`. We pass either a `character` vector or `list` of class names. These can be the names of common types in R, e.g., "numeric", "integer", "factor", etc. The XML package introduces a few other useful classes/targets. These are "Currency", "FormattedInteger", "FormattedNumber" and "Percent" for dealing with the cases described above. We can also specify NULL or "NULL" for a column that we want to drop/omit. We can use these facilities to read and convert the values in our table in a single call:

```
colTypes = c("character",
            "NULL",
            rep("Currency", 8),
            "NULL",
            rep("Currency", 8))
tbl = readHTMLTable(u, which = 1, colClasses = colTypes)
```

We leave the first column as a string and we will convert it to a date later. We use "NULL" for the second column since it is empty and just a visual separator. The next 8 columns are currency values so we use the class `Currency`. And similarly, we drop the column between the branded and unbranded and convert the 8 columns in the second table to `Currency` values.

The first few rows of the table

In many cases, we want just one table from a page and we want to be able to specify the details of the column classes and the header. However, if we specify values for the `colClasses` and `header()` parameters, those apply to all of the tables in the page. However, the `which` parameter for `readHTMLTable()` allows us to identify one or more tables in the page we want. Then the `colClasses` and `header` apply to just those tables. So we typically start by calling `readHTMLTable()` once to retrieve all the tables, and then determining which is the one we want. Then we can call `readHTMLTable()` with that table's index via the `which` argument and add values for `header` and/or `colClasses`.

12.3.1.1 HTML Forms

We retrieved the data gas price values for 2016. However, we want them for many other years. If we explore the page, we see a pull-down menu and a button with the label Get different year, shown in Figure 12.2. We chose a year and then click on the button. When we do this, we get a new page. However, the URL in the Web browser's navigation field does not change. So it is not clear how we would request this page using `readHTMLTable()`. We need to understand what the browser does to send the request and mimic this.

There are two ways to find out how to make the request for the different year's data. One is to read the details of the `HTML` form on the original page, i.e., the pull-down menu and button. The other is to use tools within the Web browser to see the details of the browser's request. Both can be very useful in different circumstances, especially with requests made by `JavaScript` code within an `HTML` document where we cannot simply read the `HTML`.

We can look at the `HTML` for the original page either in our Web browser (using the View Developer View source menu sequence in Google Chrome) or in `R` directly. We'll do this in `R` a little later. For now, we'll use the Web browser to view the source. We search for the string Select Year and find it within the following `<form>` element:

```

<form action='index.php' method='post'>
  <label for='year'>
    <select name='year' id='year'>
      <option value='2016'>Select Year</option>
      <option value='2015'>2015</option>
      <option value='2014'>2014</option>
      <option value='2013'>2013</option>
      <option value='2012'>2012</option>
      <option value='2011'>2011</option>
      <option value='2010'>2010</option>
      <option value='2009'>2009</option>
      <option value='2008'>2008</option>
      <option value='2007'>2007</option>
      <option value='2006'>2006</option>
      <option value='2005'>2005</option>
      <option value='2004'>2004</option>
      <option value='2003'>2003</option>
      <option value='2002'>2002</option>
      <option value='2001'>2001</option>
      <option value='2000'>2000</option>
      <option value='1999'>1999</option>
    </select>
  </label>
  <input name='newYear' type='submit' value='Get different year' />
</form>

```

This is *HTML*. We have a `<form>` element or node, and within it, we have two sub-elements named `<label>` and `<input>`. The `<input>` element is quite simple and has no sub-nodes of its own.

It does have attributes within the `<input>`. These are of the form `attributeName='value'`. These are `name='newYear'`, `type='submit'` and `value='Get different year'`. The attribute names are `name`, `<type>` and `<value>`. The `<type>` of this button is `submit` which means that when a viewer clicks on it, the browser submits the form. This involves collecting the currently selected values from the elements in this form and then send these as `elementName=value` pairs. The browser sends the request to the URL specified in the `<action>` attribute of the `<form>`. In this case, this is `index.php`. This is a relative name and we have to convert it to a full URL. This is relative to the URL of the current page, `http://www.energy.ca.gov/almanac/transportation_data/gasoline/margins/`.

The `<label>` element has a child node `<select>`. It has an associated variable name `year`. It also has an `<option>` sub-node for each of the possible years we can select. Note that each of these `<option>` elements has the year repeated twice, once in the `value` attribute and again as the content between the opening `<option>` and closing `</option>` element. The latter is what is displayed by the browser in the menu; the former is the value that is sent when the form is submitted. These can be different but in this case, they are the same.

When we click on the button to submit the form, the Web browser gets the current value of each of the form elements, i.e., the `<select>` and the `<input>` button and sends them to the URL in the `action` attribute of the `<form>`. It sends this as an *HTTP* request, in this case using a **POST** request rather than a **GET** request due to the `method` attribute of the `<form>`. This sends the `name=value` pairs of the form variables and selected values in

the body of the *HTTP* request. It combines them into a string for the body of the request, separating them by the & character. It also “escapes”, or URL encodes, certain characters such as & and space by mapping them to base64 encoding. The browser takes care of all of this.

How do we emulate this in *R*. When we used `readHTMLTable()` above, we gave it the URL and it both made the request to the Web server for the document and then parsed the result and extracted the table(s). However, `readHTMLTable()` doesn’t know how to make an *HTTP POST* request. Instead, we have to do this separately and then pass the result from the request to `readHTMLTable()`. So we make the *HTTP* request and then parse the document in two separate steps.

We can use the `RCurl`, `curl` or `httr` packages to make an *HTTP* request. The `postForm()` function in the `RCurl` package takes the URL and `name = value` pairs.

```
library(RCurl)
txt = postForm(u, year = "2013", newYear = 'Get different year')
```

The result is a character string that contains the entire *HTML* document. We pass this text to `readHTMLTable()` which recognizes that it is not the name of a file, but the actual content of an *HTML* document:

```
g13 = readHTMLTable(txt, which = 1, colClasses = colTypes)
```

We often don’t need to include the value of the submit button when submitting the form, but we can for completeness and in some cases, it is necessary.

	\$0.596	\$0.664	\$0.888	\$0.020	\$0.060	\$0.300	\$0.184	\$2.712	\$0.852	\$0.664	\$0.632	\$0.020	\$0.060	\$0.300	\$0.184	\$2.712
Jan 18	\$0.596	\$0.664	\$0.888	\$0.020	\$0.060	\$0.300	\$0.184	\$2.712	\$0.852	\$0.664	\$0.632	\$0.020	\$0.060	\$0.300	\$0.184	\$2.712
Jan 11	\$0.532	\$0.709	\$1.034	\$0.020	\$0.063	\$0.300	\$0.184	\$2.842	\$0.698	\$0.709	\$0.868	\$0.020	\$0.063	\$0.300	\$0.184	\$2.842
Jan 04	\$0.315	\$0.856	\$1.137	\$0.020	\$0.063	\$0.300	\$0.184	\$2.875	\$0.368	\$0.856	\$1.084	\$0.020	\$0.063	\$0.300	\$0.184	\$2.875

Select Year **Get different year**

2015
2014
2013
2012
2011
2010
2009
2008
2007
2006
2005
2004
2003
2002
2001
2000

Price: The average wholesale gasoline price is the average of 13 unbranded and 13 branded wholesale prices at various wholesale fuel loading racks around the country for a single day. The wholesale gasoline price is calculated for the same day as EIA's weekly average gasoline price.

Branded Gasoline: Branded gasoline refers to fuel that is sold under a brand name (such as BP, Shell, Exxon, Chevron, and Valero). Branded gasoline will include taxes. Unbranded gasoline is not associated with a specific brand name, and is typically sold by single-station retail outlets, relatively small chain retailers that sell gasoline, and large supermarket chain stores (such as Costco and Safeway).

Marketing Costs, and Profits: The costs associated with the distribution from terminals to stations and retailing of gasoline, including but not limited to: franchise fees, utilities, supplies, equipment maintenance, environmental fees, licenses, permitting fees, credit card fees, insurance, depreciation, advertising, and profit.

Crude Oil Price: The daily market price of Alaska North Slope crude oil which is used as a proxy for this composite crude oil acquisition cost for California refineries.

Profits: The costs associated with refining and terminal operations, crude oil processing, propane additives, product shipment and storage, oil spill fees, etc.

Figure 12.2: *HTML* Form Menu to Select a Different Year. The viewer can use the menu to select a different year to get a new page with the values for that year.

What’s the difference between **POST** and **GET**? When we request a regular document, we typically use **GET** and just pass the fixed URL. Many *HTML* forms use **GET** to send additional `name=value` parameters, similar to what we did above. Unlike `postForm()`, `getForm()` submits a form using **GET** and appends the parameters and their values to the URL. We separate the actual URL from the parameters with a ? and separate the `name=value` pairs again with a &. A familiar example is when we use Google search. For example, when we search for *HTML* form **GET**, the actual request is

```
https://www.google.com/search?q=HTML+form+GET&oq=HTML+form+GET&aqs=chrome..69i57j015..
```

depending on your particular browser. The URL is `https://www.google.com/search` and the parameters are named q, oq, aqs, surceid and ie. Their values are HTML form GET (for the first two), `chrome..69i57j015.16129j0j8`, chrome and UTF-8. Note how we have converted the spaces into +.

The `getForm()` and `postForm()` both take the URL and the name = value pairs. They hide the how they request is actually made. The `getForm()` creates the URL, while `postForm()` puts the parameters and values in the body of the request, not in the URL. One of the important differences is that a URL can only contain so many characters, whereas we can have a large amount of data in the body of a request. We can specify that we want just one specific table. Often we get them all, determine which we want, and then ask for just that one in a second request. This allows us to control the header, column classes, etc. for just that table.

12.3.2 Specifying the Column Types

12.4 Scraping Links from *HTML* Pages

12.5 Overview of *HTML* and *XML*

12.6 Scraping Arbitrary Content from *HTML* Pages with *XPath*

12.6.1 Finding Patterns in the *HTML* Content and using *XPath*

12.7 Scraping Data from *HTML* Forms

12.8 Dynamic *JavaScript* Content with Selenium

12.9 Basics of *HTTP* Queries

12.10 REST-based Web Services

12.10.1 *XML*

12.10.2 *JSON*

12.11 Summary

12.12 Guided Practice

12.13 Exercises

Bibliography

Part IV

Data Science Projects and Case Studies

13

Projects

CONTENTS

13.1	Digit Recognition	445
13.1.1	Goal	445
13.1.2	Background	445
13.1.3	The Data	446
13.1.4	Tasks	447
13.1.5	k-Nearest Neighbors	448
13.1.6	Cross Validation and Model Selection	449
13.2	Adhoc Networks (Simulation Study)	450
13.3	Graphs: perhaps voting records of senators, citation networks, Enron Email (graphs)	451
13.4	Web Cache (Poisson arrival process)	451
13.5	Presidential Election (Maps)	451
13.6	NASA Satellite	451
13.7	Indoor Positioning System (Nearest neighbor and Cross-Validation)	451
13.8	Spam Filtering (classification trees)	451
13.9	State of the Union Speeches (Term Frequencies and Hierarchical Clustering and Multi-dimensional Scaling)	451
13.10	(Large Data)	451
	Bibliography	451

13.1 Digit Recognition

13.1.1 Goal

The goal of this project is to take an image of a handwritten single digit, i.e., 0, 1, 2, 3, ..., 9, and determine which digit it is. These numerals come from, e.g., ZIP codes on mail envelopes. The mail delivery system scans the ZIP code on the envelope and tries to recognize the sequence of 5 numerals in order to automate the sorting and delivery of mail. These ZIP codes are typically hand-written and vary in appearance, making it a challenge to recognize some digits. In this project, our goal is to develop a classifier to recognize handwritten digits. However, we work with single digits, rather than sequences of 5 digits.

13.1.2 Background

The data for this project are originally from the MNIST (Modified National Institute of Standards and Technology) dataset. This dataset is a classic that has been extensively studied within the Machine Learning community for bench marking the effectiveness of

classifiers and image processing systems. More details about the original dataset, including algorithms that have been tried on it and their levels of success, can be found at <http://yann.lecun.com/exdb/mnist/index.html>. The particular data that we use here are a subset of the data available from Kaggle at <https://www.kaggle.com/c/digit-recognizer>.

13.1.3 The Data

The data we have are for 5,000 images, each displaying one digit. An image is a 28×28 grid of pixels, for 784 pixels in total. The value of each pixel is an integer between 0 and 255, inclusive. These values indicate the lightness or darkness of the pixels, with 0 being white or empty and 255 being black. Figure 13.1 shows a sample image for each digit. These images have been ‘hand’-classified so that we have the true number that was written in addition to the image pixels.

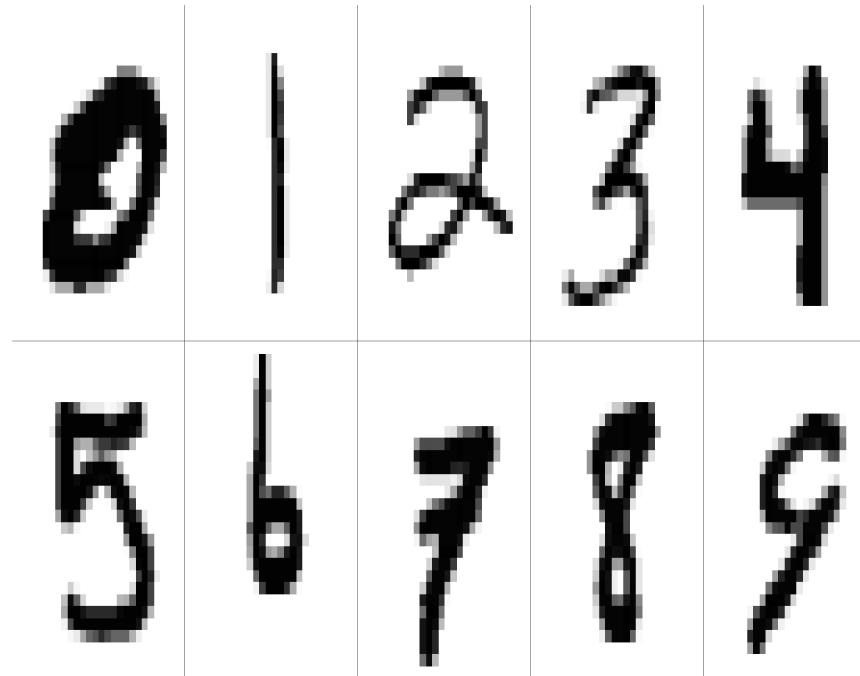


Figure 13.1: Sample Digit Images. A set of 10 sample images of the handwritten digits, one for each of the numerals 0, 1, 2, ..., 9. These images are from the NMIST database.

While the 28×28 grid of pixels corresponds to rows and columns in the image, they are provided as a comma-separate sequence of 784 values in the data file `digitsTrain.csv`. Each row in the file contains 785 comma-separated values. The first row provides the header, and subsequent rows contain the pixel values with one row per image. The first value in each row is the hand-classified ‘label’ that identifies the true digit drawn by the user, i.e., 0, 1, ..., 9. The remaining 784 values in a row contain the pixel-values of the associated image; these are called ‘pixel0’, ‘pixel1’, ..., ‘pixel783’. Below is an example of the file format:

```
"label","pixel0","pixel1",...,"pixel212","pixel213",...,"pixel783"
3,0,0,0,...,32,19,...,0
```

```
0,0,0,0, ...,252,253,...,0  
...
```

Note the first of the images is a 3 and the second is a 0. The ‘pixel0’ value corresponds to the top-left corner of the 28×28 -pixel image; pixel1 is the next pixel over from this, i.e., in the top row and second column; pixel2 is the next one over, and so on; pixel27 corresponds to the pixel in the top-right corner, i.e., the final one in the top row; and pixel28 is the leftmost pixel in the second row from the top.

13.1.4 Tasks

Creating a Data Structure for Analysis in R

It is relatively simple to read the data into a data frame in *R* with 5000 rows and 785 columns. However, this structure may not be the most amenable to analysis. What other structures might be more computationally convenient for exploration and classification? If we ignore for the moment the digit identifier, the values for each record in the data frame are 784 integers between 0 and 255. Would a 3-dimensional array with dimensions 28 by 28 by 5000 be better suited? Or, how about organizing the data as a list of 5000 28 by 28 matrices? We can begin working with the data frame, and as we get a better understanding of how we want to analyze the data, revisit this step and modify the data structure.

Viewing an Image

One of the first things we want to do is to ‘look’ at the images. That is, make a plot, such as one of those in Figure 13.1 that displays the 784 values as an image. This helps us confirm the data are as expected. Moreover, if we wrap our code to display an image into a function, then it can be handy for when we examine the data, debug our code, and check which images or features our classification method has difficulties with. Call this function `draw()`. It has one required parameter, called `vals`, which takes the pixel values for one image, and one optional parameter, called `colors`. The default value for `colors` is a vector of 256 grey-scale colors in sequence from white to black, i.e., `colors = rgb(255:0, 255:0, 255:0, maxColorValue = 255)`. You might also consider adding a ... argument to `draw()` in order to allow the caller to pass other plotting parameters through to the plotting function within `draw()`.

Exploration

Use the `draw()` function to examine individual images. Use other statistical summaries and plots to explore the observations. Consider the variability within digits and across digits, and try to identify potential outliers and anomalies. What do your findings tell you about the classification process?

Setting Aside a Test Set

When developing a prediction method, we often work with two sets of data: a training set and a test set. We use the training set to build our model, and once the model is built, we use the test set to assess the quality of the predictions. If we are given only one dataset, we can split it into 2 parts and use one part as our test set and the other as our training set. We want both the training and test data to be representative of the whole dataset. One way to achieve this is to create groups at random. We can also control the random sampling process so that the proportions of each of the 10 digits in the test and training sets match the proportions in the whole data set, or nearly so. Also, we typically want at least half of the data to be in the training set.

Building a Classifier

There are numerous different statistical techniques we can use to classify digits. Here, we

use a relatively simple and intuitive approach, called k -nearest neighbors or k -NN for short. This method is briefly described in Section 13.1.5. Another method that may be fruitful is Naive Bayes, where we compute the probability of each digit given the pixel-values of the image and select the digit with the highest probability. Yet another approach is to apply either recursive partitioning (classification trees) or the related method, random forests.

Selecting a Classifier from Among Competing Methods

To apply the k -NN methodology, we need to select a value for k . We want the k that gives us the smallest prediction error. However, there is no closed-form analytic method to determine the optimal value of this parameter. Instead, we use cross-validation to assist us in estimating the prediction error for the various k s and choosing the best k . We can also think of the metric used in determining the distance metric between neighbors as a parameter that needs estimation. A simple choice is between the traditional Euclidean distance and \mathcal{L}^∞ distance (also known as Manhattan distance). We can use cross-validation to choose between these metrics as well. We describe the technique of cross-validation in the context of this project in Section 13.1.6. What model, i.e., value of k and choice of metric, is selected based on cross-validation? We can draw a plot showing the overall cross-validation misclassification rate versus k and the distance metrics to make a visual comparison of the competing predictors.

Distance to Average

We can compare the results for the k -nearest neighbors with a similar idea. Here we compute the average ‘image’ for each label, i.e., the average for each pixel variable/location. Then, we find the distance between a new observation and each of these group mean images. We take the closest average image as our prediction. This technique does not use individual observations as neighbors, but the distance to the average image for each label. We can use cross-validation to estimate the misclassification rate for this procedure. How do the results compare to the k -nearest neighbors results?

Assessing the Predictor with Test Data

After the prediction method has been chosen, we want to assess how well our predictor performs on the test data. The test data were set aside before we started our model fitting so they have no influence of the selection of the model. For each image/observation in the test data, we find the k closest observations in our training data, and use these to make our prediction. Since we also have the true digit for each image in the test data, we can then summarize the accuracy of our method based on predictions for the test data. To do this, we consider the following questions:

1. What are the Type I and II errors for the test set, using the chosen value of k and metric? In other words, calculate the confusion matrix.
2. Which digits are generally classified best? worst?
3. Which digits were generally confused with others?
4. In answering these questions, can we find some of examples of digits that were misclassified and, based on their image, suggest why these were misclassified?

13.1.5 k-Nearest Neighbors

The idea behind the nearest neighbor method (for $k = 1$) is quite simple. To recognize an image as representing a particular (unknown) digit, we first find the image in our training data that is closest to this unclassified image. By close we mean that we calculate the

distance between 784 pixel readings of the image of the unknown digit and an image in our training data. Then, we simply use the true digit of the closest training observation as our prediction for the digit represented by the unclassified image.

We naturally think of measuring the distance between two images with Euclidean distance, i.e.,

$$\sqrt{(pixel_0^* - pixel_0)^2 + \dots + (pixel_{283}^* - pixel_{283})^2},$$

where $pixel_i$ is the value of the i th pixel for a training observation, and $pixel_i^*$ is the i th pixel value for new image whose true digit we are trying to predict.

We can generalize this approach to k -nearest neighbors, for k larger than 1. In this case, we find the k closest images in the training data to the unclassified image. Then, we use the true digits for these k images to predict the digit of the unknown image. Essentially, we take the most common digit among these k neighboring images as our prediction.

More generally, we describe the k -nearest neighbor method as follows. Suppose we have n observations in a training data set, where each observation consists of a known label y and p variables x_1, x_2, \dots, x_p . In this project, y is the true digit and x_1, x_2, \dots, x_{784} are the pixel values for the 784 pixels in the image. Then, when a new observation arrives, e.g., $x_1^*, x_2^*, \dots, x_p^*$, we predict the label for this observation as follows:

1. Compute the distance between the ‘new’ observation, i.e., $x_1^*, x_2^*, \dots, x_p^*$ and all of the observations in our training set.
2. Find the k closest observations in the training set to this new observation.
3. Use the y values for these k nearest training observations, to find the most common label, and that is our prediction for y^* .

13.1.6 Cross Validation and Model Selection

Cross-validation is a technique for estimating model prediction error. A natural approach to see how the model predicts new data, i.e., external data that was not used to fit the model(s). Unfortunately, we often do not have such new data. If we use our data to both fit a model and assess how well the fitted model predicts that same data, then when choosing between models, we run into the danger of over-fitting, i.e., picking models that fit our data too well and under estimates the error in predicting new data. Cross-validation was developed as a way to avoid overfitting a model when comparing the predictive abilities of two or more competing models.

There are a few different kinds of cross-validation. We describe two here: leave-one-out cross-validation and v -fold. Leave-one-out cross-validation is conceptually the simplest so we describe it first. With leave-one-out cross-validation, we drop a single observation from our data, fit the model on the remaining $n-1$ observations, and treat the dropped observation as new data and find the prediction error for this left-out observation. We repeat this process, each time dropping a single observation, fitting the model on the remaining data, and computing the prediction error for the observation that was dropped. We can then combine these errors into an overall estimate of the expected error in predicting a new observation. When we have competing models, we use this approach to estimate performance for each model and select the model with the smallest error.

The technique of v -fold cross-validation is similar to leave-one-out, except that rather than leave out one observation at a time, we drop a group of observations. For example, when $v = 5$, we divide our data at random into 5 equal-sized non-overlapping groups, also known as folds. Then we drop one fold (rather than one observation) from the data, fit the

model on the remaining folds (4/5 of the data), and compute the prediction errors using this fitted model for each observation in the fold that we dropped. We repeat this process 4 more times, each time dropping a fold (a different 1/5 of the data). As with leave-one-out, we aggregate the prediction errors for all observations in all folds and use it to choose between models.

More concretely, suppose we have n observations in a data set, where each observation consists of a known label y and p variables x_1, x_2, \dots, x_p . Our goal is to predict the label y^* for a new observation given the variables $x_1^*, x_2^*, \dots, x_p^*$ and the model fitted on the n observations. We use cross-validation to estimate the expected prediction error. For example, this may be $E[(y^* - \hat{y})^2]$, or for a categorical response y it may be the expected misclassification rate.

The steps for 5-fold CV are as follows:

1. Randomly permute the n observations.
2. Remove the first 20% as the test/hold-out set (i.e., $1/v$ of the data).
3. Fit the model using the remaining 80% of observations (i.e., $(v - 1)/v$ of the data).
4. Predict the y values for the test/hold-out set.
5. Return the hold-out observations to the original data set, and remove the next 20%.
6. Repeat the previous 3 steps 4 times ($v - 1$ times) so that each fold.
7. Compute the total squared prediction error across all of the observations that we held-out in all steps. This consists of n predictions, one for each observation in our original data set.

Useful Functions

A few functions that may prove useful in your work include: `dist()`, `colMeans()`, `order()`, `sapply()`, `image()`, `table()`, and `diag()`.

13.2 Adhoc Networks (Simulation Study)

13.3 Graphs: perhaps voting records of senators, citation networks, Enron Email (graphs)

13.4 Web Cache (Poisson arrival process)

13.5 Presidential Election (Maps)

13.6 NASA Satellite

13.7 Indoor Positioning System (Nearest neighbor and Cross-Validation)

13.8 Spam Filtering (classification trees)

13.9 State of the Union Speeches (Term Frequencies and Hierarchical Clustering and Multi-dimensional Scaling)

13.10 (Large Data)

Bibliography

14

Case: Reading Data with a Diverse Set of Approaches

CONTENTS

Bibliography	453
--------------------	-----

Bibliography

Index

.[,], 428, 430
..[,], 428
//[,], 427, 428
abline(), 140
abs(), 6, 12, 49
aFunc(), 158, 159
all(), 17, 18, 218, 219
all(), 45
all.equal(), 355
ancestor, 426, 428
ancestor-or-self, 426, 428
any(), 18, 218
any(), 45
API, 433, 434
apply(), 33, 68, 70, 189, 230, 233
apply(), 188
as.datatype(), 46
as.numeric(), 19, 48
as.POSIXlt(), 200
attribute, 428

barchart(), 139
barplot(), 141, 142
BMLGrid, 341
bootQuantile(), 249
bootStat(), 256–259
boxplot(), 139
browser(), 262, 264, 288

C, iv, 4, 345, 357, 413, 419, 430
c(), 26, 34, 207, 238, 351
c(), 46
calcBMI(), 198, 199, 203, 205, 209, 210, 212, 213
calcBMI.alt1(), 210
calcBMI.alt2(), 210, 211
calcDist(), 283–286, 289, 292, 297
calcError(), 294
calcErrors(), 291, 293
call frame, 340
call stack, 340
cbind(), 355

ceiling(), 255
character, 15, 46, 59, 60, 86, 277
child, 426, 428
class(), 12–14, 21, 33, 156, 161, 163
class(), 45
codetools, 211, 238, 241
col(), 345
colMeans(), 450
color, 345
convert(), 209–211
convertLiquid(), 220
countYears(), 204
createConfig(), 355
createGrid(), 335, 338, 340, 344, 350, 356
CSV, 433, 434
cumsum(), 292
curl, 439
Currency, 437
cut(), 105, 162
cvNNPred(), 299

data frame, 46
data.frame, 435
datatype, 46
debugging, 339
descendant, 426, 428
descendant-or-self, 426–428
diag(), 450
dim(), 21, 351, 352
dim(), 45
dist(), 281, 284–286, 297, 298, 450
dnorm(), 47, 313, 333
dnormalmixture(), 333
do.call(), 259
do.call(), 188
DocBook, 410, 411
docName(), 413
DOM, 412, 414, 419, 420
dotchart(), 139
dplyr, 78, 85
draw(), 447
DTD, 411
dxxx(), 313

error, 340
 European Central Bank, 405
`exists()`, 42
`exists()`, 47
`expand.grid()`, 44, 231
`expand.grid()`, 46
`expect_identical()`, 264

`facet()`, 145
`factor()`, 63
`factor`, 15, 46, 59, 60, 63, 73, 80, 103, 278
`fAndG()`, 241
`fAndg()`, 241
`fAndgNoCheck()`, 241
`fibFor()`, 227, 228
`fibRec()`, 228
`fibTau()`, 227, 228
`filter()`, 44
`find()`, 41, 42, 210
`find()`, 47
`findGlobals()`, 211, 238, 348
`findGlobals()`, 241
`findNN()`, 283, 286, 288, 292
`findPattern()`, 374, 375
`flipRepeat()`, 231
`flipWhile()`, 229, 231, 232
`floor()`, 10, 189, 255, 296
`following`, 428
`following-sibling`, 426, 428
 FORTRAN, 4
`fromJSON()`, 178, 182, 195, 222
`fromJSON()`, 187
`fromJson()`, 222
`fsex()`, 13
 FTP, 380
`full_join()`, 85
`fwf_empty()`, 103
`fwf_positions()`, 58, 103
`fwf_widths()`, 103

`gather()`, 75
`geom_point()`, 145
`geom_XXX()`, 143
GET, 379, 380, 438, 439
`getCarLocations()`, 348, 349, 353, 355, 356
`getDefaultNamespace()`, 420
`getForm()`, 439, 440
`getNextPosition()`, 348, 351, 352, 356
`getNextPositions()`, 348
`getNodeSet()`, 418, 430
`getSibling()`, 419, 421

`getURL()`, 174
`getURL()`, 187
`getURLContent()`, 187
`ggplot()`, 142, 143, 145
`ggplot2`, iii, 108, 139, 142, 144, 146
`gregexpr()`, 373
`gregexpr()`, 385
`grep()`, 37, 371, 372, 389
`grep()`, 385
`grid`, 356
`grid`, 139
`gsub()`, 37, 370, 371, 373, 379, 380, 388, 390
`gsub()`, 385

`haven`, 53
`head()`, 14, 21, 45, 73, 162
`head()`, 45
`header()`, 437
`help()`, 42
`help()`, 47
`hist()`, 6, 139
`HTML`, iv, v, 152, 172, 173, 187, 377, 378, 381, 399, 400, 405, 431, 433–435, 437–439, 441
`htmlParse()`, 431
 HTTP, v, 178, 380, 433, 438, 439, 441
`http://community.amstat.org/stats101/home`, 104
`httr`, 439

`iCDF()`, 316
`identical()`, 17, 18, 355
`identical()`, 45
`ifelse()`, 212, 219, 353, 355
`ifelse()`, 240
`image()`, 342–344, 450
`inner_join()`, 85
`integer`, 46, 61, 70
`is.atomic()`, 183
`is.atomic()`, 187
`is.datatype()`, 46
`is.factor()`, 13
`is.logical()`, 13
`is.logical()`, 46
`is.na()`, 13
`is.na()`, 46
`is.null()`, 13, 185
`is.null()`, 46
`isRe()`, 392

 Java, 4

JavaScript, 175, 437
`jitter()`, 140
`jkSE()`, 259, 261, 262, 264, 266
`jkTrimSE()`, 266
JSON, iii, 68, 153, 154, 175–178, 180, 181, 187, 406, 407, 431, 433, 434, 441
Kiva, 153
KML, 402

`lapply()`, 33, 160, 169, 181, 188, 189, 204, 222, 230, 417
`lapply()`, 188
`lattice`, 74, 146
`left_join()`, 85
`legend()`, 140
`length()`, 14, 21, 151, 156, 161–163, 414, 415
`length()`, 45
`levels()`, 13
`levels()`, 45
`libxml2`, 430
`lines()`, 140, 141
`list.files()`, 221
`list.files()`, 188
`lm()`, 24
`load()`, 39
`load()`, 47
`loess()`, 140, 141
`logical`, 15, 46
`logPlus()`, 214, 216, 217
`ls()`, 39
`ls()`, 47

`manySimTrim()`, 243
`map()`, 162
`mapply()`, 188, 189, 230, 309
`mapply()`, 188
`maps`, 162
`match()`, 342, 344
`Matlab`, 4
`matrix()`, 205, 206, 211
`matrix`, 59, 342
`max()`, 14, 17, 49
`mean()`, 14, 17, 48, 257, 275, 282
`median()`, 14, 17, 33
`merge()`, 82–85
`min()`, 14, 17
`minmax()`, 242
`minmaxCheckEmpty()`, 242
`minmaxNumeric()`, 242
Monte Carlo, 303

`mosaicplot()`, 139
`moveCars()`, 346–352, 354–357

`names()`, 14, 21, 151, 156, 161
`names()`, 45
`namespace`, 428
`nchar()`, 18, 388
`nchar()`, 45, 385
nearest neighbor, 280, 448, 449
New York Times, 177
New York Times, 177, 190
`newSqrt()`, 236
`nnPred()`, 282, 288, 289, 291, 292, 298
`numeric`, 15, 46, 55, 61, 104
`nzchar()`, 385

`objects()`, 39, 40, 47
`objects()`, 47
`one_of()`, 78
`Ops.factor()`, 273
`options()`, 20, 264
`options()`, 47
`order()`, 36, 66, 282, 450
`order()`, 46

`par()`, 140
`parent`, 426, 428
`paste()`, 385
PeMS, 52
Perl, 4
`plot()`, 24, 61, 139, 140, 207
`plot.BMLGrid()`, 343
`plotCI()`, 310
`plotRatio()`, 207
`plotrix`, 310
`pnorm()`, 47, 313, 323
`points()`, 140, 162
`polygon()`, 140
`POSIX()`, 199
`POSIXct`, 61, 86, 104
`POST`, 379, 438, 439
`postForm()`, 439, 440
`preceding`, 428
`preceding-sibling`, 426, 428
`predict()`, 141
`predNN()`, 292
`print()`, 19, 286, 342
`print()`, 45
`print.BMLGrid()`, 342
`prod()`, 17
`PUT`, 380

pxxx(), 313
Python, 4

q(), 39
qnorm(), 47, 313
qr(), 9
quantile(), 33, 49, 250
qxxx(), 313

R.matlab, 53
rbind(), 188
rbinom(), 313, 329
RBioFabric, 85
RColorBrewer, 137
RCurl, 174, 187, 439
RDocBook, 411
read.csv(), 55
read.dcf(), 58
read.dcf(), 103
read.delim(), 55
read.fwf(), 103
read.table(), 55, 103, 166, 437
read.table(), 103
read_csv(), 55
read_delim(), 55, 60, 76, 77, 103, 166, 167, 195, 278
read_delim(), 103
read_fwf(), 58, 63, 103
read_fwf(), 103
readBin(), 103
readHTMLTable(), 152, 172–174, 435, 437, 439
readHTMLTable(), 187, 431
readLines(), 166, 167, 195
readLines(), 188
readr, 55, 58, 63, 103
recover(), 288, 289, 340
rect(), 342
regexpr(), 372, 373, 383, 385
regexpr(), 385
remove(), 39, 40
remove(), 47
rep(), 36, 38, 44, 336, 340
rep(), 46
rep_len(), 340
replicate(), 227, 230, 233, 249
replicate(), 241
return(), 211, 212, 215, 218, 238, 263
return(), 241
returnZero1(), 241
returnZero2(), 241

rev(), 31, 36
rev(), 46
rexp(), 313
rexbound(), 325
rgeom(), 313
rhyper(), 313
rjson, 178, 182
RJSONIO, 178, 187
rjsonlite, 178, 182
rm(), 39, 47
rm(), 47
rmag(), 306, 308, 309
rmultinom(), 313
rnbnom(), 313
rnorm(), 42, 48, 49, 313, 318, 327
rnormalmixture(), 327, 333
rnormalmixture.faster(), 330
round(), 12
row(), 345
rpois(), 313
Rprof(), 272, 350
rrayleigh(), 308
RSS, 399
rtruncnorm(), 318, 320, 323, 325
runBML(), 350, 352, 357
runif(), 313, 318
RXQuery, 430

S, 43, 45
sadm(), 242
sample(), 6, 270, 274, 275, 296, 313, 320
sapply(), 33, 160, 161, 163, 169, 174, 179, 181, 185, 188, 189, 204, 205, 230, 256, 257, 327, 417, 450
sapply(), 46, 188
Saratoga County, 104
save(), 39, 207
save(), 47
save.image(), 39
save.image(), 47
scale(), 143
scale_color_manual(), 144
scale_linetype_manual(), 144
scale_xxx_xxx(), 145
scale_y_continuous(), 143
scatter plot, 342
sd(), 260
SDMX, 405, 406, 409, 422, 423
search(), 41
search(), 47
seed, 341

SELECT, v
`select()`, 78
`self`, 426, 428, 430
`seq()`, 35, 44
`seq()`, 46
SGML, 399
`showValueChange()`, 242
`simStudy()`, 329
`simSumDice()`, 242
`simTrim()`, 243
simulation, 303
`slice()`, 44
`sort()`, 36
`sort()`, 46
`source()`, 197, 203, 237, 341
`spread()`, 75, 77, 78
`SQL`, iv, viii
`sqrt()`, 9, 12
`squared()`, 241
`squaredNoCheck()`, 241
`stack()`, 73, 75
`stop()`, 216, 287, 293, 340
`stop()`, 241
`stopifnot()`, 341
`str()`, 62, 157, 162
`str()`, 187
`stripchart()`, 165
`strptime()`, 104
`strsplit()`, 37, 168, 170, 374
`strsplit()`, 385
`sub()`, 370, 371, 373
`sub()`, 385
`subset()`, 25, 32, 44
`subset()`, 46
`substr()`, 388
`substr()`, 385
`substring()`, 37, 373, 374, 437
`sum()`, 17, 207
`sumDice()`, 242
`summary()`, 15, 22, 44, 60, 63, 277
`summaryRProf()`, 352
`summaryRprof()`, 272, 350
survey, 95
SVG, 399, 402
`svytable()`, 95
`switch()`, 212, 220, 221
`switch()`, 240
`sys.call()`, 273
`system.time()`, 227, 267
`system.time()`, 240
`table()`, 277, 339, 402, 450
`tail()`, 14, 73, 435
`tail()`, 45
`tallyVotes()`, 283
`tapply()`, 33, 189
`tapply()`, 188
`test_that()`, 255, 257, 264
`testthat`, 255
`text()`, 140, 142
`tidyR`, 44, 75, 77, 78
`toJSON()`, 178
`toJSON()`, 187
`trace()`, 273
`traceback()`, 288
`trimMean()`, 243, 254, 256, 264, 265
`try()`, 434
`tryCatch()`, 294, 434
`unique()`, 189
UNIX, viii, 421
`unlist()`, 182
`unlist()`, 187
`untrace()`, 274
URI, 412, 429
USGS, 412, 414
vectorized, 345, 349, 352, 354, 355
VGAM, 307, 308
`voteNN()`, 283, 287, 292
`warning()`, 287
`warning()`, 241
`which()`, 31
`whisker.endpoints()`, 238
`winningStrat()`, 233
`with()`, 140, 165
`with()`, 103
`XHTML`, 402
`XML`, 172, 173, 187, 412, 414, 420, 421, 431, 437
`xmlAncestors()`, 419, 421
`xmlApply()`, 417
`xmlAttrs()`, 413, 420
`xmlAttrs()`, 431
`xmlChildren()`, 413, 417, 419, 421
`xmlChildren()`, 431
`xmlGetAttr()`, 417, 418, 420
`xmlGetAttr()`, 431
`XMLInternalDocument`, 413, 420
`XMLInternalElementNode`, 413, 415, 419
`XMLInternalNodeList`, 415

`xmlName()`, 413, 415, 420
`xmlNamespace()`, 420
`XMLNodeSet`, 422
`xmlParent()`, 419, 421
`xmlParse()`, 412, 413, 420
`xmlParse()`, 431
`xmlRoot()`, 413
`xmlRoot()`, 431
`xmlSApply()`, 417, 418
`xmlSize()`, 414, 420
`xmlToDataFrame()`, 431
`xmlToList()`, 413
`xmlValue()`, 413, 415, 420
XPath, iv, v, 399, 412, 413, 418, 419,
 421–430
 ., 428, 430
 ..., 428
 //, 427, 428
`ancestor`, 426, 428
`ancestor-or-self`, 426, 428
`attribute`, 428
`child`, 426, 428
`descendant`, 426, 428
`descendant-or-self`, 426–428
`following`, 428
`following-sibling`, 426, 428
`namespace`, 428
`parent`, 426, 428
`preceding`, 428
`preceding-sibling`, 426, 428
`self`, 426, 428, 430
XQuery, 430, 432
`xyplot()`, 74

`yLim()`, 242