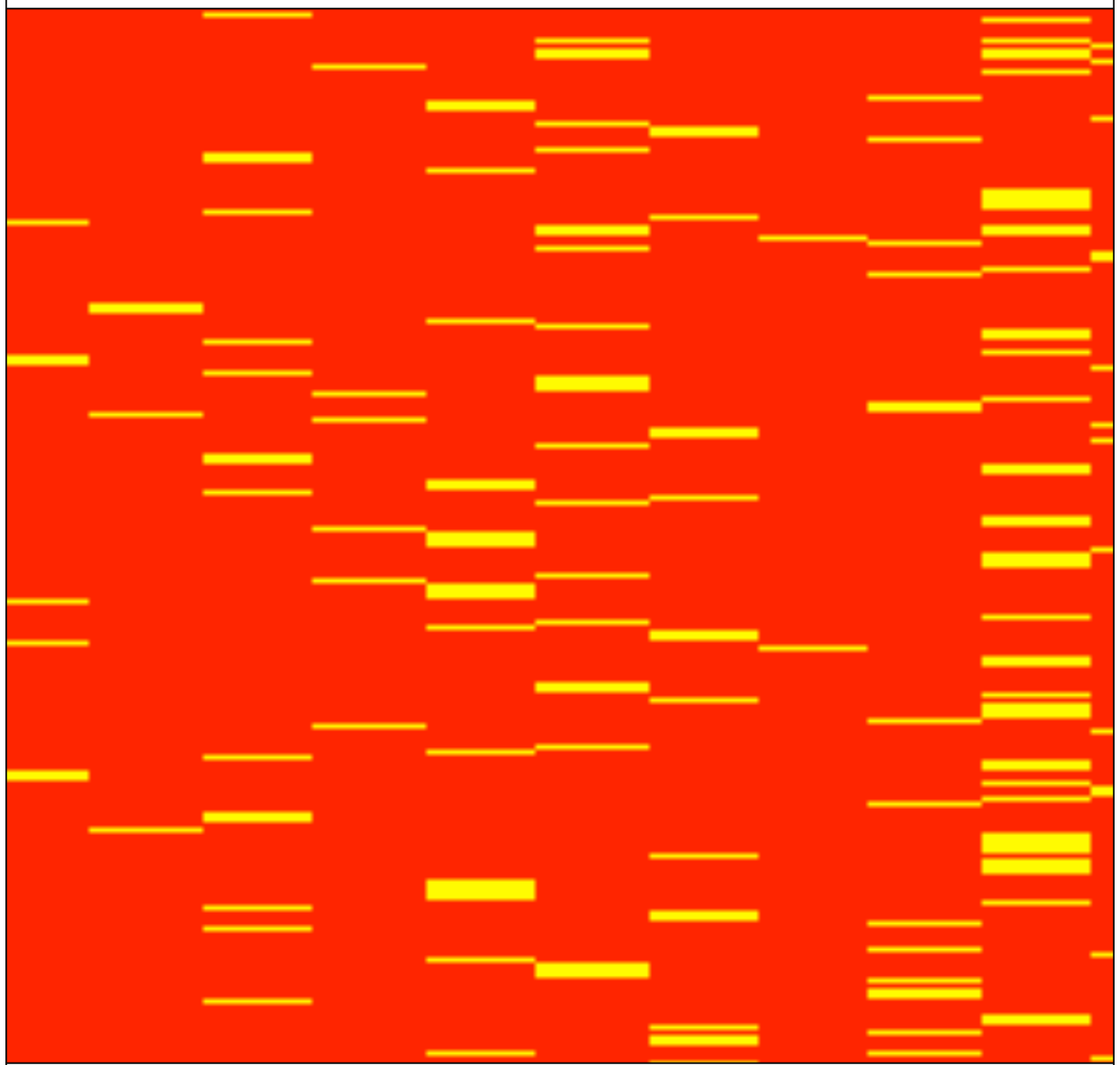
AN INTRODUCTION TO

Data Science



Jeffrey Stanton, Syracuse University

INTRODUCTION TO DATA SCIENCE

© 2012, Jeffrey Stanton

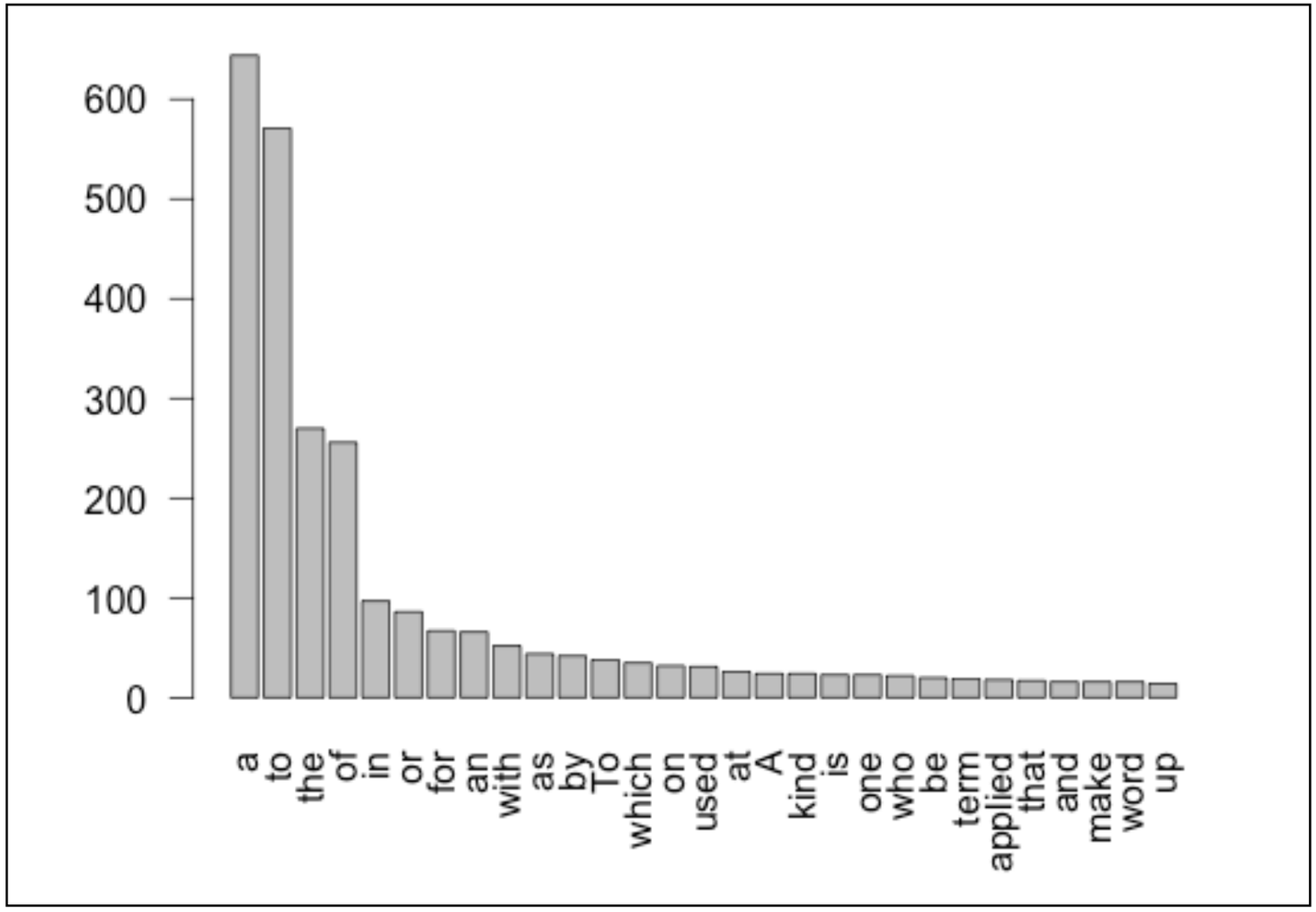
This book is distributed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 license. You are free to copy, distribute, and transmit this work. You are free to add or adapt the work. You must attribute the work to the author(s) listed above. You may not use this work or derivative works for commercial purposes. If you alter, transform, or build upon this work you may distribute the resulting work only under the same or similar license.

For additional details, please see:
http://creativecommons.org/licenses/by-nc-sa/3.0/

This book was developed for the Certificate of Data Science program at Syracuse University's School of Information Studies. If you find errors or omissions, please contact the author, Jeffrey Stanton, at jmstanto@syr.edu. A PDF version of this book and code examples used in the book are available at:

http://jsresearch.net/groups/teachdatascience

Data Science: Many Skills

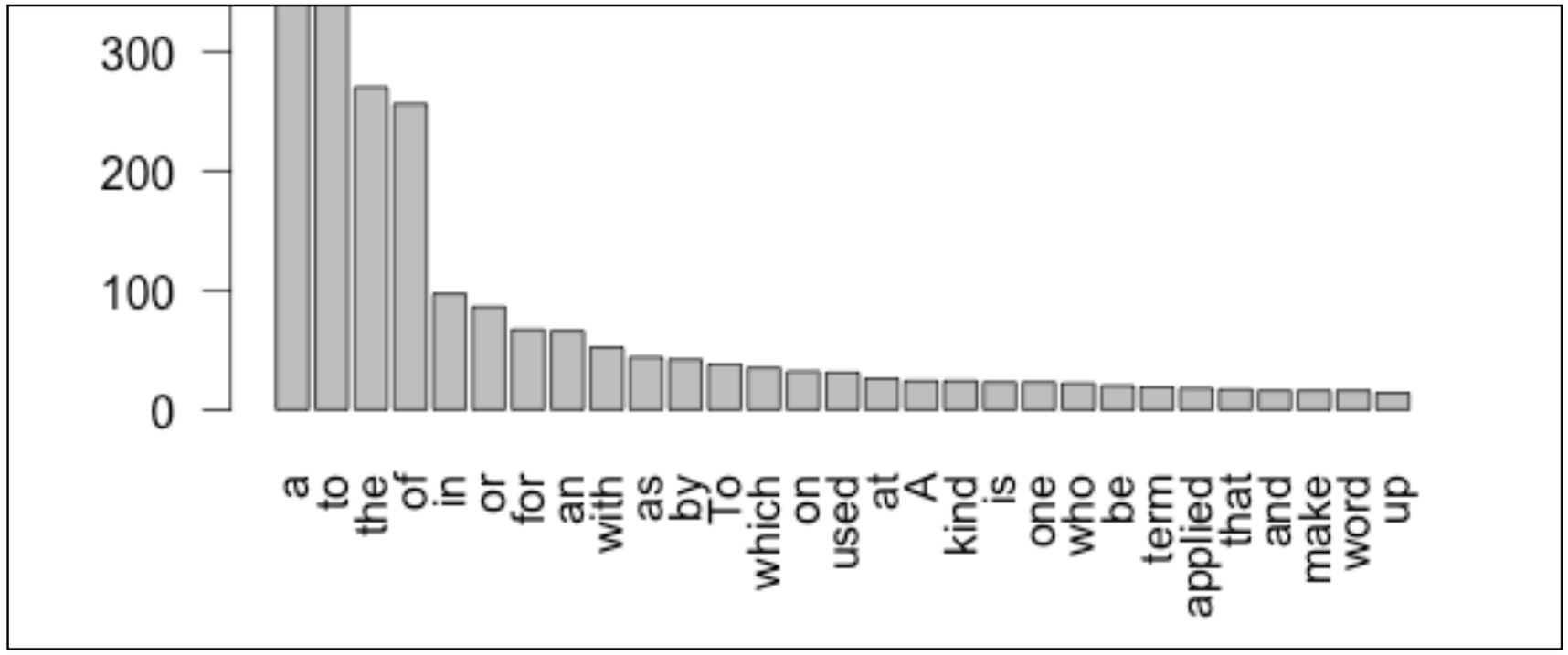


Data Science refers to an emerging area of work concerned with the collection, preparation, analysis, visualization, management, and preservation of large collections of information. Although the name Data Science seems to connect most strongly with areas such as databases and computer science, many different kinds of skills - including non-mathematical skills - are needed.

Data Science: Many Skills

Overview

1. Data science includes data analysis as an important component of the skill set required for many jobs in this area, but is not the only necessary skill.
2. A brief case study of a supermarket point of sale system illustrates the many challenges involved in data science work.
3. Data scientists play active roles in the design and implementation work of four related areas: data architecture, data acquisition, data analysis, and data archiving.
4. Key skills highlighted by the brief case study include communication skills, data analysis skills, and ethical reasoning skills.



Word frequencies from the definitions in a Shakespeare glossary. While professional data scientists do need skills with mathematics and statistics, much of the data in the world is unstructured and non-numeric.

For some, the term "Data Science" evokes images of statisticians in white lab coats staring fixedly at blinking computer screens filled with scrolling numbers. Nothing could be further from the truth. First of all, statisticians do not wear lab coats: this fashion statement is reserved for biologists, doctors, and others who have to keep their clothes clean in environments filled with unusual fluids. Second, much of the data in the world is non-numeric and unstructured. In this context, unstructured means that the data are not arranged in neat rows and columns. Think of a web page full of photographs and short messages among friends: very few numbers to work

with there. While it is certainly true that companies, schools, and governments use plenty of numeric information - sales of products, grade point averages, and tax assessments are a few examples - there is lots of other information in the world that mathematicians and statisticians look at and cringe. So, while it is always useful to have great math skills, there is much to be accomplished in the world of data science for those of us who are presently more comfortable working with words, lists, photographs, sounds, and other kinds of information.

In addition, data science is much more than simply analyzing data. There are many people who enjoy analyzing data and who could happily spend all day looking at histograms and averages, but for those who prefer other activities, data science offers a range of roles and requires a range of skills. Let's consider this idea by thinking about some of the data involved in buying a box of cereal.

Whatever your cereal preferences - fruity, chocolaty, fibrous, or nutty - you prepare for the purchase by writing "cereal" on your grocery list. Already your planned purchase is a piece of data, albeit a pencil scribble on the back on an envelope that only you can read. When you get to the grocery store, you use your data as a reminder to grab that jumbo box of FruityChocoBoms off the shelf and put it in your cart. At the checkout line the cashier scans the barcode on your box and the cash register logs the price. Back in the warehouse, a computer tells the stock manager that it is time to request another order from the distributor, as your purchase was one of the last boxes in the store. You also have a coupon for your big box and the cashier scans that, giving you a predetermined discount. At the end of the week, a report of all the scanned manufacturer coupons gets uploaded to the cereal company so that they can issue a reimbursement to the grocery store for all of the coupon discounts they have handed out to customers. Finally, at the end of the month, a store manager looks at a colorful collection of pie charts showing all of the different kinds of cereal that were sold, and on the basis of strong sales of fruity cereals, decides to offer more varieties of these on the store's limited shelf space next month.

So the small piece of information that began as a scribble on your grocery list ended up in many different places, but most notably on the desk of a manager as an aid to decision making. On the trip from your pencil to manager's desk, the data went through many transformations. In addition to the computers where the data may have stopped by or stayed on for the long term, lots of other pieces of hardware - such as the barcode scanner - were involved in collecting, manipulating, transmitting, and storing the data. In addition, many different pieces of software were used to organize, aggregate, visualize, and present the data. Finally, many different "human systems" were involved in working with the data. People decided which systems to buy and install, who should get access to what kinds of data, and what would happen to the data after its immediate purpose was fulfilled. The personnel of the grocery chain and its partners made a thousand other detailed decisions and negotiations before the scenario described above could become reality.

Obviously data scientists are not involved in all of these steps. Data scientists don't design and build computers or barcode readers, for instance. So where would the data scientists play the most valuable role? Generally speaking, data scientists play the most active roles in the four A's of data: data architecture, data acquisition,

data analysis, and data archiving. Using our cereal example, let's look at them one by one. First, with respect to architecture, it was important in the design of the "point of sale" system (what retailers call their cash registers and related gear) to think through in advance how different people would make use of the data coming through the system. The system architect, for example, had a keen appreciation that both the stock manager and the store manager would need to use the data scanned at the registers, albeit for somewhat different purposes. A data scientist would help the system architect by providing input on how the data would need to be routed and organized to support the analysis, visualization, and presentation of the data to the appropriate people.

Next, acquisition focuses on how the data are collected, and, importantly, how the data are represented prior to analysis and presentation. For example, each barcode represents a number that, by itself, is not very descriptive of the product it represents. At what point after the barcode scanner does its job should the number be associated with a text description of the product or its price or its net weight or its packaging type? Different barcodes are used for the same product (for example, for different sized boxes of cereal). When should we make note that purchase X and purchase Y are the same product, just in different packages? Representing, transforming, grouping, and linking the data are all tasks that need to occur before the data can be profitably analyzed, and these are all tasks in which the data scientist is actively involved.

The analysis phase is where data scientists are most heavily involved. In this context we are using analysis to include summarization of the data, using portions of data (samples) to make inferences about the larger context, and visualization of the data by presenting it in tables, graphs, and even animations. Although there are many technical, mathematical, and statistical aspects to these activities, keep in mind that the ultimate audience for data analysis is always a person or people. These people are the "data users" and fulfilling their needs is the primary job of a data scientist. This point highlights the need for excellent communication skills in data science. The most sophisticated statistical analysis ever developed will be useless unless the results can be effectively communicated to the data user.

Finally, the data scientist must become involved in the archiving of the data. Preservation of collected data in a form that makes it highly reusable - what you might think of as "data curation" - is a difficult challenge because it is so hard to anticipate all of the future uses of the data. For example, when the developers of Twitter were working on how to store tweets, they probably never anticipated that tweets would be used to pinpoint earthquakes and tsunamis, but they had enough foresight to realize that "geocodes" - data that shows the geographical location from which a tweet was sent - could be a useful element to store with the data.

All in all, our cereal box and grocery store example helps to highlight where data scientists get involved and the skills they need. Here are some of the skills that the example suggested:

- Learning the application domain - The data scientist must quickly learn how the data will be used in a particular context.
- Communicating with data users - A data scientist must possess strong skills for learning the needs and preferences of users. Translating back and forth between the technical terms of com-

puting and statistics and the vocabulary of the application domain is a critical skill.

- Seeing the big picture of a complex system - After developing an understanding of the application domain, the data scientist must imagine how data will move around among all of the relevant systems and people.
- Knowing how data can be represented - Data scientists must have a clear understanding about how data can be stored and linked, as well as about "metadata" (data that describes how other data are arranged).
- Data transformation and analysis - When data become available for the use of decision makers, data scientists must know how to transform, summarize, and make inferences from the data. As noted above, being able to communicate the results of analyses to users is also a critical skill here.
- Visualization and presentation - Although numbers often have the edge in precision and detail, a good data display (e.g., a bar chart) can often be a more effective means of communicating results to data users.
- Attention to quality - No matter how good a set of data may be, there is no such thing as perfect data. Data scientists must know the limitations of the data they work with, know how to quantify its accuracy, and be able to make suggestions for improving the quality of the data in the future.
- Ethical reasoning - If data are important enough to collect, they are often important enough to affect people's lives. Data scientists must understand important ethical issues such as privacy, and must be able to communicate the limitations of data to try to prevent misuse of data or analytical results.

The skills and capabilities noted above are just the tip of the iceberg, of course, but notice what a wide range is represented here. While a keen understanding of numbers and mathematics is important, particularly for data analysis, the data scientist also needs to have excellent communication skills, be a great systems thinker, have a good eye for visual displays, and be highly capable of thinking critically about how data will be used to make decisions and affect people's lives. Of course there are very few people who are good at all of these things, so some of the people interested in data will specialize in one area, while others will become experts in another area. This highlights the importance of teamwork, as well.

In this Introduction to Data Science eBook, a series of data problems of increasing complexity is used to illustrate the skills and capabilities needed by data scientists. The open source data analysis program known as "R" and its graphical user interface companion "R-Studio" are used to work with real data examples to illustrate both the challenges of data science and some of the techniques used to address those challenges. To the greatest extent possible, real datasets reflecting important contemporary issues are used as the basis of the discussions.

No one book can cover the wide range of activities and capabilities involved in a field as diverse and broad as data science. Throughout the book references to other guides and resources provide the interested reader with access to additional information. In the open source spirit of "R" and "R Studio" these are, wherever possible, web-based and free. In fact, one of guides that appears most fre-

quently in these pages is "Wikipedia," the free, online, user sourced encyclopedia. Although some teachers and librarians have legitimate complaints and concerns about Wikipedia, and it is admittedly not perfect, it is a very useful learning resource. Because it is free, because it covers about 50 times more topics than a printed encyclopedia, and because it keeps up with fast moving topics (like data science) better than printed encyclopedias, Wikipedia is very useful for getting a quick introduction to a topic. You can't become an expert on a topic by only consulting Wikipedia, but you can certainly become smarter by starting there.

Another very useful resource is Khan Academy. Most people think of Khan Academy as a set of videos that explain math concepts to middle and high school students, but thousands of adults around the world use Khan Academy as a refresher course for a range of topics or as a quick introduction to a topic that they never studied before. All of the lessons at Khan Academy are free, and if you log in with a Google or Facebook account you can do exercises and keep track of your progress.

At the end of each chapter of this book, a list of Wikipedia sources and Khan Academy lessons (and other resources too!) shows the key topics relevant to the chapter. These sources provide a great place to start if you want to learn more about any of the topics that chapter does not explain in detail.

Obviously if you are reading this book you probably have access to an iBook reader app, probably on an iPad or other Apple device. You can also access this book as a PDF on the book's website: **TBD**. It is valuable to have access to the Internet while you are reading, so that you can follow some of the many links this book provides.

Also, as you move into the sections in the book where open source software such as the R data analysis system is used, you will sometimes need to have access to a desktop or laptop computer where you can run these programs.

One last thing: The book presents topics in an order that should work well for people with little or no experience in computer science or statistics. If you already have knowledge, training, or experience in one or both of these areas, you should feel free to skip over some of the introductory material and move right into the topics and chapters that interest you most. There's something here for everyone and, after all, you can't beat the price!

Sources

http://en.wikipedia.org/wiki/E-Science

http://en.wikipedia.org/wiki/E-Science_librarianship

http://en.wikipedia.org/wiki/Wikipedia:Size_comparisons

http://en.wikipedia.org/wiki/Statistician

http://en.wikipedia.org/wiki/Visualization_(computer_graphics)

http://www.khanacademy.org/

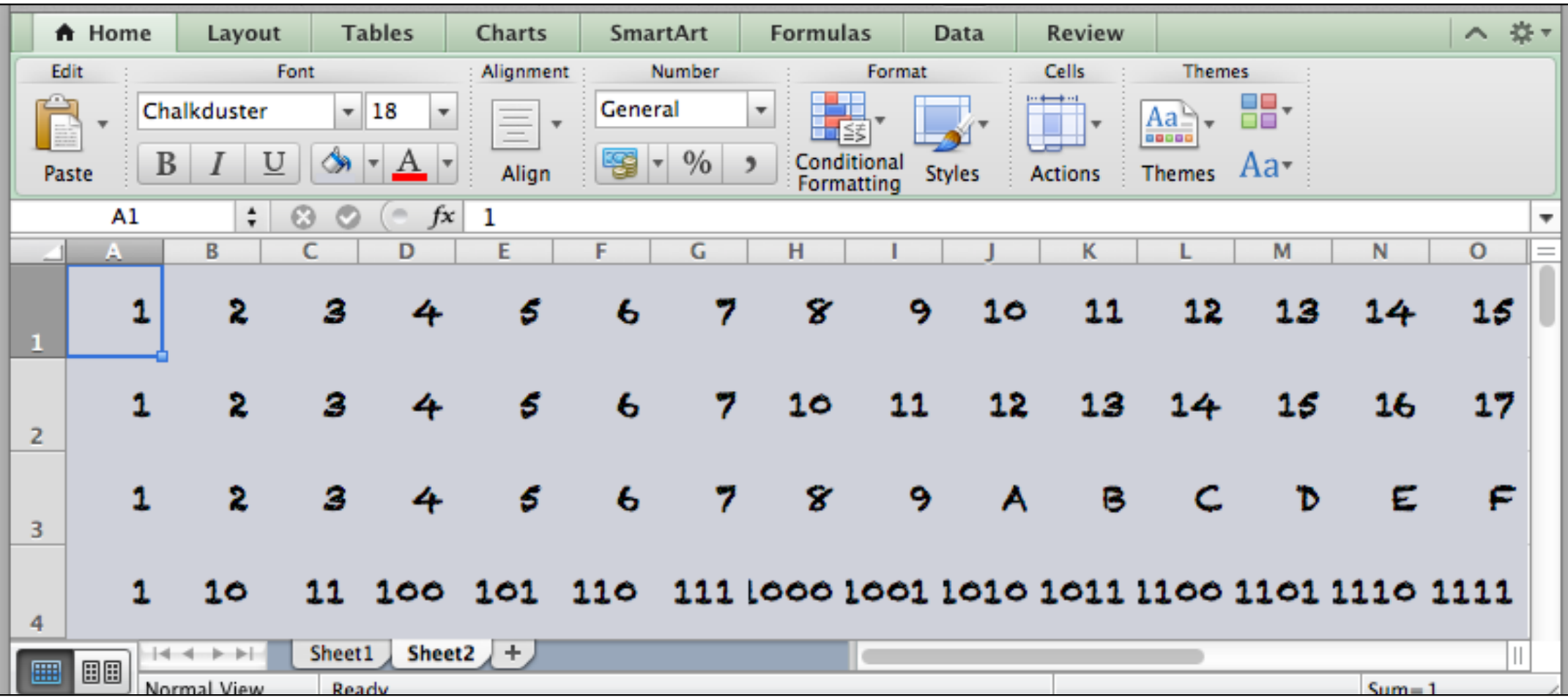
http://www.r-project.org/

http://www.readwriteweb.com/hack/2011/09/unlocking-big-data-with-r.php

http://rstudio.org/

CHAPTER 1

About Data



Data comes from the Latin word, "datum," meaning a "thing given." Although the term "data" has been used since as early as the 1500s, modern usage started in the 1940s and 1950s as practical electronic computers began to input, process, and output data. This chapter discusses the nature of data and introduces key concepts for newcomers without computer science experience.

The inventor of the World Wide Web, Tim Berners-Lee, is often quoted as having said, "Data is not information, information is not knowledge, knowledge is not understanding, understanding is not wisdom." This quote suggests a kind of pyramid, where data are the raw materials that make up the foundation at the bottom of the pile, and information, knowledge, understanding and wisdom represent higher and higher levels of the pyramid. In one sense, the major goal of a data scientist is to help people to turn data into information and onwards up the pyramid. Before getting started on this goal, though, it is important to have a solid sense of what data actually are. (Notice that this book treats the word "data" as a plural noun - in common usage you may often hear it referred to as singular instead.) If you have studied computer science or mathematics, you may find the discussion in this chapter a bit redundant, so feel free to skip it. Otherwise, read on for an introduction to the most basic ingredient to the data scientist's efforts: data.

A substantial amount of what we know and say about data in the present day comes from work by a U.S. mathematician named Claude Shannon. Shannon worked before, during, and after World War II on a variety of mathematical and engineering problems related to data and information. Not to go crazy with quotes, or anything, but Shannon is quoted as having said, "The fundamental problem of communication is that of reproducing at one point either exactly or approximately a message selected at another point." This quote helpfully captures key ideas about data that are important in this book by focusing on the idea of data as a message that moves from a source to a recipient. Think about the simplest possible message that you could send to another person over the phone, via a text message, or even in person. Let's say that a friend had asked you a question, for example whether you wanted to come to their house for dinner the next day. You can answer yes or no. You can call the person on the phone, and say yes or no. You might have a bad connection, though, and your friend might not be able to hear you. Likewise, you could send them a text message with your answer, yes or no, and hope that they have their phone turned on so that they can receive the message. Or you could tell your friend face to face, hoping that she did not have her earbuds turned up so loud that she couldn't hear you. In all three cases you have a one "bit" message that you want to send to your friend, yes or no, with the goal of "reducing her uncertainty" about whether you will appear at her house for dinner the next day. Assuming that message gets through without being garbled or lost, you will have successfully transmitted one bit of information from you to her. Claude Shannon developed some mathematics, now often referred to as "Information Theory," that carefully quantified how bits of data transmitted accurately from a source to a recipient can reduce uncertainty by providing information. A great deal of the computer networking equipment and software in the world today - and especially the huge linked worldwide network we call the Internet - is primarily concerned with this one basic task of getting bits of information from a source to a destination.

Once we are comfortable with the idea of a "bit" as the most basic unit of information, either "yes" or "no," we can combine bits together to make more complicated structures. First, let's switch labels just slightly. Instead of "no" we will start using zero, and instead of "yes" we will start using one. So we now have a single digit, albeit one that has only two possible states: zero or one (we're temporarily making a rule against allowing any of the bigger digits like three or seven). This is in fact the origin of the word "bit," which is a squashed down version of the phrase "Binary

digIT." A single binary digit can be 0 or 1, but there is nothing stopping us from using more than one binary digit in our messages. Have a look at the example in the table below:

MEANING	2ND DIGIT	1ST DIGIT
No	0	0
Maybe	0	1
Probably	1	0
Definitely	1	1

Here we have started to use two binary digits - two bits - to create a "code book" for four different messages that we might want to transmit to our friend about her dinner party. If we were certain that we would not attend, we would send her the message 0 0. If we definitely planned to attend we would send her 1 1. But we have two additional possibilities, "Maybe" which is represented by 0 1, and "Probably" which is represented by 1 0. It is interesting to compare our original yes/no message of one bit with this new four-option message with two bits. In fact, every time you add a new bit you double the number of possible messages you can send. So three bits would give eight options and four bits would give 16 options. How many options would there be for five bits?

When we get up to eight bits - which provides 256 different combinations - we finally have something of a reasonably useful size to work with. Eight bits is commonly referred to as a "byte" - this term probably started out as a play on words with the word bit. (Try looking up the word "nybble" online!) A byte offers enough different combinations to encode all of the letters of the alphabet, including capital and small letters. There is an old rulebook called "ASCII" - the American Standard Code for Information Interchange - which matches up patterns of eight bits with the letters of the alphabet, punctuation, and a few other odds and ends. For example the bit pattern 0100 0001 represents the capital letter A and the next higher pattern, 0100 0010, represents capital B. Try looking up an ASCII table online (for example, http://www.asciitable.com/) and you can find all of the combinations. Note that the codes may not actually be shown in binary because it is so difficult for people to read long strings of ones and zeroes. Instead you may see the equivalent codes shown in hexadecimal (base 16), octal (base 8), or the most familiar form that we all use everyday, base 10. Although you might remember base conversions from high school math class, it would be a good idea to practice this a little bit - particularly the conversions between binary, hexadecimal, and decimal (base 10). You might also enjoy Vi Hart's "Binary Hand Dance" video at Khan Academy (search for this at http://www.khanacademy.org or follow the link at the end of the chapter). Most of the work we do in this book will be in decimal, but more complex work with data often requires understanding hexadecimal and being able to know how a hexadecimal number, like 0xA3, translates into a bit pattern. Try searching online for "binary conversion tutorial" and you will find lots of useful sites.

Combining Bytes into Larger Structures

Now that we have the idea of a byte as a small collection of bits (usually eight) that can be used to store and transmit things like letters and punctuation marks, we can start to build up to bigger and better things. First, it is very easy to see that we can put bytes to-

gether into lists in order to make a "string" of letters, what is often referred to as a "character string." If we have a piece of text, like "this is a piece of text" we can use a collection of bytes to represent it like this:

0111010001101000011010010111001100100000011010010111001100100000011000010010000001110000011010010110010101100011011001010010000001101111011001100010000001110100011001010111100001110100

Now nobody wants to look at that, let alone encode or decode it by hand, but fortunately, the computers and software we use these days takes care of the conversion and storage automatically. For example, when we tell the open source data language "R" to store "this is a piece of text" for us like this:

```
myText <- "this is a piece of text"
```

...we can be certain that inside the computer there is a long list of zeroes and ones that represent the text that we just stored. By the way, in order to be able to get our piece of text back later on, we have made a kind of storage label for it (the word "myText" above). Anytime that we want to remember our piece of text or use it for something else, we can use the label "myText" to open up the chunk of computer memory where we have put that long list of binary digits that represent our text. The left-pointing arrow made up out of the less-than character ("<") and the dash character ("-") gives R the command to take what is on the right hand side (the quoted text) and put it into what is on the left hand side (the storage area we have labeled "myData"). Some people call this the assignment arrow and it is used in some computer languages to make it clear to the human who writes or reads it which direction the information is flowing.

From the computer's standpoint, it is even simpler to store, remember, and manipulate numbers instead of text. Remember than an eight bit byte can hold 256 combinations, so just using that very small amount we could store the numbers from 0 to 255. (Of course, we could have also done 1 to 256, but much of the counting and numbering that goes on in computers starts with zero instead of one.) Really, though, 255 is not much to work with. We couldn't count the number of houses in most towns or the number of cars in a large parking garage unless we can count higher than 255. If we put together two bytes to make 16 bits we can count from zero up to up to 65,535, but that is still not enough for some of the really big numbers in the world today (for example, there are more than 200 million cars in the U.S. alone). Most of the time, if we want to be flexible in representing an integer (a number with no decimals), we use four bytes stuck together. Four bytes stuck together is a total of 32 bits, and that allows us to store an integer as high as 4,294,967,295.

Things get slightly more complicated when we want to store a negative number or a number that has digits after the decimal point. If you are curious, try looking up "two's complement" for more information about how signed numbers are stored and "floating point" for information about how numbers with digits after the decimal point are stored. For our purposes in this book, the most important thing to remember is that text is stored differently than numbers, and among numbers integers are stored differently than floating point. Later we will find that it is sometimes necessary to

convert between these different representations, so it is always important to know how it is represented.

So far we have mainly looked at how to store one thing at a time, like one number or one letter, but when we are solving problems with data we often need to store a group of related things together. The simplest place to start is with a list of things that are all stored in the same way. For example, we could have a list of integers, where each thing in the list is the age of a person in your family. The list might look like this: 43, 42, 12, 8, 5. The first two numbers are the ages of the parents and the last three numbers are the ages of the kids. Naturally, inside the computer each number is stored in binary, but fortunately we don't have to type them in that way or look at them that way. Because there are no decimal points, these are just plain integers and a 32 bit integer (4 bytes) is more than enough to store each one. This list contains items that are all the same "type" or "mode." The open source data program "R" refers to a list where all of the items are of the same mode as a "vector." We can create a vector with R very easily by listing the numbers, separated by commas and inside parentheses:

```
c(43, 42, 12, 8, 5)
```

The letter "c" in front of the opening parenthesis stands for concatenate, which means to join things together. Slightly obscure, but easy enough to get used to with some practice. We can also put in some of what we learned a couple of days ago to store our vector in a named location (remember that a vector is list of items of the same mode/type):

```
myFamilyAges <- c(43, 42, 12, 8, 5)
```

We have just created our first "data set." It is very small, for sure, only five items, but also very useful for illustrating several major concepts about data. Here's a recap:

- In the heart of the computer, all data are represented in binary. One binary digit, or bit, is the smallest chunk of data that we can send from one place to another.
- Although all data are at heart binary, computers and software help to represent data in more convenient forms for people to see. Three important representations are: "character" for representing text, "integer" for representing numbers with no digits after the decimal point, and "floating point" for numbers that may have digits after the decimal point. The list of numbers in our tiny data set just above are integers.
- Numbers and text can be collected into lists, which the open source program "R" calls vectors. A vector has a length, which is the number of items in it, and a "mode" which is the type of data stored in the vector. The vector we were just working on has a length of 5 and a mode of integer.
- In order to be able to remember where we stored a piece of data, most computer programs, including R, give us a way of labeling a chunk of computer memory. We chose to give the 5-item vector up above the name "myFamilyAges." Some people might refer to this named list as a "variable," because the value of it varies, depending upon which member of the list you are examining.
- If we gather together one or more variables into a sensible group, we can refer to them together as a "data set." Usually, it doesn't make sense to refer to something with just one variable

as a data set, so usually we need at least two variables. Technically, though, even our very simple "myFamilyAges" counts as a data set, albeit a very tiny one.

In the next chapter we will install and run the open source "R" data program and learn more about how to create data sets, summarize the information in those data sets, and perform some simple calculations and transformations on those data sets.

Chapter Challenge

Discover the meaning of "Boolean Logic" and the rules for "and", "or", "not", and "exclusive or". Once you have studied this for a whole, write down on a piece of paper, without looking, all of the binary operations that demonstrate these rules.

Sources

http://en.wikipedia.org/wiki/Claude_Shannon

http://en.wikipedia.org/wiki/Information_theory

http://cran.r-project.org/doc/manuals/R-intro.pdf

http://www.khanacademy.org/math/vi-hart/v/binary-hand-dance

http://www.khanacademy.org/science/computer-science/v/introduction-to-programs-data-types-and-variables

http://www.asciitable.com/

Test Yourself

Review 1.1 About Data

Question 1 of 3

The smallest unit of information commonly in use in today's computers is called:

- ✓ **A.** A Bit
- **B.** A Byte
- **C.** A Nybble
- **D.** An Integer

Check Answer

CHAPTER 2

Identifying Data Problems

Data Science is different from other areas such as mathematics of statistics. Data Science is an applied activity and data scientists serve the needs and solve the problems of data users. Before you can solve a problem, you need to identify it and this process is not always as obvious as it might seem. In this chapter, we discuss the identification of data problems.

Apple farmers live in constant fear, first for their blossoms and later for their fruit. A late spring frost can kill the blossoms. Hail or extreme wind in the summer can damage the fruit. More generally, farming is an activity that is first and foremost in the physical world, with complex natural processes and forces, like weather, that are beyond the control of humankind.

In this highly physical world of unpredictable natural forces, is there any role for data science? On the surface there does not seem to be. But how can we know for sure? Having a nose for identifying data problems requires openness, curiosity, creativity, and a willingness to ask a lot of questions. In fact, if you took away from the first chapter the impression that a data scientist sits in front a of computer all day and works a crazy program like R, that is a mistake. Every data scientist must (eventually) become immersed in the problem domain where she is working. The data scientist may never actually become a farmer, but if you are going to identify a data problem that a farmer has, you have to learn to think like a farmer, to some degree.

To get this domain knowledge you can read or watch videos, but the best way is to ask "subject matter experts" (in this case farmers) about what they do. The whole process of asking questions deserves its own treatment, but for now there are three things to think about when asking questions. First, you want the subject matter experts, or SMEs, as they are sometimes called, to tell stories of what they do. Then you want to ask them about anomalies: the unusual things that happen for better or for worse. Finally, you want to ask about risks and uncertainty: what are the situations where it is hard to tell what will happen next - and what happens next could have a profound effect on whether the situation ends badly or well. Each of these three areas of questioning reflects an approach to identifying data problems that may turn up something good that could be accomplished with data, information, and the right decision at the right time.

The purpose of asking about stories is that people mainly think in stories. From farmers to teachers to managers to CEOs, people know and tell stories about success and failure in their particular domain. Stories are powerful ways of communicating wisdom between different members of the same profession and they are ways of collecting a sense of identity that sets one profession apart from another profession. The only problem is that stories can be wrong.

If you can get a professional to tell the main stories that guide how she conducts her work, you can then consider how to verify that story. Without questioning the veracity of the person that tells the story, you can imagine ways of measuring the different aspects of how things happen in the story with an eye towards eventually verifying (or sometimes debunking) the stories that guide professional work.

For example, the farmer might say that in the deep spring frost that occurred five years ago, the trees in the hollow were spared frost damage while the trees around the ridge of the hill had more damage. For this reason, on a cold night the farmer places most of the smudgepots (containers that hold a fuel that creates a smoky fire) around the ridge. The farmer strongly believes that this strategy works, but does it? It would be possible to collect time-series temperature data from multiple locations within the orchard on cold and warm nights, and on nights with and without smudgepots. The data could be used to create a model of temperature

changes in the different areas of the orchard and this model could support, improve, or debunk the story.

A second strategy for problem identification is to look for the exception cases, both good and bad. A little later in the book we will learn about how the core of classic methods of statistical inference is to characterize "the center" - the most typical cases that occur - and then examine the extreme cases that are far from the center for information that could help us understand an intervention or an unusual combination of circumstances. Identifying unusual cases is a powerful way of understanding how things work, but it is necessary first to define the central or most typical occurrences in order to have an accurate idea of what constitutes an unusual case.

Coming back to our farmer friend, in advance of a thunderstorm late last summer, a powerful wind came through the orchard, tearing the fruit off the trees. Most of the trees lost a small amount of fruit: the dropped apples could be seen near the base of the tree. One small grouping of trees seemed to lose a much larger amount of fruit, however, and the drops were apparently scattered much further from the trees. Is it possible that some strange wind conditions made the situation worse in this one spot? Or is just a matter of chance that a few trees in the same area all lost a bit more fruit than would be typical.

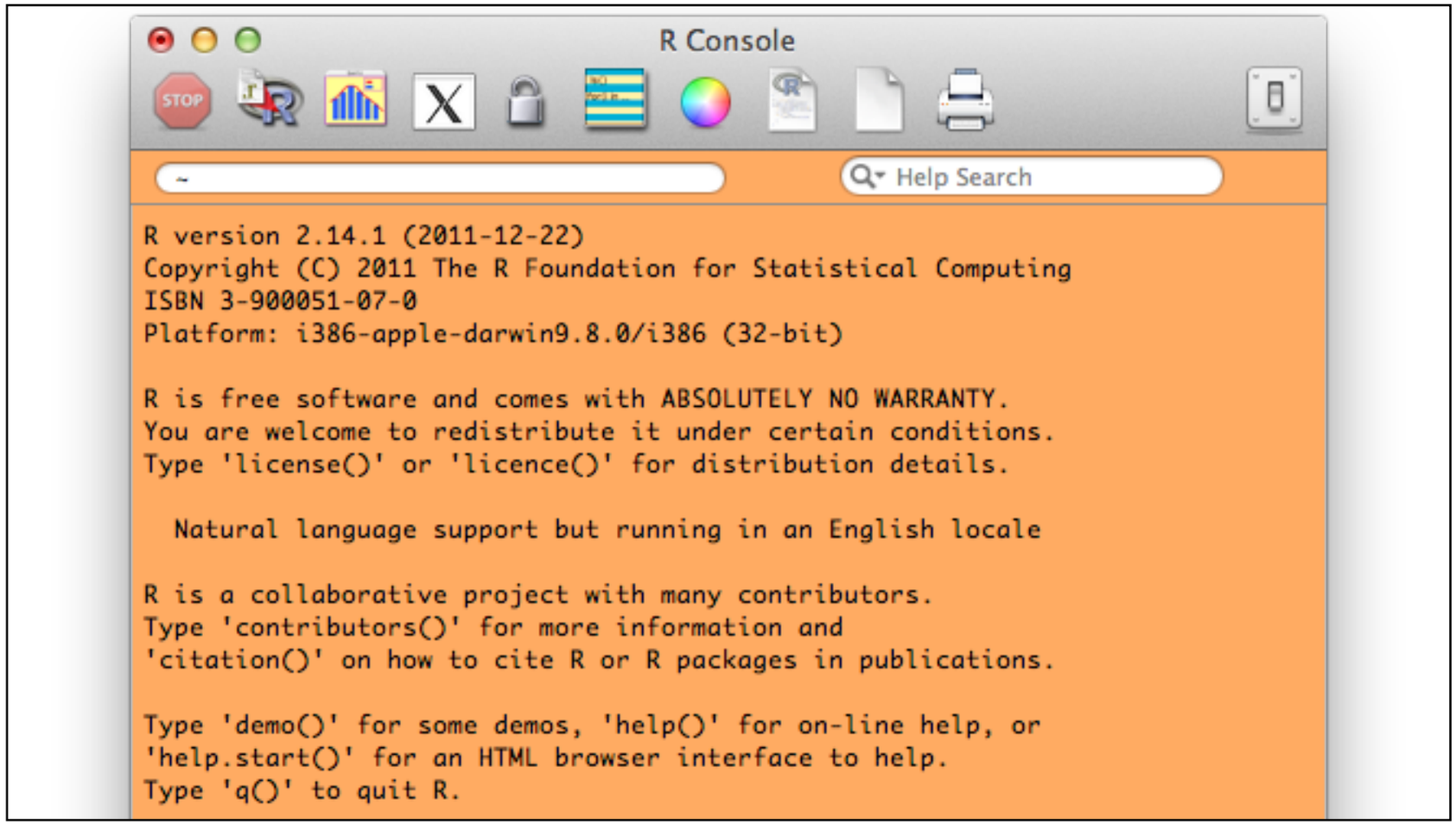
A systematic count of lost fruit underneath a random sample of trees would help to answer this question. The bulk of the trees would probably have each lost about the same amount, but more importantly, that "typical" group would give us a yardstick against which we could determine what would really count as unusual. When we found an unusual set of cases that was truly beyond the limits of typical, we could rightly focus our attention on these to try to understand the anomaly.

A third strategy for identifying data problems is to find out about risk and uncertainty. If you read the previous chapter you may remember that a basic function of information is to reduce uncertainty. It is often valuable to reduce uncertainty because of how risk affects the things we all do. At work, at school, at home, life is full of risks: making a decision or failing to do so sets off a chain of events that may lead to something good or something not so good. It is difficult to say, but in general we would like to narrow things down in a way that maximizes the chances of a good outcome and minimizes the chance of a bad one. To do this, we need to make better decisions and to make better decisions we need to reduce uncertainty. By asking questions about risks and uncertainty (and decisions) a data scientist can zero in on the problems that matter. You can even look at the previous two strategies - asking about the stories that comprise professional wisdom and asking about anomalies/unusual cases - in terms of the potential for reducing uncertainty and risk.

In the case of the farmer, much of the risk comes from the weather, and the uncertainty revolves around which countermeasures will be cost effective under prevailing conditions. Consuming lots of expensive oil in smudgepots on a night that turns out to be quite warm is a waste of resources that could make the difference between a profitable or an unprofitable year. So more precise and timely information about local weather conditions might be a key focus area for problem solving with data. What if a live stream of national weather service doppler radar could appear on the farmer's smart phone? Let's build an app for that...

CHAPTER 3

Getting Started with R



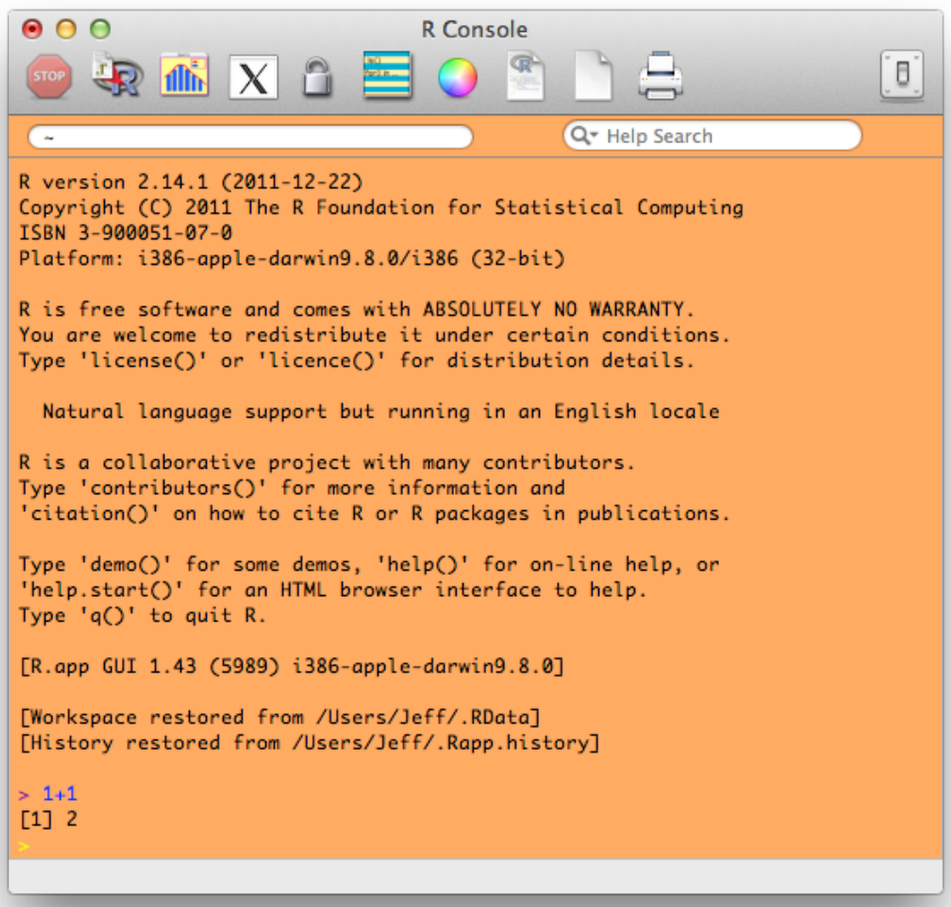
"R" is an open source software program, developed by volunteers as a service to the community of scientists, researchers, and data analysts who use it. R is free to download and use. Lots of advice and guidance is available online to help users learn R, which is good because it is a powerful and complex program, in reality a full featured programming language dedicated to data.

If you are new to computers, programming, and/or data science welcome to an exciting chapter that will open the door to the most powerful free data analytics tool ever created anywhere in the universe, no joke. On the other hand, if you are experienced with spreadsheets, statistical analysis, or accounting software you are probably thinking that this book has now gone off the deep end, never to return to sanity and all that is good and right in user interface design. Both perspectives are reasonable. The "R" open source data analysis program is immensely powerful, flexible, and especially "extensible" (meaning that people can create new capabilities for it quite easily). At the same time, R is "command line" oriented, meaning that most of the work that one needs to perform is done through carefully crafted text instructions, many of which have tricky syntax (the punctuation and related rules for making a command that works). In addition, R is not especially good at giving feedback or error messages that help the user to repair mistakes or figure out what is wrong when results look funny.

But there is a method to the madness here. One of the virtues of R as a teaching tool is that it hides very little. The successful user must fully understand what the "data situation" is or else the R commands will not work. With a spreadsheet, it is easy to type in a lot of numbers and a formula like =FORECAST() and a result pops into a cell like magic, whether it makes any sense or not. With R you have to know your data, know what you can do it, know how it has to be transformed, and know how to check for problems. Because R is a programming language, it also forces users to think about problems in terms of data objects, methods that can be applied to those objects, and procedures for applying those methods. These are important metaphors used in modern programming languages, and no data scientist can succeed without having at least a rudimentary understanding of how software is programmed, tested, and integrated into working systems. The extensibility of R means that new modules are being added all the time by volunteers: R was among the first analysis program to integrate capabilities for drawing data directly from the Twitter(r) social media platform. So you can be sure that whatever the next big development is in the world of data, that someone in the R community will start to develop a new "package" for R that will make use of it. Finally, the lessons one learns in working with R are almost universally applicable to other programs and environments. If one has mastered R, it is a relatively small step to get the hang of the SAS(r) statistical programming language and an even smaller step to being able to follow SPSS(r) syntax. (SAS and SPSS are two of the most widely used commercial statistical analysis programs). So with no need for any licensing fees paid by school, student, or teacher it is possible to learn the most powerful data analysis system in the universe and take those lessons with you no matter where you go. It will take a bit patience though, so please hang in there!

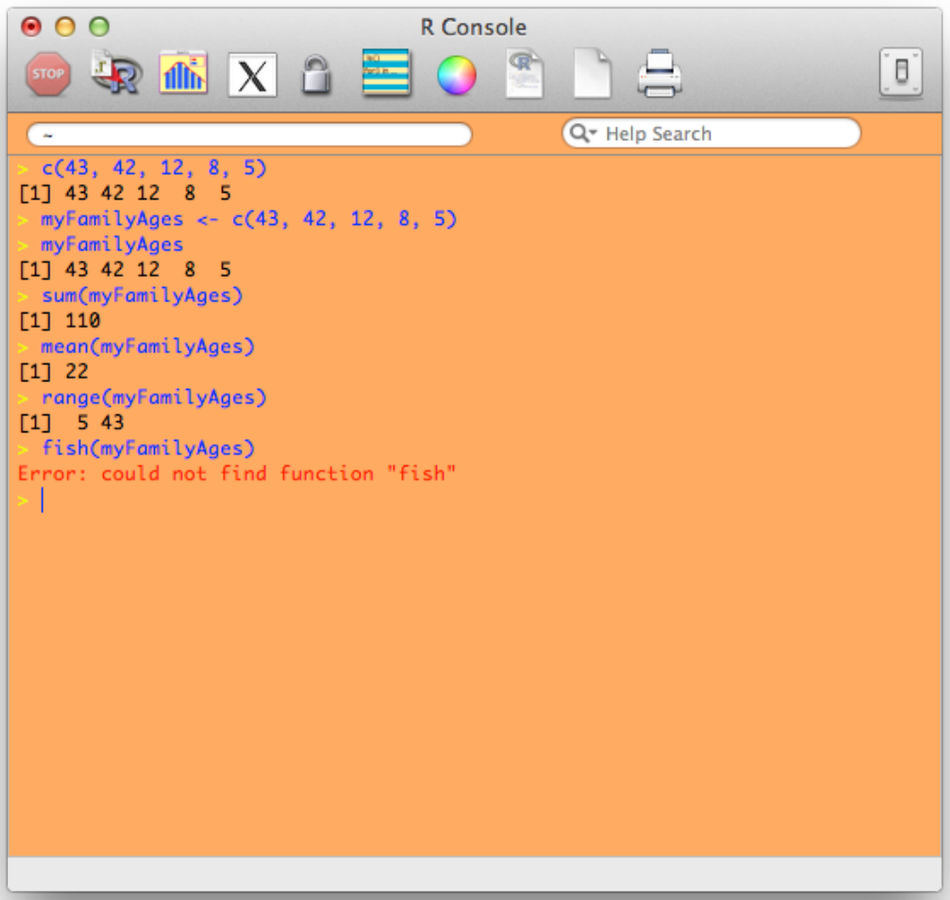
Let's get started. Obviously you will need a computer. If you are working on a tablet device or smartphone, you may want to skip forward to the chapter on R-Studio, because regular old R has not yet been reconfigured to work on tablet devices (but there is a workaround for this that uses R-studio). There are a few experiments with web-based interfaces to R, like this one - http://dssm.unipa.it/R-php/R-php-1/R/ - but they are still in a very early stage. If your computer has the Windows(r), Mac-OS-X(r) or a Linux operating system, there is a version of R waiting for you at http://cran.r-project.org/. Download and install your own copy. If you sometimes have difficulties with installing new software and you need some help, there is a wonderful little book by

Thomas P. Hogan called, *Bare Bones R: A Brief Introductory Guide* that you might want to buy or borrow from your library. There are lots of sites online that also give help with installing R, although many of them are not oriented towards the inexperienced user. I searched online using the term "help installing R" and I got a few good hits. One site that was quite informative for installing R on Windows was at "readthedocs.org," and you can try to access it at this TinyUrl: http://tinyurl.com/872ngtt. For Mac users there is a video by Jeremy Taylor at Vimeo.com,



http://vimeo.com/36697971, that outlines both the initial installation on a Mac and a number of other optional steps for getting started. YouTube also had four videos that provide brief tutorials for installing R. Try search for "install R" in the YouTube search box. The rest of this chapter assumes that you have installed R and can run it on your computer as shown in the screenshot on this page. (Note that this screenshot is from the Mac version of R: if you are running Windows or Linux your R screen may appear slightly different from this.) Just for fun, one of the first things you can do when you have R running is to click on the color wheel and customize the appearance of R. This screen shot uses Syracuse orange as a background color. The screenshot also shows a simple command to type that shows the most basic method of interaction with R. Notice near the bottom of the screenshot a greater than (">") symbol. This is the command prompt: When R is running and it is the active application on your desktop, if you type a command it appears after the ">" symbol. If you press the "enter" or "return" key, the command is sent to R for processing. When the processing is done, a result may appear just under the ">." When R is done processing, another command prompt (">") appears and R is ready for your next command. In the screen shot, the user has typed "1+1" and pressed the enter key. The formula 1+1 is used by elementary school students everywhere to insult each other's math skills, but R dutifully reports the result as 2. If you are a careful observer, you will notice that just before the 2 there is a "1" in brackets, like this: [1]. That [1] is a line number that helps to keep track of the results that R displays. Pretty pointless when only showing one line of results, but R likes to be consistent, so we will see quite a lot of those numbers in brackets as we dig deeper.

Remember the list of ages of family members from the last chapter? No? Well, here it is again: 43, 42, 12, 8, 5, for dad, mom, sis, bro, and the dog, respectively. The previous chapter mentioned that this was a list of items, all of the same mode, namely "integer."



Remember that you can tell that they are OK to be integers because there are no decimal points and therefore nothing after the decimal point. We can create a vector of integers in r using the "c()" command. Take a look at the next screenshot:

This is just about the last time that the whole screenshot from the R console will appear in the book. From here on out we will just look at commands and output so we don't waste so much space on the page. The first command line in the screen shot is exactly what appeared in the previous chapter:

```
c(43, 42, 12, 8, 5)
```

You may notice that on the following line, R dutifully reports the vector that you just typed. After the line number "[1]", we see the list 43, 42, 12, 8, and 5. R "echoes" this list back to us, because we didn't ask it to store the vector anywhere. In contrast, the next command line (also the same as in the previous chapter), says:

```
myFamilyAges <- c(43, 42, 12, 8, 5)
```

We have typed in the same list of numbers, but this time we have assigned it, using the left pointing arrow, into a storage area that we have named "myFamilyAges." This time, R responds just with an empty command prompt. That's why the third command line requests a report of what myFamilyAges contains (Look after the yellow ">". The text in blue is what you should type.) This is a simple but very important tool. Any time you want to know what is in a data object in R, just type the name of the object and R will report it back to you. In the next command we begin to see the power of R:

```
sum(myFamilyAges)
```

This command asks R to add together all of the numbers in myFamilyAges, which turns out to be 110 (you can check it yourself with a calculator if you want). This is perhaps a bit of a weird thing to do with the ages of family members, but it shows how

with a very short and simple command you can unleash quite a bit of processing on your data. In the next line we ask for the "mean" (what non-data people call the average) of all of the ages and this turns out to be 22 years. The command right afterwards, called "range," shows the lowest and highest ages in the list. Finally, just for fun, we tried to issue the command "fish(myFamilyAges)." Pretty much as you might expect, R does not contain a "fish()" function and so we received an error message to that effect. This shows another important principle for working with R: You can freely try things out at anytime without fear of breaking anything. If R can't understand what you want to accomplish, or you haven't quite figured out how to do something, R will calmly respond with an error message and will not make any other changes until you give it a new command. The error messages from R are not always super helpful, but with some strategies that the book will discuss in future chapters you can break down the problem and figure out how to get R to do what you want.

Let's take stock for a moment. First, you should definitely try all of the commands noted above on your own computer. You can read about the commands in this book all you want, but you will learn a lot more if you actually try things out. Second, if you try a command that is shown in these pages and it does not work for some reason, you should try to figure out why. Begin by checking your spelling and punctuation, because R is very persnickety about how commands are typed. Remember that capitalization matters in R: myFamilyAges is not the same as myfamilyages. If you verify that you have typed a command just as you see in the book and it still does not work, try to go online and look for some help. There's lots of help at http://stackoverflow.com, at https://stat.ethz.ch, and also at http://www.statmethods.net/. If you can figure out what went wrong on your own you will probably learn something very valuable about working with R. Third, you should take a moment to experiment a bit with each new set of commands that you learn. For example, just using the commands shown in the last screen shot you could do this totally new thing:

```
myRange <- range(myFamilyAges)
```

What would happen if you did that command, and then typed "myRange" (without the double quotes) on the next command line to report back what is stored there ? What would you see? Then think about how that worked and try to imagine some other experiments that you could try. The more you experiment on your own, the more you will learn. Some of the best stuff ever invented for computers was the result of just experimenting to see what was possible. At this point, with just the few commands that you have already tried, you already know the following things about R (and about data):

- How to install R on your computer and run it.
- How to type commands on the R console.
- The use of the "c()" function. Remember that "c" stands for concatenate, which just means to join things together. You can put a list of items inside the parentheses, separated by commas.
- That a vector is pretty much the most basic form of data storage in R, and that it consists of a list of items of the same mode.
- That a vector can be stored in a named location using the assignment arrow (a left pointing arrow made of a dash and a less than symbol, like this: "<-").

- That you can get a report of the data object that is in any named location just by typing that name at the command line.
- That you can "run" a function, such as mean(), on a vector of numbers to transform them into something else. (The mean() function calculates the average, which is one of the most basic numeric summaries there is.)
- That sum(), mean(), and range() are all legal functions in R whereas fish() is not.

In the next chapter we will move forward a step or two by starting to work with text and by combining our list of family ages with the names of the family members and some other information about them.

Chapter Challenge

Using logic and online resources to get help if you need it, learn how to use the c() function to add another family member's age on the end of the myFamilyAges vector.

Sources

http://a-little-book-of-r-for-biomedical-statistics.readthedocs.org/en/latest/src/installr.html

http://cran.r-project.org/

http://dssm.unipa.it/R-php/R-php-1/R/ (UNIPA experimental web interface to R)

http://en.wikibooks.org/wiki/R_Programming

https://plus.google.com/u/0/104922476697914343874/posts (Jeremy Taylor's blog: Stats Make Me Cry)

http://stackoverflow.com

https://stat.ethz.ch

http://www.statmethods.net/

Test Yourself

Review 3.1 Getting Started with R

Question 1 of 3

What is the cost of each software license for the R open source data analysis program?

- ✓ **A.** R is free
- **B.** 99 cents in the iTunes store
- **C.** $10
- **D.** $100

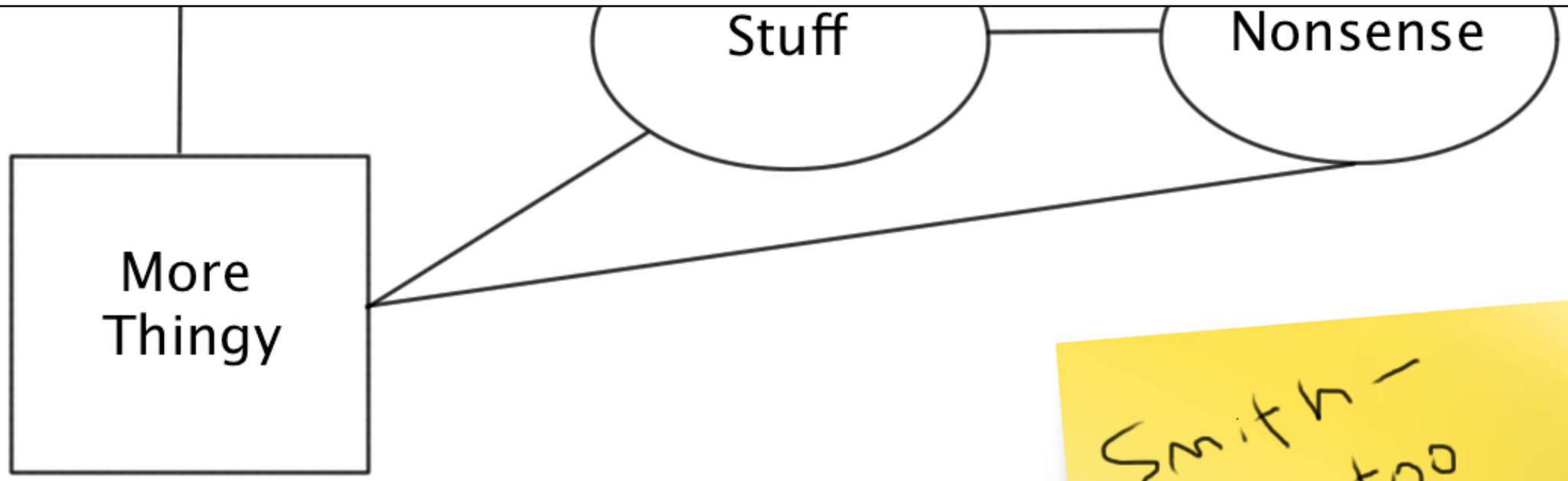
Check Answer

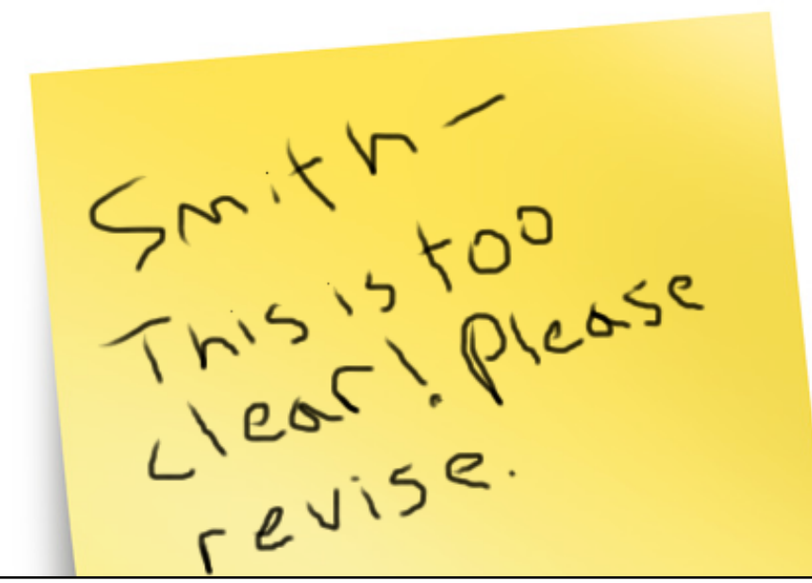
R Functions Used in This Chapter

c()	Concatenates data elements together
<-	Assignment arrow
sum()	Adds data elements
range()	Max value minus min value
mean()	The average

CHAPTER 4

Follow the Data





An old adage in detective work is to, "follow the money." In data science, one key to success is to "follow the data." In most cases, a data scientist will not help to design an information system from scratch. Instead, there will be several or many legacy systems where data resides; a big part of the challenge to the data scientist lies in integrating those systems.

Hate to nag, but have you had a checkup lately? If you have been to the doctor for any reason you may recall that the doctor's office is awash with data. First off, the doctor has loads of digital sensors, everything from blood pressure monitors to ultrasound machines, and all of these produce mountains of data. Perhaps of greater concern in this era of debate about health insurance, the doctors office is one of the big jumping off points for financial and insurance data. One of the notable "features" of the U.S. healthcare system is our most common method of healthcare delivery: paying by the procedure. When you experience a "procedure" at the doctor's office, whether it is a consultation, an examination, a test, or something else, this initiates a chain of data events with far reaching consequences.

If your doctor is typical, the starting point of these events is a paper form. Have you ever looked at one of these in detail? Most of the form will be covered by a large matrix of procedures and codes. Although some of the better equipped places may use this form digitally on a tablet or other computer, paper forms are still ubiquitous. Somewhere either in the doctor's office or at an outsourced service company, the data on the paper form are entered into a system that begins the insurance reimbursement and/or billing process.

Where do these procedure data go? What other kinds of data (such as patient account information) may get attached to them in a subsequent step? What kinds of networks do these linked data travel over, and what kind of security do they have? How many steps are there in processing the data before they get to he insurance company? How does the insurance company process and analyze the data before issuing the reimbursement? How is the money "transmitted" once the insurance company's systems have given approval to the reimbursement? These questions barely scratch the surface: there are dozens or hundreds of processing steps that we haven't yet imagined.

It is easy to see from this example, that the likelihood of being able to throw it all out and start designing a better or at least more standardized system from scratch is nil. But what if you had the job of improving the efficiency of the system, or auditing the insurance reimbursements to make sure they were compliant with insurance records, or using the data to detect and predict outbreaks and epidemics, or providing feedback to consumers about how much they can expect to pay out of pocket for various procedures?

The critical starting point for your project would be to follow the data. You would need to be like a detective, finding out in a substantial degree of detail the content, format, senders, receivers, transmission methods, repositories, and users of data at each step in the process and at each organization where the data are processed or housed.

Fortunately there is an extensive area of study and practice called "data modeling" that provides theories, strategies, and tools to help with the data scientist's goal of following the data. These ideas started in earnest in the 1970s with the introduction by computer scientist Ed Yourdon of a methodology called Data Flow Diagrams. A more contemporary approach, that is strongly linked with the practice of creating relational databases, is called the entity-relationship model. Professionals using this model develop Entity-Relationship Diagrams (ERDs) that describe the structure and movement of data in a system.

Entity-relationship modeling occurs at different levels ranging from an abstract conceptual level to a physical storage level. At the conceptual level an entity is an object or thing, usually something in the real world. In the doctor's office example, one important "object" is the patient. Another entity is the doctor. The patient and the doctor are linked by a relationship: in modern health care lingo this is the "provider" relationship. If the patient is Mr. X and the doctor is Dr. Y, the provider relationship provides a bidirectional link:

- Dr. Y is the provider for Mr. X
- Mr. X's provider is Dr. Y

Naturally there is a range of data that can represent Mr. X: name address, age, etc. Likewise, there are data that represent Dr. Y: years of experience as a doctor, specialty areas, certifications, licenses. Importantly, there is also a chunk of data that represents the linkage between X and Y, and this is the relationship.

Creating an ERD requires investigating and enumerating all of the entities, such as patients and doctors, as well as all of the relationships that may exist among them. As the beginning of the chapter suggested, this may have to occur across multiple organizations (e.g., the doctor's office and the insurance company) depending upon the purpose of the information system that is being designed. Eventually, the ERDs must become detailed enough that they can serve as a specification for the physical storage in a database.

In an application area like health care, there are so many choices for different ways of designing the data that it requires some experience and possibly some "art" to create a workable system. Part of the art lies in understanding the users' current information needs and anticipating how those needs may change in the future. If an organization is redesigning a system, adding to a system, or creating brand new systems, they are doing so in the expectation of a future benefit. This benefit may arise from greater efficiency, reduction of errors/inaccuracies, or the hope of providing a new product or service with the enhanced information capabilities.

Whatever the goal, the data scientist has an important and difficult challenge of taking the methods of today - including paper forms and manual data entry - and imagining the methods of tomorrow. Follow the data!

Sources

http://en.wikipedia.org/wiki/Data_modeling

http://en.wikipedia.org/wiki/Entity-relationship_diagram

CHAPTER 5

Rows and Columns

One of the most basic and widely used methods of representing data is to use rows and columns, where each row is a case or instance and each column is a variable and attribute. Most spreadsheets arrange their data in rows and columns, although spreadsheets don't usually refer to these as cases or variables. R represents rows and columns in an object called a data frame.

Although we live in a three dimensional world, where a box of cereal has height, width, and depth, it is a sad fact of modern life that pieces of paper, chalkboards, whiteboards, and computer screens are still only two dimensional. As a result, most of the statisticians, accountants, computer scientists, and engineers who work with lots of numbers tend to organize them in rows and columns. There's really no good reason for this other than it makes it easy to fill a rectangular piece of paper with numbers. Rows and columns can be organized any way that you want, but the most common way is to have the rows be "cases" or "instances" and the columns be "attributes" or "variables." Take a look at this nice, two dimensional representation of rows and columns:

NAME	AGE	GENDER	WEIGHT
Dad	43	Male	188
Mom	42	Female	136
Sis	12	Female	83
Bro	8	Male	61
Dog	5	Female	44

Pretty obvious what's going on, right? The top line, in bold, is not really part of the data. Instead, the top line contains the attribute or variable names. Note that computer scientists tend to call them attributes while statisticians call them variables. Either term is OK. For example, age is an attribute that every living thing has, and you could count it in minutes, hours, days, months, years, or other units of time. Here we have the Age attribute calibrated in years. Technically speaking, the variable names in the top line are "metadata" or what you could think of as data about data. Imagine how much more difficult it would be to understand what was going on in that table without the metadata. There's lot of different kinds of metadata: variable names are just one simple type of metadata.

So if you ignore the top row, which contains the variable names, each of the remaining rows is an instance or a case. Again, computer scientists may call them instances, and statisticians may call them cases, but either term is fine. The important thing is that each row refers to an actual thing. In this case all of our things are living creatures in a family. You could think of the Name column as "case labels" in that each one of these labels refers to one and only one row in our data. Most of the time when you are working with a large dataset, there is a number used for the case label, and that number is unique for each case (in other words, the same number would never appear in more than one row). Computer scientists sometimes refer to this column of unique numbers as a "key." A key is very useful particularly for matching things up from different data sources, and we will run into this idea again a bit later. For now, though, just take note that the "Dad" row can be distinguished from the "Bro" row, even though they are both Male. Even if we added an "Uncle" row that had the same Age, Gender, and Weight as "Dad" we would still be able to tell the two rows apart because one would have the name "Dad" and the other would have the name "Uncle."

One other important note: Look how each column contains the same kind of data all the way down. For example, the Age column is all numbers. There's nothing in the Age column like "Old" or "Young." This is a really valuable way of keeping things organized. After all, we could not run the mean() function on the Age col-

umn if it contained little piece of text, like "Old" or "Young." On a related note, every cell (that is an intersection of a row and a column, for example, Sis's Age) contains just one piece of information. Although a spreadsheet or a word processing program might allow us to put more than one thing in a cell, a real data handling program will not. Finally, see that every column has the same number of entries, so that the whole forms a nice rectangle. When statisticians and other people who work with databases work with a dataset, they expect this rectangular arrangement.

Now let's figure out how to get these rows and columns into R. One thing you will quickly learn about R is that there is almost always more than one way to accomplish a goal. Sometimes the quickest or most efficient way is not the easiest to understand. In this case we will build each column one by one and then join them together into a single data frame. This is a bit labor intensive, and not the usual way that we would work with a data set, but it is easy to understand. First, run this command to make the column of names:

```
myFamilyNames <- c("Dad","Mom","Sis","Bro","Dog")
```

One thing you might notice is that every name is placed within double quotes. This is how you signal to R that you want it to treat something as a string of characters rather than the name of a storage location. If we had asked R to use Dad instead of "Dad" it would have looked for a storage location (a data object) named Dad. Another thing to notice is that the commas separating the different values are outside of the double quotes. If you were writing a regular sentence this is not how things would look, but for computer programming the comma can only do its job of separating the different values if it is not included inside the quotes. Once you have typed the line above, remember that you can check the contents of myFamilyNames by typing it on the next command line:

```
myFamilyNames
```

The output should look like this:

```
[1] "Dad" "Mom" "Sis" "Bro" "Dog"
```

Next, you can create a vector of the ages of the family members, like this:

```
myFamilyAges <- c(43, 42, 12, 8, 5)
```

Note that this is exactly the same command we used in the last chapter, so if you have kept R running between then and now you would not even have to retype this command because myFamilyAges would still be there. Actually, if you closed R since working the examples from the last chapter you will have been prompted to "save the workspace" and if you did so, then R restored all of the data objects you were using in the last session. You can always check by typing myFamilyAges on a blank command line. The output should look like this:

```
[1] 43 42 12  8  5
```

Hey, now you have used the c() function and the assignment arrow to make myFamilyNames and myFamilyAges. If you look at the data table earlier in the chapter you should be able to figure out the commands for creating myFamilyGenders and myFamilyWeights. In case you run into trouble, these commands also appear on the next page, but you should try to figure them out for yourself before you turn the page. In each case after you type the command to create the new data object, you should also type the name of the data

object at the command line to make sure that it looks the way it should. Four variables, each one with five values in it. Two of the variables are character data and two of the variables are integer data. Here are those two extra commands in case you need them:

```
myFamilyGenders <- c("Male","Female","Female","Male","Female")

myFamilyWeights <- c(188,136,83,61,44)
```

Now we are ready to tackle the dataframe. In R, a dataframe is a list (of columns), where each element in the list is a vector. Each vector is the same length, which is how we get our nice rectangular row and column setup, and generally each vector also has its own name. The command to make a data frame is very simple:

```
myFamily <- data.frame(myFamilyNames, +
myFamilyAges, myFamilyGenders, myFamilyWeights)
```

Look out! We're starting to get commands that are long enough that they break onto more than one line. The + at the end of the first line tells R to wait for more input on the next line before trying to process the command. If you want to, you can type the whole thing as one line in R, but if you do, just leave out the plus sign. Anyway, the data.frame() function makes a dataframe from the four vectors that we previously typed in. Notice that we have also used the assignment arrow to make a new stored location where R puts the data frame. This new data object, called myFamily, is our dataframe. Once you have gotten that command to work, type myFamily at the command line to get a report back of what the data frame contains. Here's the output you should see:

```
  myFamilyNames myFamilyAges myFamilyGenders myFamilyWeights
1           Dad           43            Male             188
2           Mom           42          Female             136
3           Sis           12          Female              83
4           Bro            8            Male              61
5           Dog            5          Female              44
```

This looks great. Notice that R has put row numbers in front of each row of our data. These are different from the output line numbers we saw in brackets before, because these are actual "indices" into the data frame. In other words, they are the row numbers that R uses to keep track of which row a particular piece of data is in.

With a small data set like this one, only five rows, it is pretty easy just to take a look at all of the data. But when we get to a bigger data set this won't be practical. We need to have other ways of summarizing what we have. The first method reveals the type of "structure" that R has used to store a data object.

```
> str(myFamily)

'data.frame':   5 obs. of  4 variables:

 $ myFamilyNames  : Factor w/ 5 levels

        "Bro","Dad","Dog",..: 2 4 5 1 3

 $ myFamilyAges   : num  43 42 12 8 5

 $ myFamilyGenders: Factor w/ 2 levels

        "Female","Male": 2 1 1 2 1

 $ myFamilyWeights: num  188 136 83 61 44
```

Take note that for the first time, the example shows the command prompt ">" in order to differentiate the command from the output that follows. You don't need to type this: R provides it whenever it is ready to receive new input. From now on in the book, there will

be examples of R commands and output that are mixed together, so always be on the lookout for ">" because the command after that is what you have to type.

OK, so the function "str()" reveals the structure of the data object that you name between the parentheses. In this case we pretty well knew that myFamily was a data frame because we just set that up in a previous command. In the future, however, we will run into many situations where we are not sure how R has created a data object, so it is important to know str() so that you can ask R to report what an object is at any time.

In the first line of output we have the confirmation that myFamily is a data frame as well as an indication that there are five observations ("obs." which is another word that statisticians use instead of cases or instances) and four variables. After that first line of output, we have four sections that each begin with "$". For each of the four variables, these sections describe the component columns of the myFamily dataframe object.

Each of the four variables has a "mode" or type that is reported by R right after the colon on the line that names the variable:

```
$ myFamilyGenders: Factor w/ 2 levels
```

For example, myFamilyGenders is shown as a "Factor." In the terminology that R uses, Factor refers to a special type of label that can be used to identify and organize groups of cases. R has organized these labels alphabetically and then listed out the first few cases (because our dataframe is so small it actually is showing us all of the cases). For myFamilyGenders we see that there are two "levels," meaning that there are two different options: female and male. R assigns a number, starting with one, to each of these levels, so every case that is "Female" gets assigned a 1 and every case that is "Male" gets assigned a 2 (because Female comes before Male in the alphabet, so Female is the first Factor label, so it gets a 1). If you have your thinking cap on, you may be wondering why we started out by typing in small strings of text, like "Male," but then R has gone ahead and converted these small pieces of text into numbers that it calls "Factors." The reason for this lies in the statistical origins of R. For years, researchers have done things like calling an experimental group "Exp" and a control, group "Ctl" without intending to use these small strings of text for anything other than labels. So R assumes, unless you tell it otherwise, that when you type in a short string like "Male" that you are referring to the label of a group, and that R should prepare for the use of Male as a "Level" of a "Factor." When you don't want this to happen you can instruct R to stop doing this with an option on the data.frame() function: stringsAsFactors=FALSE. We will look with more detail at options and defaults a little later on.

Phew, that was complicated! By contrast, our two numeric variables, myFamilyAges and myFamilyWeights, are very simple. You can see that after the colon the mode is shown as "num" (which stands for numeric) and that the first few values are reported:

```
$ myFamilyAges   : num  43 42 12 8 5
```

Putting it altogether, we have pretty complete information about the myFamily dataframe and we are just about ready to do some more work with it. We have seen firsthand that R has some pretty cryptic labels for things as well as some obscure strategies for converting this to that. R was designed for experts, rather than nov-

ices, so we will just have to take our lumps so that one day we can be experts too.

Next, we will examine another very useful function called summary(). Summary() provides some overlapping information to str() but also goes a little bit further, particularly with numeric variables. Here's what we get:

```
> summary(myFamily)

 myFamilyNames  myFamilyAges

 Bro: 1         Min.    : 5

 Dad: 1         1st Qu. : 8

 Dog: 1         Median  : 12

 Mom: 1         Mean    : 22

 Sis: 1         3rd Qu. : 42


myFamilyGenders myFamilyWeights

 Female : 3     Min.    : 44

 Male   : 2     1st Qu. : 61.0

                Median  : 83.0

                Mean    : 102.4

                3rd Qu. : 136.0

                Max     : 188.0
```

In order to fit on the page properly, these columns have been reorganized a bit. The name of a column/variable, sits up above the information that pertains to it, and each block of information is independent of the others (so it is meaningless, for instance, that "Bro: 1" and "Min." happen to be on the same line of output). Notice, as with str(), that the output is quite different depending upon whether we are talking about a Factor, like myFamilyNames or myFamilyGenders, versus a numeric variable like myFamilyAges and myFamilyWeights. The columns for the Factors list out a few of the Factor names along with the number of occurrences of cases that are coded with that factor. So for instance, under myFamilyGenders it shows three females and two males. In contrast, for the numeric variables we get five different calculated quantities that help to summarize the variable. There's no time like the present to start to learn about what these are, so here goes:

- "Min." refers to the minimum or lowest value among all the cases. For this dataframe, 5 is the age of the dog and it is the lowest age of all of the family members.
- "1st Qu." refers to the dividing line at the top of the first quartile. If we took all the cases and lined them up side by side in order of age (or weight) we could then divide up the whole into four groups, where each group had the same number of observations.

1ST QUARTILE	2ND QUARTILE	3RD QUARTILE	4TH QUARTILE
25% of cases with the smallest values here	25% of cases with low medium values here	25% of cases with high medium values here	25% of cases with the largest values here

Just like a number line, the smallest cases would be on the left with the largest on the right. If we're looking at myFamilyAges, the leftmost group, which contains one quarter of all the cases, would start with five on the low end (the dog) and would have eight on the high end (Bro). So the "first quartile" is the value of age (or another variable) that divides the first quarter of the cases from the other three quarters. Note that if we don't have a number of cases that divides evenly by four, the value is an approximation.

- Median refers to the value of the case that splits the whole group in half, with half of the cases having higher values and half having lower values. If you think about it a little bit, the median is also the dividing line that separates the second quartile from the third quartile.
- Mean, as we have learned before, is the numeric average of all of the values. For instance, the average age in the family is reported as 22.
- "3rd Qu." is the third quartile. If you remember back to the first quartile and the median, this is the third and final dividing line that splits up all of the cases into four equal sized parts. You may be wondering about these quartiles and what they are useful for. Statisticians like them because they give a quick sense of the shape of the distribution. Everyone has the experience of sorting and dividing things up - pieces of pizza, playing cards into hands, a bunch of players into teams - and it is easy for most people to visualize four equal sized groups and useful to know how high you need to go in age or weight (or another variable) to get to the next dividing line between the groups.
- Finally, "Max" is the maximum value and as you might expect displays the highest value among all of the available cases. For example, in this dataframe Dad has the highest weight: 188. Seems like a pretty trim guy.

Just one more topic to pack in before ending this chapter: How to access the stored variables in our new dataframe. R stores the dataframe as a list of vectors and we can use the name of the dataframe together with the name of a vector to refer to each one using the "$" to connect the two labels like this:

```
> myFamily$myFamilyAges

[1] 43 42 12  8  5
```

If you're alert you might wonder why we went to the trouble of typing out that big long thing with the $ in the middle, when we could have just referred to "myFamilyAges" as we did earlier when we were setting up the data. Well, this is a very important point. When we created the myFamily dataframe, we **copied** all of the information from the individual vectors that we had before into a brand new storage space. So now that we have created the myFamily dataframe, myFamilyData$myFamilyAges actually refers to a completely separate (but so far identical) vector of values. You can prove this to yourself very easily, and you should, by adding some data to the original vector, myFamilyAges:

```
> myFamilyAges <- c(myFamilyAges, 11)

> myFamilyAges

[1] 43 42 12  8  5 11

> myFamily$myFamilyAges
```

```
[1] 43 42 12  8  5
```

Look very closely at the five lines above. In the first line, we use the c() command to add the value 11 to the original list of ages that we had stored in myFamilyAges (perhaps we have adopted an older cat into the family). In the second line we ask R to report what the vector myFamilyAges now contains. Dutifully, on the third line above, R reports that myFamilyAges now contains the original five values and the new value of 11 on the end of the list. When we ask R to report myFamily$myFamilyAges, however, we still have the original list of five values only. This shows that the dataframe and its component columns/vectors is now a completely independent piece of data. We must be very careful, if we established a dataframe that we want to use for subsequent analysis, that we don't make a mistake and keep using some of the original data from which we assembled the dataframe.

Here's a puzzle that follows on from this question. We have a nice dataframe with five observations and four variables. This is a rectangular shaped data set, as we discussed at the beginning of the chapter. What if we tried to add on a new piece of data on the end of one of the variables? In other words, what if we tried something like this command:

```
myFamily$myFamilyAges<-c(myFamily$myFamilyAges, 11)
```

If this worked, we would have a pretty weird situation: The variable in the dataframe that contained the family members' ages would all of a sudden have one more observation than the other variables: no more perfect rectangle! Try it out and see what happens. The result helps to illuminate how R approaches situations like this.

So what new skills and knowledge do we have at this point? Here are a few of the key points from this chapter:

- In R, as in other programs, a vector is a list of elements/things that are all of the same kind, or what R refers to as a mode. For example, a vector of mode "numeric" would contain only numbers.
- Statisticians, database experts and others like to work with rectangular datasets where the rows are cases or instances and the columns are variables or attributes.
- In R, one of the typical ways of storing these rectangular structures is in an object known as a dataframe. Technically speaking a dataframe is a list of vectors where each vector has the exact same number of elements as the others (making a nice rectangle).
- In R, the data.frame() function organizes a set of vectors into a dataframe. A dataframe is a conventional, rectangular shaped data object where each column is a vector of uniform mode and having the same number of elements as the other columns in the dataframe. Data are copied from the original source vectors into new storage space. The variables/columns of the dataframe can be accessed using "$" to connect the name of the dataframe to the name of the variable/column.
- The str() and summary() functions can be used to reveal the structure and contents of a dataframe (as well as of other data objects stored by R). The str() function shows the structure of a data object, while summary() provides an numerical summaries of numeric variables and overviews of non-numeric variables.

- A factor is a labeling system often used to organize groups of cases or observations. In R, as well as in many other software programs, a factor is represented internally with a numeric ID number, but factors also typically have labels like "Male" and "Female" or "Experiment" and "Control." Factors always have "levels," and these are the different groups that the factor signifies. For example, if a factor variable called Gender codes all cases as either "Male" or "Female" then that factor has exactly two levels.

Review 5.1 Rows and columns

Question 1 of 7

What is the name of the data object that R uses to store a rectangular dataset of cases and variables?

- **A.** A list
- **B.** A mode
- **C.** A vector
- ✓ **D.** A dataframe

Check Answer

- Quartiles are a division of a sorted vector into four evenly sized groups. The first quartile contains the lowest-valued elements, for example the lightest weights, whereas the fourth quartile contains the highest-valued items. Because there are four groups, there are three dividing lines that separate them. The middle dividing line that splits the vector exactly in half is the median. The term "first quartile" often refers to the dividing line to the left of the median that splits up the lower two quarters and the value of the first quartile is the value of the element of the vector that sits right at that dividing line. Third quartile is the same idea, but to the right of the median and splitting up the two higher quarters.
- Min and max are often used as abbreviations for minimum and maximum and these are the terms used for the highest and lowest values in a vector. Bonus: The "range" of a set of numbers is the maximum minus the minimum.
- The mean is the same thing that most people think of as the average. Bonus: The mean and the median are both measures of what statisticians call "central tendency."

Chapter Challenge

Create another variable containing information about family members (for example, each family member's estimated IQ; you can make up the data). Take that new variable and put it in the existing

myFamily dataframe. Rerun the summary() function on myFamily to get descriptive information on your new variable.

Sources

http://en.wikipedia.org/wiki/Central_tendency

http://en.wikipedia.org/wiki/Median

http://en.wikipedia.org/wiki/Relational_model

http://msenux.redwoods.edu/math/R/dataframe.php

http://stat.ethz.ch/R-manual/R-devel/library/base/html/data.frame.html

http://www.burns-stat.com/pages/Tutor/hints_R_begin.html

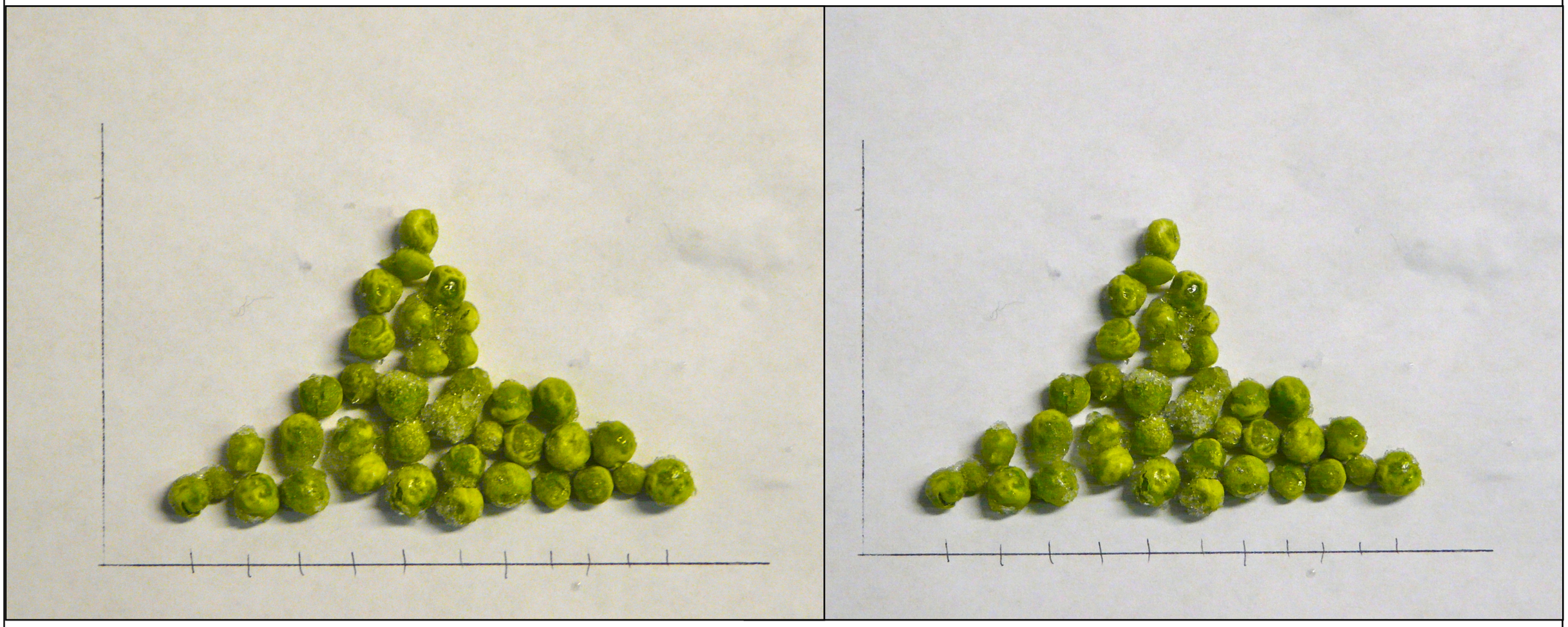
http://www.khanacademy.org/math/statistics/v/mean-median-and-mode

R Functions Used in This Chapter

c()	Concatenates data elements together
<-	Assignment arrow
data.frame()	Makes a dataframe from separate vectors
str()	Reports the structure of a data object
summary()	Reports data modes/types and a data overview

CHAPTER 6

Beer, Farms, and Peas



Many of the simplest and most practical methods for summarizing collections of numbers come to us from four guys who were born in the 1800s at the start of the industrial revolution. A considerable amount of the work they did was focused on solving real world problems in manufacturing and agriculture by using data to describe and draw inferences from what they observed.

The end of the 1800s and the early 1900s were a time of astonishing progress in mathematics and science. Given enough time, paper, and pencils, scientists and mathematicians of that age imagined that just about any problem facing humankind - including the limitations of people themselves - could be measured, broken down, analyzed, and rebuilt to become more efficient. Four Englishmen who epitomized both this scientific progress and these idealistic beliefs were Francis Galton, Karl Pearson, William Sealy Gosset, and Ronald Fisher.

First on the scene was Francis Galton, a half-cousin to the more widely known Charles Darwin, but quite the intellectual force himself. Galton was an English gentleman of independent means who studied Latin, Greek, medicine, and mathematics, and who made a name for himself as an African explorer. He is most widely known as a proponent of "eugenics" and is credited with coining the term. Eugenics is the idea that the human race could be improved through selective breeding. Galton studied heredity in peas, rabbits, and people and concluded that certain people should be paid to get married and have children because their offspring would improve the human race. These ideas were later horribly misused in the 20th century, most notably by the Nazis as a justification for killing people because they belonged to supposedly inferior races. Setting eugenics aside, however, Galton made several notable and valuable contributions to mathematics and statistics, in particular illuminating two basic techniques that are widely used today: correlation and regression.

For all his studying and theorizing, Galton was not an outstanding mathematician, but he had a junior partner, Karl Pearson, who is often credited with founding the field of mathematical statistics. Pearson refined the math behind correlation and regression and did a lot else besides to contribute to our modern abilities to manage numbers. Like Galton, Pearson was a proponent of eugenics, but he also is credited with inspiring some of Einstein's thoughts about relativity and was an early advocate of women's rights.

Next to the statistical party was William Sealy Gosset, a wizard at both math and chemistry. It was probably the latter expertise that led the Guinness Brewery in Dublin Ireland to hire Gosset after college. As a forward looking business, the Guinness brewery was on the lookout for ways of making batches of beer more consistent in quality. Gosset stepped in and developed what we now refer to as small sample statistical techniques - ways of generalizing from the results of a relatively few observations. Of course, brewing a batch of beer is a time consuming and expensive process, so in order to draw conclusions from experimental methods applied to just a few batches, Gosset had to figure out the role of chance in determining how a batch of beer had turned out. Guinness frowned upon academic publications, so Gosset had to publish his results under the modest pseudonym, "Student." If you ever hear someone discussing the "Student's t-Test," that is where the name came from.

Last but not least among the born-in-the-1800s bunch was Ronald Fisher, another mathematician who also studied the natural sciences, in his case biology and genetics. Unlike Galton, Fisher was not a gentleman of independent means, in fact, during his early married life he and his wife struggled as subsistence farmers. One of Fisher's professional postings was to an agricultural research farm called Rothhamsted Experimental Station. Here, he had access to data about variations in crop yield that led to his development of an essential statistical technique known as the analysis of

variance. Fisher also pioneered the area of experimental design, which includes matters of factors, levels, experimental groups, and control groups that we noted in the previous chapter.

Of course, these four are certainly not the only 19th and 20th century mathematicians to have made substantial contributions to practical statistics, but they are notable with respect to the applications of mathematics and statistics to the other sciences (and "Beer, Farms, and Peas" makes a good chapter title as well).

One of the critical distinctions woven throughout the work of these four is between the "sample" of data that you have available to analyze and the larger "population" of possible cases that may or do exist. When Gosset ran batches of beer at the brewery, he knew that it was impractical to run every possible batch of beer with every possible variation in recipe and preparation. Gosset knew that he had to run a few batches, describe what he had found and then generalize or infer what might happen in future batches. This is a fundamental aspect of working with all types and amounts of data: Whatever data you have, there's always more out there. There's data that you might have collected by changing the way things are done or the way things are measured. There's future data that hasn't been collected yet and might never be collected. There's even data that we might have gotten using the exact same strategies we did use, but that would have come out subtly different just due to randomness. Whatever data you have, it is just a snapshot or "sample" of what might be out there. This leads us to the conclusion that we can never, ever 100% trust the data we have. We must always hold back and keep in mind that there is always uncertainty in data. A lot of the power and goodness in statistics comes from the capabilities that people like Fisher developed to help us characterize and quantify that uncertainty and for us to know when to guard against putting too much stock in what a sample of data have to say. So remember that while we can always **describe** the sample of data we have, the real trick is to **infer** what the data may mean when generalized to the larger population of data that we don't have. This is the key distinction between descriptive and inferential statistics.

We have already encountered several descriptive statistics in previous chapters, but for the sake of practice here they are again, this time with the more detailed definitions:

- The mean (technically the arithmetic mean), a measure of central tendency that is calculated by adding together all of the observations and dividing by the number of observations.
- The median, another measure of central tendency, but one that cannot be directly calculated. Instead, you make a sorted list of all of the observations in the sample, then go halfway up that list. Whatever the value of the observation is at the halfway point, that is the median.
- The range, which is a measure of "dispersion" - how spread out a bunch of numbers in a sample are - calculated by subtracting the lowest value from the highest value.

To this list we should add three more that you will run into in a variety of situations:

- The mode, another measure of central tendency. The mode is the value that occurs most often in a sample of data. Like the median, the mode cannot be directly calculated. You just have to

count up how many of each number there are and then pick the category that has the most.

- The variance, a measure of dispersion. Like the range, the variance describes how spread out a sample of numbers is. Unlike the range, though, which just uses two numbers to calculate dispersion, the variance is obtained from all of the numbers through a simple calculation that compares each number to the mean. If you remember the ages of the family members from the previous chapter and the mean age of 22, you will be able to make sense out of the following table:

WHO	AGE	AGE - MEAN	(AGE-MEAN)^2
Dad	43	43-22 = 21	21*21=441
Mom	42	42-22=20	20*20=400
Sis	12	12-22=-10	-10*-10=100
Bro	8	8-22=-14	-14*-14=196
Dog	5	5-22=-17	-17*-17=289
		Total:	**1426**
		Total/4:	**356.5**

This table shows the calculation of the variance, which begins by obtaining the "deviations" from the mean and then "squares" them (multiply each times itself) to take care of the negative deviations (for example, -14 from the mean for Bro). We add up all of the squared deviations and then divide by the number of observations to get a kind of "average squared deviation." Note that it was not a mistake to divide by 4 instead of 5 - the reasons for this will become clear later in the book when we examine the concept of degrees of freedom. This result is the variance, a very useful mathematical concept that appears all over the place in statistics. While it is mathematically useful, it is not too nice too look at. For instance, in this example we are looking at the 356.5 squared-years of deviation from the mean. Who measures anything in squared years? Squared feet maybe, but that's a different discussion. So, to address this weirdness, statisticians have also provided us with:

- The standard deviation, another measure of dispersion, and a cousin to the variance. The standard deviation is simply the square root of the variance, which puts us back in regular units like "years." In the example above, the standard deviation would be about 18.88 years (rounding to two decimal places, which is plenty in this case).

Now let's have R calculate some statistics for us:

```
> var(myFamily$myFamilyAges)

[1] 356.5

> sd(myFamily$myFamilyAges)

[1] 18.88121
```

Note that these commands carry on using the data used in the previous chapter, including the use of the $ to address variables within a dataframe. If you do not have the data from the previous chapter you can also do this:

```
> var(c(43,42,12,8,5))

[1] 356.5
```

```
> sd(c(43,42,12,8,5))

[1] 18.88121
```

This is a pretty boring example, though, and not very useful for the rest of the chapter, so here's the next step up in looking at data. We will use the Windows or Mac clipboard to cut and paste a larger data set into R. Go to the U.S. Census website where they have stored population data:

http://www.census.gov/popest/data/national/totals/2011/index.html

Assuming you have a spreadsheet program available, click on the XLS link for "Annual Estimates of the Resident Population for the United States." When the spreadsheet is open, select the population estimates for the fifty states. The first few looked like this in the 2011 data:

.Alabama	4,779,736
.Alaska	710,231
.Arizona	6,392,017
.Arkansas	2,915,918

To make use of the next R command, make sure to choose just the numbers and not the text. Before you copy the numbers, take out the commas by switching the cell type to "General." This can usually be accomplished under the Format menu, but you might also have a toolbar button to do the job. Copy the numbers to the clipboard with ctrl+C (Windows) or command+C (Mac). On a Windows machine use the following command:

```
read.DIF("clipboard",transpose=TRUE)
```

On a Mac, this command does the same thing:

```
read.table(pipe("pbpaste"))
```

It is very annoying that there are two different commands for the two types of computers, but this is an inevitable side effect of the different ways that the designers at Microsoft and Apple set up the clipboard, plus the fact that R was designed to work across many platforms. Anyway, you should have found that the long string of population numbers appeared on the R output. The numbers are not much use to us just streamed to the output, so let's assign the numbers to a new vector.

Windows, using read.DIF:

```
> USstatePops <- +
read.DIF("clipboard",transpose=TRUE)

> USstatePops

          V1

1   4779736

2    710231

3   6392017

...
```

Or Mac, using read.table:

```
> USstatePops <- read.table(pipe("pbpaste"))

> USstatePops
```

```
          V1
1   4779736
2    710231
3   6392017
...
```

Only the first three observations are shown in order to save space on the page. Your output R should show the whole list. Note that the only thing new here over and above what we have done with R in previous chapters is the use of the read.DIF() or read.table() functions to get a bigger set of data that we don't have to type ourselves. Functions like read.table() are quite important as we move forward with using R because they provide the usual way of getting data stored in external files into R's storage areas for use in data analysis. If you had trouble getting this to work, you can cut and paste the commands at the end of the chapter under "If All Else Fails" to get the same data going in your copy of R.

Note that we have used the left pointing assignment arrow ("<-") to take the results of the read.DIF() or read.table() function and place it in a data object. This would be a great moment to practice your skills from the previous chapter by using the str() and summary() functions on our new data object called USstatePops. Did you notice anything interesting from the results of these functions? One thing you might have noticed is that there are 51 observations instead of 50. Can you guess why? If not, go back and look at your original data from the spreadsheet or the U.S. Census site. The other thing you may have noticed is that USstatePops is a dataframe, and not a plain vector of numbers. You can actually see this in the output above: In the second command line where we request that R reveal what is stored in USstatePops, it responds with a column topped by the designation "V1". Because we did not give R any information about the numbers it read in from the clipboard, it called them "V1", short for Variable One, by default. So anytime we want to refer to our list of population numbers we actually have to use the name USstatePops$V1. If this sounds unfamiliar, take another look at the previous "Rows and Columns" chapter for more information on addressing the columns in a dataframe.

Now we're ready to have some fun with a good sized list of numbers. Here are the basic descriptive statistics on the population of the states:

```
> mean(USstatePops$V1)
[1] 6053834
> median(USstatePops$V1)
[1] 4339367
> mode(USstatePops$V1)
[1] "numeric"
> var(USstatePops$V1)
[1] 4.656676e+13
> sd(USstatePops$V1)
[1] 6823984
```

Some great summary information there, but wait, a couple things have gone awry:

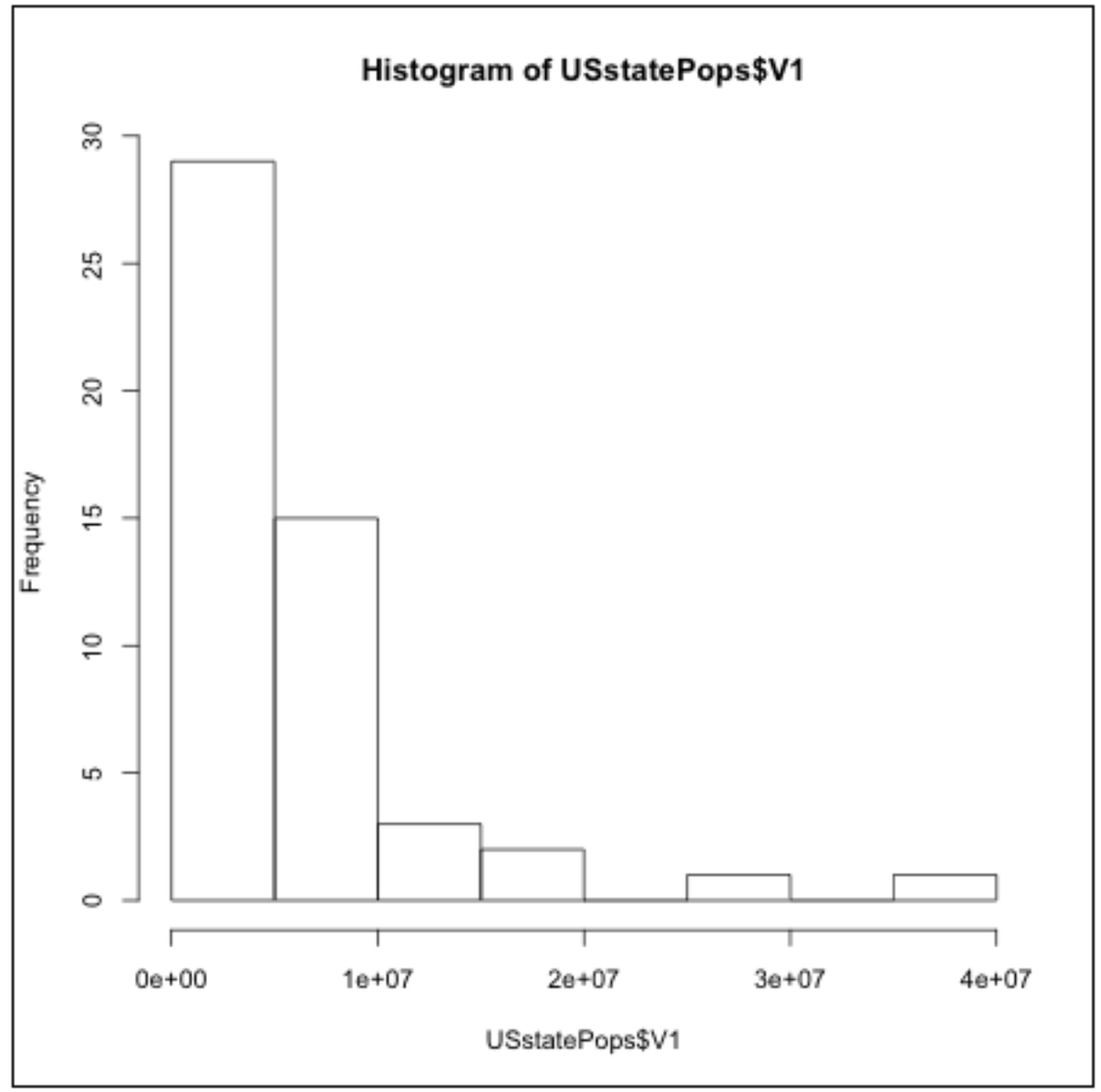
- The mode() function has returned the data type of our vector of numbers instead of the statistical mode. This is weird but true: the basic R package does not have a statistical mode function! This is partly due to the fact that the mode is only useful in a very limited set of situations, but we will find out in later packages how add-on packages can be used to get new functions in R including one that calculates the statistical mode.

- The variance is reported as 4.656676e+13. This is the first time that we have seen the use of scientific notation in R. If you haven't seen this notation before, the way you interpret it is to imagine 4.656676 multiplied by 10,000,000,000,000 (also known as 10 raised to the 13th power). You can see that this is ten trillion, a huge and unwieldy number, and that is why scientific notation is used. If you would prefer not to type all of that into a calculator, another trick to see what number you are dealing with is just to move the decimal point 13 digits to the right.

Other than these two issues, we now know that the average population of a U.S. state is 6,053,834 with a standard deviation of 6,823,984. You may be wondering, though, what does it mean to have a standard deviation of almost seven million? The mean and standard deviation are OK, and they certainly are mighty precise, but for most of us, it would make much more sense to have a *picture* that shows the central tendency and the dispersion of a large set of numbers. So here we go. Run this command:

```
hist(USstatePops$V1)
```

Here's the output you should get:



A histogram is a specialized type of bar graph designed to show "frequencies." Frequencies means how often a particular value or range of values occurs in a dataset. This histogram shows a very interesting picture. There are nearly 30 states with populations under five million, another 10 states with populations under 10 million, and then a very small number of states with populations greater than 10 million. Having said all that, how do we glean this kind of information from the graph? First, look along the Y-axis (the vertical axis on the left) for an indication of how often the data

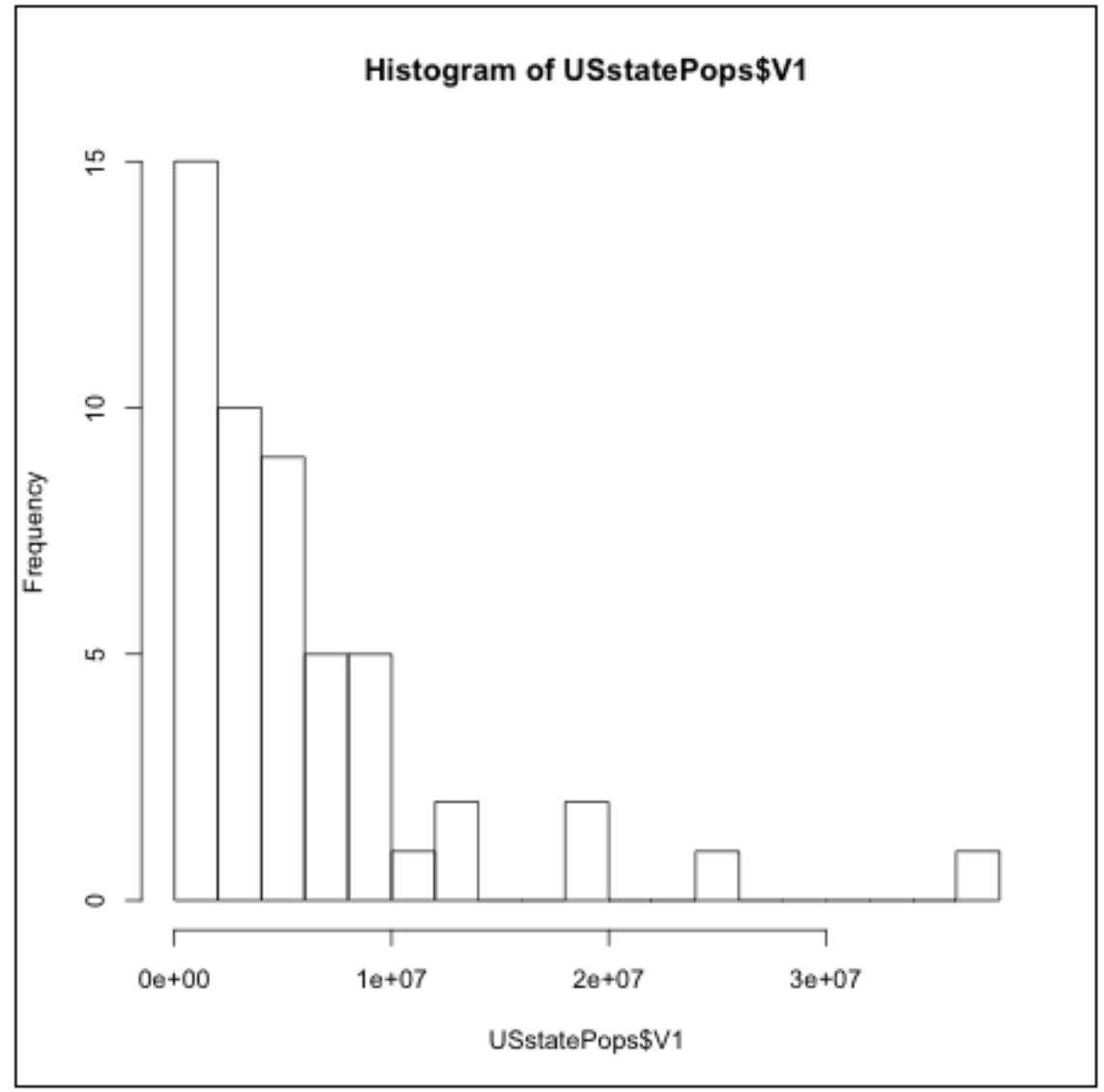
occur. The tallest bar is just to the right of this and it is nearly up to the 30 mark. To know what this tall bar represents, look along the X-axis (the horizontal axis at the bottom) and see that there is a tick mark for every two bars. We see scientific notation under each tick mark. The first tick mark is 1e+07, which translates to 10,000,000. So each new bar (or an empty space where a bar would go) goes up by five million in population. With these points in mind it should now be easy to see that there are nearly 30 states with populations under five million.

If you think about presidential elections, or the locations of schools and businesses, or how a single U.S. state might compare with other countries in the world, it is interesting to know that there are two really giant states and then lots of much smaller states. Once you have some practice reading histograms, all of the knowledge is available at a glance.

On the other hand there is something unsatisfying about this diagram. With over forty of the states clustered into the first couple of bars, there might be some more details hiding in there that we would like to know about. This concern translates into the number of bars shown in the histogram. There are eight shown here, so why did R pick eight?

The answer is that the hist() function has an algorithm or recipe for deciding on the number of categories/bars to use by default. The number of observations and the spread of the data and the amount of empty space there would be are all taken into account. Fortunately it is possible and easy to ask R to use more or fewer categories/bars with the "breaks" parameter, like this:

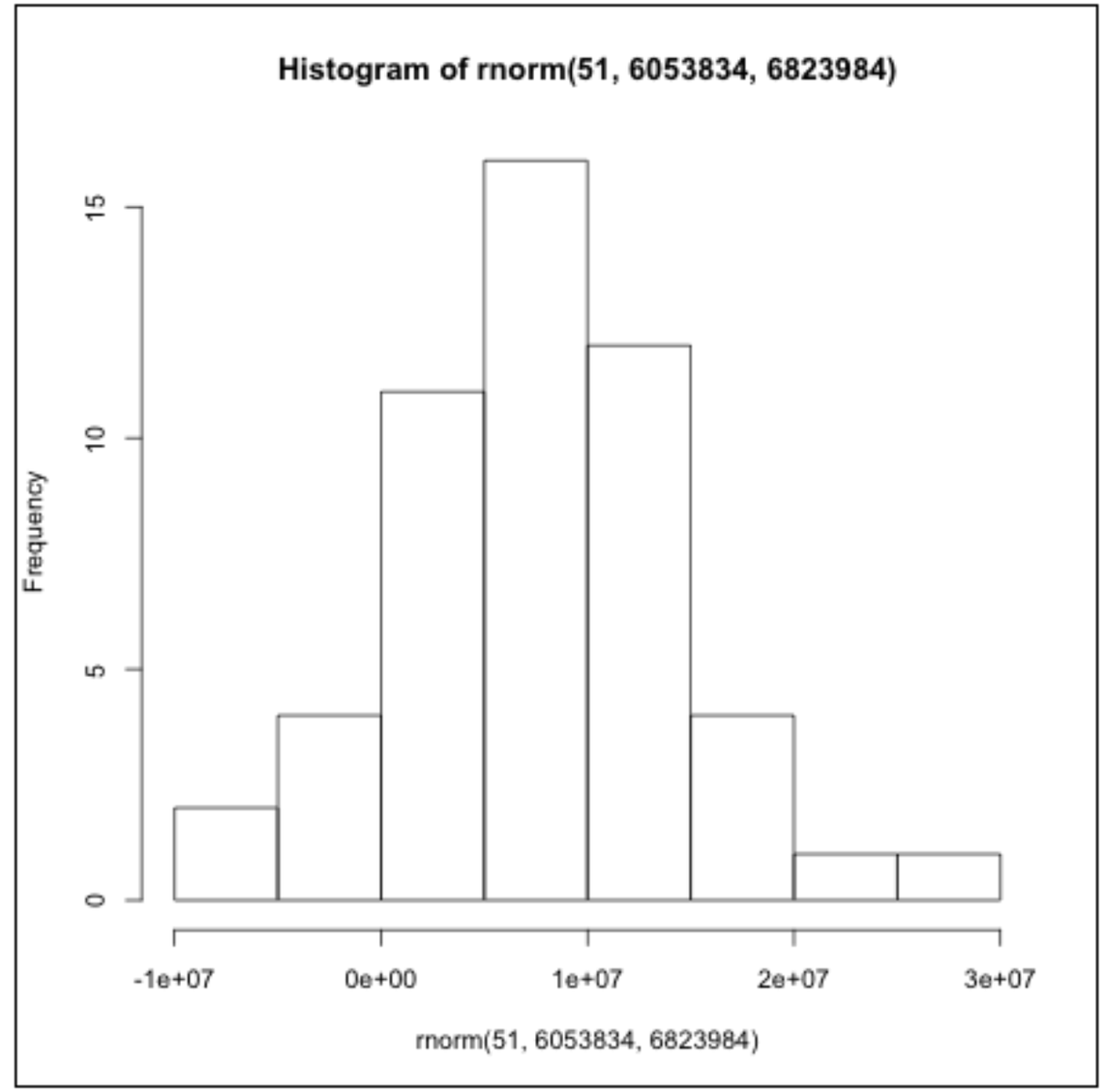
```
hist(USstatePops$V1, breaks=20)
```



This gives us five bars per tick mark or about two million for each bar. So the new histogram above shows very much the same pattern as before: 15 states with populations under two million. The pattern that you see here is referred to as a distribution. This is a distribution that starts off tall on the left and swoops downward quickly as it moves to the right. You might call this a "reverse-J" distribution because it looks a little like the shape a J makes, although flipped around vertically. More technically this could be referred to as a geometric distribution. We don't have to worry about

why it is called that at this stage, but we can speculate on why the distribution looks the way it does. First, you can't have a state with no people in it, or worse yet negative population. It just doesn't make any sense. So a state has to have at least a few people in it, and if you look through U.S. history every state began as a colony or a territory that had at least a few people in it. On the other hand, what does it take to grow really large in population? You need a lot of land, first of all, and then a good reason for lots of people to move there or lots of people to be born there. So there are lots of limits to growth: Rhode Island is too small too have a bazillion people in it and Alaska, although it has tons of land, is too cold for lots of people to want to move there. So all states probably started small and grew, but it is really difficult to grow really huge. As a result we have a distribution where most of the cases are clustered near the bottom of the scale and just a few push up higher and higher. But as you go higher, there are fewer and fewer states that can get that big, and by the time you are out at the end, just shy of 40 million people, there's only one state that has managed to get that big. By the way, do you know or can you guess what that humongous state is?

There are lots of other distribution shapes. The most common one that almost everyone has heard of is sometimes called the "bell" curve because it is shaped like a bell. The technical name for this is the normal distribution. The term "normal" was first introduced by Carl Friedrich Gauss (1777-1855), who supposedly called it that in a belief that it was the most typical distribution of data that one might find in natural phenomena. The following histogram depicts the typical bell shape of the normal distribution.



If you are curious, you might be wondering how R generated the histogram above, and, if you are alert, you might notice that the histogram that appears above has the word "rnorm" in a couple of places. Here's another of the cool features in R: it is incredibly easy to generate "fake" data to work with when solving problems or giving demonstrations. The data in this histogram were generated by R's rnorm() function, which generates a random data set that fits

the normal distribution (more closely if you generate a lot of data, less closely if you only have a little. Some further explanation of the rnorm() command will make sense if you remember that the state population data we were using had a mean of 6,053,834 and a standard deviation of 6,823,984. The command used to generate this histogram was:

```
hist(rnorm(51, 6043834, 6823984))
```

There are two very important new concepts introduced here. The first is a nested function call: The hist() function that generates the graph "surrounds" the rnorm() function that generates the new fake data. (Pay close attention to the parentheses!) The inside function, rnorm(), is run by R first, with the results of that sent directly and immediately into the hist() function.

The other important thing is the "arguments that" were "passed" to the rnorm() function. We actually already ran into arguments a little while ago with the read.DIF() and read.table() functions but we did not talk about them then. "Argument" is a term used by computer scientists to refer to some extra information that is sent to a function to help it know how to do its job. In this case we passed three arguments to rnorm() that it was expecting in this order: the number of observations to generate in the fake dataset, the mean of the distribution, and the standard deviation of the distribution. The rnorm() function used these three numbers to generate 51 random data points that, roughly speaking, fit the normal distribution. So the data shown in the histogram above are an approximation of what the distribution of state populations might look like if, instead of being reverse-J-shaped (geometric distribution), they were normally distributed.

The normal distribution is used extensively through applied statistics as a tool for making comparisons. For example, look at the rightmost bar in the previous histogram. The label just to the right of that bar is 3e+07, or 30,000,000. We already know from our real state population data that there is only one actual state with a population in excess of 30 million (if you didn't look it up, it is California). So if all of a sudden, someone mentioned to you that he or she lived in a state, *other than* California, that had 30 million people, you would automatically think to yourself, "Wow, that's unusual and I'm not sure I believe it." And the reason that you found it hard to believe was that you had a distribution to compare it to. Not only did that distribution have a characteristic shape (for example, J-shaped, or bell shaped, or some other shape), it also had a center point, which was the mean, and a "spread," which in this was case the standard deviation. Armed with those three pieces of information - the type/shape of distribution, an anchoring point, and a spread (also known as the amount of variability), you have a powerful tool for making comparisons.

In the next chapter we will conduct some of these comparisons to see what we can infer about the ways things are in general, based on just a subset of available data, or what statisticians call a sample.

Chapter Challenge

In this chapter, we used rnorm() to generate random numbers that closely fit a normal distribution. We also learned that the state population data was a "geometric" distribution. Do some research to find out what R function generates random numbers using the geometric distribution. Then run that function with the correct pa-

rameters to generate 51 random numbers (hint: experiment with different probability values). Create a histogram of these random numbers and describe the shape of the distribution.

Sources

http://en.wikipedia.org/wiki/Carl_Friedrich_Gauss

http://en.wikipedia.org/wiki/Francis_Galton

http://en.wikipedia.org/wiki/Karl_Pearson

http://en.wikipedia.org/wiki/Ronald_Fisher

http://en.wikipedia.org/wiki/William_Sealy_Gosset

http://en.wikipedia.org/wiki/Normal_distribution

http://stat.ethz.ch/R-manual/R-devel/library/utils/html/read.table.html

http://www.census.gov/popest/data/national/totals/2011/index.html

http://www.r-tutor.com/elementary-statistics/numerical-measures/standard-deviation

Review 6.1 Beer, Farms, and Peas

A bar graph that displays the frequencies of occurrence for a numeric variable is called a

- ✓ **A.** Histogram
- **B.** Pictogram
- **C.** Bar Graph
- **D.** Bar Chart

Check Answer

R Functions Used in This Chapter

read.DIF()	Reads data in interchange format
read.table()	Reads data table from external source
mean()	Calculate arithmetic mean
median()	Locate the median
mode()	Tells the data type/mode of a data object Note: This is NOT the statistical mode
var()	Calculate the sample variance
sd()	Calculate the sample standard deviation
hist()	Produces a histogram graphic

Test Yourself

If All Else Fails

In case you have difficulty with the read.DIF() or read.table() functions, the code shown below can be copied and pasted (or, in the worst case scenario, typed) into the R console to create the data set used in this chapter.

```
V1 <- c(4779736,710231,6392017,2915918,37253956,
5029196,3574097,897934,601723,18801310,9687653,
1360301,1567582,12830632,6483802,3046355,2853118,
4339367,4533372,1328361,5773552,6547629,9883640,
5303925,2967297,5988927,989415,1826341,2700551,
1316470,8791894,2059179,19378102,9535483,672591,
11536504,3751351,3831074,12702379,1052567,
4625364,814180,6346105,25145561,2763885,625741,
8001024,6724540,1852994,5686986,563626)

USstatePops <- data.frame(V1)
```

CHAPTER 7

Sample in a Jar

Sampling distributions are the conceptual key to statistical inference. Many approaches to understanding sampling distributions use examples of drawing marbles or gumballs from a large jar to illustrate the influences of randomness on sampling. Using the list of U.S. states illustrates how a non-normal distribution nonetheless has a normal sampling distribution of means.

Imagine a gum ball jar full of gumballs of two different colors, red and blue. The jar was filled from a source that provided 100 red gum balls and 100 blue gum balls, but when these were poured into the jar they got all mixed up. If you drew eight gumballs from the jar at random, what colors would you get? If things worked out perfectly, which they never do, you would get four red and four blue. This is half and half, the same ratio of red and blue that is in the jar as a whole. Of course, it rarely works out this way, does it? Instead of getting four red and four blue you might get three red and five blue or any other mix you can think of. In fact, it would be possible, though perhaps not likely, to get eight red gumballs. The basic situation, though, is that we really don't know what mix of red and blue we will get with one draw of eight gumballs. That's uncertainty for you, the forces of randomness affecting our sample of eight gumballs in unpredictable ways.

Here's an interesting idea, though, that is no help at all in predicting what will happen in any one sample, but is great at showing what will occur *in the long run*. Pull eight gumballs from the jar, count the number of red ones and then throw them back. We do not have to count the number of blue because 8 - #red = #blue. Mix up the jar again and then draw eight more gumballs and count the number of red. Keeping doing this many times. Here's an example of what you might get:

DRAW	# RED
1	5
2	3
3	6
4	2

Notice that the left column is just counting up the number of sample draws we have done. The right column is the interesting one because it is the count of the number of red gumballs in each particular sample draw. In this example, things are all over the place. In sample draw 4 we only have two red gumballs, but in sample draw 3 we have 6 red gumballs. But the most interesting part of this example is that if you *average* the number of red gumballs over all of the draws, the average comes out to *exactly four red gumballs* per draw, which is what we would expect in a jar that is half and half. Now this is a contrived example and we won't always get such a perfect result so quickly, but if you did four thousand draws instead of four, you would get pretty close to the perfect result.

This process of repeatedly drawing a subset from a "population" is called "sampling," and the end result of doing lots of sampling is a sampling distribution. Note that we are using the word population in the previous sentence in its statistical sense to refer to the totality of units from which a sample can be drawn. It is just a coincidence that our dataset contains the number of people in each state and that this value is also referred to as "population." Next we will get R to help us draw lots of samples from our U.S. state dataset.

Conveniently, R has a function called sample(), that will draw a random sample from a data set with just a single call. We can try it now with our state data:

```
> sample(USstatePops$V1,size=16,replace=TRUE)

[1] 4533372 19378102 897934 1052567 672591
18801310  2967297

[8]  5029196
```

As a matter of practice, note that we called the sample() function with three arguments. The first argument was the data source. For the second and third arguments, rather than rely on the order in which we specify the arguments, we have used "named arguments" to make sure that R does what we wanted. The size=16 argument asks R to draw a sample of 16 state data values. The replace=TRUE argument specifies a style of sampling which statisticians use very often to simplify the mathematics of their proofs. For us, sampling with or without replacement does not usually have any practical effects, so we will just go with what the statisticians typically do.

When we're working with numbers such as these state values, instead of counting gumball colors, we're more interested in finding out the average, or what you now know as the mean. So we could also ask R to calculate a mean() of the sample for us:

```
> mean(sample(USstatePops$V1,size=16, +
replace=TRUE))

[1] 8198359
```

There's the nested function call again. The output no longer shows the eight values that R sampled from the list of 51. Instead it used those eight values to calculate the mean and display that for us. If you have a good memory, or merely took the time to look in the last chapter, you will remember that the actual mean of our 51 observations is 6,053,834. So the mean that we got from this one sample of eight states is really not even close to the true mean value of our 51 observations. Are we worried? Definitely not! We know that when we draw a sample, whether it is gumballs or states, we will never hit the true population mean right on the head. We're interested not in any one sample, but in what happens over the long haul. So now we've got to get R to repeat this process for us, not once, not four times, but four hundred times or four thousand times. Like most programming languages, R has a variety of ways of repeating an activity. One of the easiest ones to use is the replicate() function. To start, let's just try four replications:

```
> replicate(4, mean(sample(USstatePops$V1,+
size=16,replace=TRUE)),simplify=TRUE)

[1] 10300486 11909337  8536523  5798488
```

Couldn't be any easier. We took the exact same command as before, which was a nested function to calculate the mean() of a random sample of eight states (shown above in bold). This time, we put that command inside the replicate() function so we could run it over and over again. The simplify=TRUE argument asks R to return the results as a simple vector of means, perfect for what we are trying to do. We only ran it four times, so that we would not have a big screen full of numbers. From here, though, it is easy to ramp up to repeating the process four hundred times. You can try that and see the output, but for here in the book we will encapsulate the whole replicate function inside another mean(), so that we can get the average of all 400 of the sample means. Here we go:

```
> mean(replicate(400, mean( +
sample(USstatePops$V1,size=16,replace=TRUE)),+
simplify=TRUE))

[1] 5958336
```

In the command above, the outermost mean() command is bolded to show what is different from the previous command. So, put into

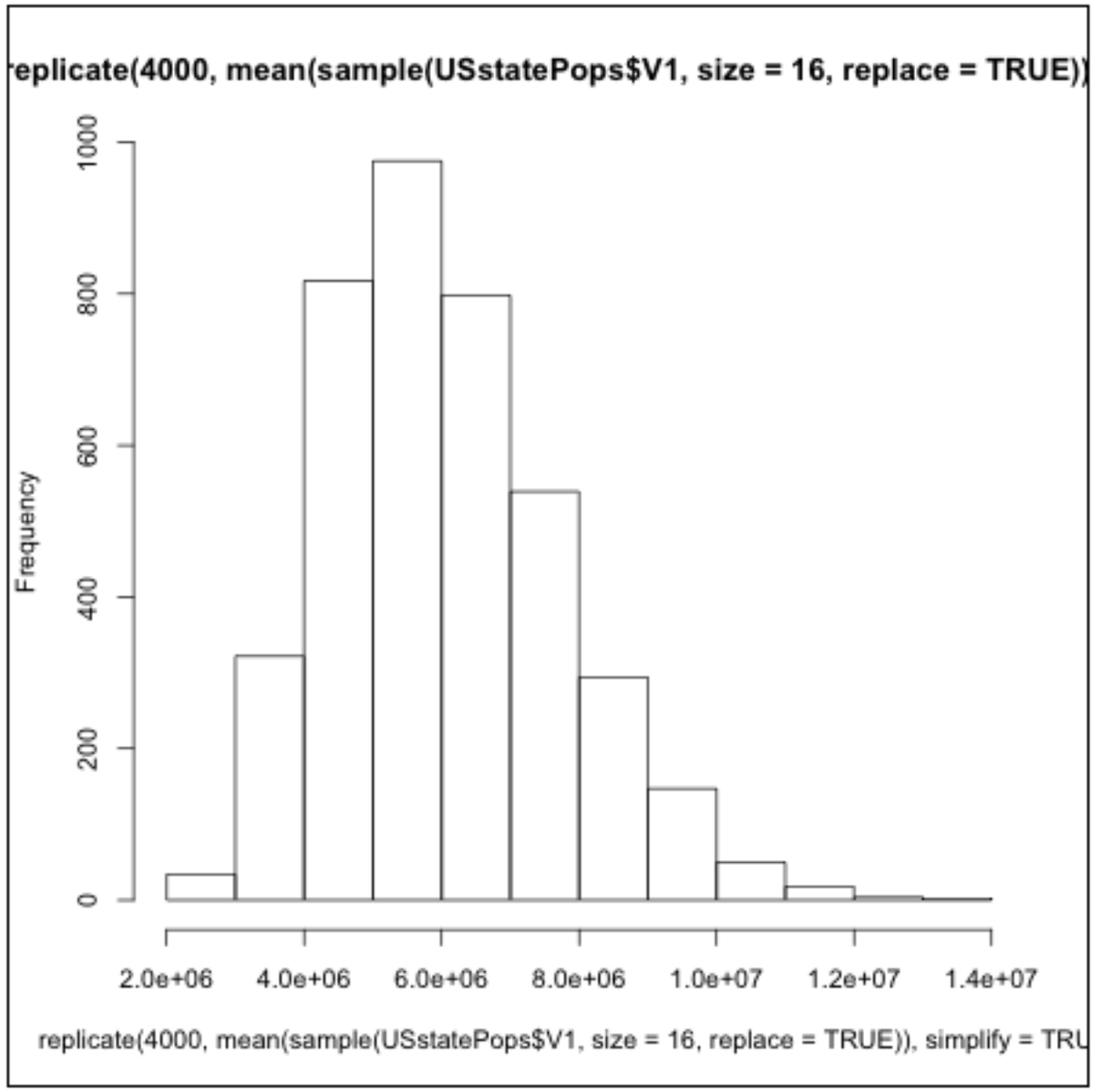
words, this deeply nested command accomplishes the following: a) Draw 400 samples of size n=8 from our full data set of 51 states; b) Calculate the mean from each sample and keep it in a list; c) When finished with the list of 400 of these means, calculate the mean of that list of means. You can see that the mean of four hundred sample means is 5,958,336. Now that is still not the exact value of the whole data set, but it is getting close. We're off by about 95,000, which is roughly an error of about 1.6% (more precisely, 95,498/6,053,834 = 1.58%. You may have also noticed that it took a little while to run that command, even if you have a fast computer. There's a lot of work going on there! Let's push it a bit further and see if we can get closer to the true mean for all of our data:

```
> mean(replicate(4000, mean( +
sample(USstatePops$V1,size=16,replace=TRUE)),+
simplify=TRUE))

[1] 6000972
```

Now we are even closer! We are now less than 1% away from the true population mean value. Note that the results you get may be a bit different, because when you run the commands, each of the 400 or 4000 samples that is drawn will be slightly different than the ones that were drawn for the commands above. What will not be much different is the overall level of accuracy.

We're ready to take the next step. Instead of summarizing our whole sampling distribution in a single average, let's look at the distribution of means using a histogram.

The histogram displays the complete list of 4000 means as frequencies. Take a close look so that you can get more practice reading frequency histograms. This one shows a very typical configuration that is almost bell-shaped, but still has a bit of "skewness" off to the right. The tallest, and therefore most frequent range of values is right near the true mean of 6,053,834.



By the way, were you able to figure out the command to generate this histogram on your own? All you had to do was substitute hist() for the outermost mean() in the previous command. In case you struggled, here it is:

```
hist(replicate(4000, mean( +
sample(USstatePops$V1,size=16,replace=TRUE)), +
simplify=TRUE))
```

This is a great moment to take a deep breath. We've just covered a couple hundred years of statistical thinking in just a few pages. In fact, there are two big ideas, "the law of large numbers" and the central limit theorem" that we have just partially demonstrated. These two ideas literally took mathematicians like Gerolamo Cardano (1501-1576) and Jacob Bernoulli (1654-1705) several centuries to figure out. If you look these ideas up, you may find a lot of bewildering mathematical details, but for our purposes, there are two really important take-away messages. First, if you run a statistical process a large number of times, it will converge on a stable result. For us, we knew what the average population was of the 50 states plus the District of Columbia. These 51 observations were our population, and we wanted to know how many smaller subsets, or samples, of size n=16 we would have to draw before we could get a good approximation of that true value. We learned that drawing one sample provided a poor result. Drawing 400 samples gave us a mean that was off by 1.5%. Drawing 4000 samples gave us a mean that was off by less than 1%. If we had kept going to 40,000 or 400,000 repetitions of our sampling process, we would have come extremely close to the actual average of 6,053,384.

Second, when we are looking at sample means, and we take the law of large numbers into account, we find that the distribution of sampling means starts to create a bell-shaped or normal distribution, and the center of that distribution, the mean of all of those sample means gets really close to the actual population mean. It gets closer faster for larger samples, and in contrast, for smaller samples you have to draw lots and lots of them to get really close. Just for fun, lets illustrate this with a sample size that is larger than 16. Here's a run that only repeats 100 times, but each time draws a sample of n=51 (equal in size to the population):

```
> mean(replicate(100, mean( +
sample(USstatePops$V1,size=51,replace=TRUE)),+
simplify=TRUE))
```

```
[1] 6114231
```

Now, we're only off from the true value of the population mean by about one tenth of one percent. You might be scratching your head now, saying, "Wait a minute, isn't a sample of 51 the same thing as the whole list of 51 observations?" This is confusing, but it goes back to the question of sampling with replacement that we examined a couple of pages ago (and that appears in the command above as replace=TRUE). Sampling with replacement means that as you draw out one value to include in your random sample, you immediately chuck it back into the list so that, potentially, it could get drawn again either immediately or later. As mentioned before, this practice simplifies the underlying proofs, and it does not cause any practical problems, other than head scratching. In fact, we could go even higher in our sample size with no trouble:

```
> mean(replicate(100, mean( +
sample(USstatePops$V1,size=120,replace=TRUE)), +
simplify=TRUE))
```

```
[1] 6054718
```

That command runs 100 replications using samples of size n=120. Look how close the mean of the sampling distribution is to the

population mean now! Remember that this result will change a little bit every time you run the procedure, because different random samples are being drawn for each run. But the rule of thumb is that the bigger your sample size, what statisticians call n, the closer your estimate will be to the true value. Likewise, the more trials you run, the closer your population estimate will be.

So, if you've had a chance to catch your breath, let's move on to making use of the sampling distribution. First, let's save one distribution of sample means so that we have a fixed set of numbers to work with:

```
SampleMeans <- replicate(10000, + mean(sample(US-
statePops$V1,size=120,+
replace=TRUE)),simplify=TRUE)
```

The bolded part is new. We're saving a distribution of sample means to a new vector called "SampleMeans". We should have 10,000 of them:

```
> length(SampleMeans)

[1] 10000
```

And the mean of all of these means should be pretty close to our population mean of 6,053,384:

```
> mean(SampleMeans)

[1] 6058718
```

You might also want to run a histogram on SampleMeans and see what the frequency distribution looks like. Right now, all we need to look at is a summary of the list of sample means:

```
> summary(SampleMeans)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.

3943000 5632000 6029000 6059000 6457000 9091000
```

If you need a refresher on the median and quartiles, take a look back at Chapter 3 - Rows and Columns.

This summary is full of useful information. First, take a look at the max and the min. The minimum sample mean in the list was 3,943,000. Think about that for a moment. Each sample that was drawn had n=120 observations in it. How could a sample have a mean that small when we know that the true mean is nearly 3 million higher? Rhode Island must have been drawn about 100 times in that sample! The answer comes from the randomness involved in sampling. If you run a process 10,000 times you are definitely going to end up with a few weird examples. Its almost like buying a lottery ticket. The vast majority of tickets are the usual - not a winner. Once in a great while, though, there is a very unusual ticket - a winner. Sampling is the same: The extreme events are unusual, but they do happen if you run the process enough times. The same goes for the maximum: at 9,091,000 the maximum sample mean is almost 3 million higher than the true mean.

At 6,029,000 the median is quite close to the mean, but not exactly the same because we still have a little bit of rightward skew (the "tail" on the high side is slightly longer than it should be because of the reverse J-shape of the original distribution). For practical purposes, we will consider the mean and the median to be interchangeable in this case. The median is very useful because it divides the sample exactly in half: 50%, or exactly 5000 of the sample means are larger than 6,029,000 and the other 50% are lower. So, if we

were to draw one more sample from the population it would have a fifty-fifty chance of being above the median. The quartiles help us to cut things up even more finely. The third quartile divides up the bottom 75% from the top 25%. So only 25% of the sample means are higher than 6,457,000. That means if we drew a new sample from the population that there is only a 25% chance that it will be larger than 6,457,000. Likewise, in the other direction, the first quartile tells us that there is only a 25% chance that a new sample would be less than 5,632,000.

There is a slightly different way of getting the same information from R that will prove more flexible for us in the long run. The quantile() function can show us the same information as the median and the quartiles, like this:

```
> quantile(SampleMeans, probs=c(0.25,0.50,0.75))

    25%     50%     75%

5631624 6028665 6457388
```

You will notice that the values are just slightly different, by less than one tenth of one percent, than those produced by the summary() function. These are actually more precise, although the less precise ones from summary() are fine for most purposes. One reason to use quantile() is that it lets us control exactly where we make the cuts. To get quartiles, we cut at 25% (0.25 in the command just above), at 50%, and at 75%. But what if we wanted instead to cut at 5% and 95%? Easy to do with quantile():

```
> quantile(SampleMeans, probs=c(0.05,0.95))

     5%     95%

5085394 7111750
```

So this result shows that, if we drew a new sample, there is only a 5% chance that the mean would be lower than 5,085,394. Likewise, there is only a 5% chance that the new sample mean would be higher than 7,111,750 (because 95% of the means in the sampling distribution are lower than that value).

Now let's put this knowledge to work. Here is a sample of the number of people in a certain area, where each these areas is some kind of a unit associated with the U.S.:

3,706,690
159,358
106,405
55,519
53,883

We can easily get these into R and calculate the sample mean:

```
> MysterySample <- c(3706690, 159358, 106405, +
   55519, 53883)

> mean(MysterySample)

[1] 816371
```

The mean of our mystery sample is 816,371. The question is, is this a sample of U.S. states or is it something else? Just on its own it would be hard to tell. The first observation in our sample has more people in it than Kansas, Utah, Nebraska, and several other states. We also know from looking at the distribution of raw population data from our previous example that there are many, many states that are quite small in the number of people. Thanks to the work

we've done earlier in this chapter, however, we have an excellent basis for comparison. We have the sampling distribution of means, and it is fair to say that if we get a new mean to look at, and the new mean is way out in the extreme areas of the sample distribution, say, below the 5% mark or above the 95% mark, then it seems much less likely that our MysterySample is a sample of states.

In this case, we can see quite clearly that 816,371 is on the extreme low end of the sampling distribution. Recall that when we ran the quantile() command we found that only 5% of the sample means in the distribution were smaller than 5,085,394.

In fact, we could even play around with a more stringent criterion:

```
> quantile(SampleMeans, probs=c(0.01,0.99))

     1%     99%

4739252 7630622
```

This quantile() command shows that only 1% of all the sample means are lower than 4,739,252. So our MysterySample mean of 816,371 would definitely be a very rare event, if it were truly a sample of states. From this we can infer, tentatively but based on good statistical evidence, that our MysterySample is *not* a sample of states. The mean of MysterySample is just too small to be very likely to be a sample of states.

And this is in fact correct: MysterySample contains the number of people in five different U.S. territories, including Puerto Rico in the Caribbean and Guam in the Pacific. These territories are land masses and groups of people associated with the U.S., but they are not states and they are different in many ways than states. For one thing they are all islands, so they are limited in land mass. Among the U.S. states, only Hawaii is an island, and it is actually bigger than 10 of the states in the continental U.S. The important thing to take away is that the characteristics of this group of data points, notably the mean of this sample, was sufficiently different from a known distribution of means that we could make an inference that the sample *was not drawn from the original population of data*.

This reasoning is the basis for virtually all statistical inference. You construct a comparison distribution, you mark off a zone of extreme values, and you compare any new sample of data you get to the distribution to see if it falls in the extreme zone. If it does, you tentatively conclude that the new sample was obtained from some other source than what you used to create the comparison distribution.

If you feel a bit confused, take heart. There's 400-500 years of mathematical developments represented in that one preceding paragraph. Also, before we had cool programs like R that could be used to create and analyze actual sample distributions, most of the material above was taught as a set of formulas and proofs. Yuck! Later in the book we will come back to specific statistical procedures that use the reasoning described above. For now, we just need to take note of three additional pieces of information.

First, we looked at the mean of the sampling distribution with mean() and we looked at its shaped with hist(), but we never quantified the spread of the distribution:

```
> sd(SampleMeans)

[1] 621088.1
```

This shows us the standard deviation of the distribution of sampling means. Statisticians call this the "standard error of the mean." This chewy phrase would have been clearer, although longer, if it had been something like this: "the standard deviation of the distribution of sample means for samples drawn from a population." Unfortunately, statisticians are not known for giving things clear labels. Suffice to say that when we are looking at a distribution and each data point in that distribution is itself a representation of a sample (for example, a mean), then the standard deviation is referred to as the standard error.

Second, there is a shortcut to finding out the standard error that does not require actually constructing an empirical distribution of 10,000 (or any other number) of sampling means. It turns out that the standard deviation of the original raw data and the standard error are closely related by a simple bit of algebra:

```
> sd(USstatePops$V1)/sqrt(120)

[1] 622941.7
```

The formula in this command takes the standard deviation of the original state data and divides it by the square root of the sample size. Remember three of four pages ago when we created the SampleMeans vector by using the replicate() and sample() commands, that we used a sample size of n=120. That's what you see in the formula above, inside of the sqrt() function. In R, and other software sqrt() is the abbreviation for "square root" and not for "squirt" as you might expect. So if you have a set of observations and you calculate their standard deviation, you can also calculate the standard error for a distribution of means (each of which has the same sample size), just by dividing by the square root of the sample size. You may notice that the number we got with the shortcut was slightly larger than the number that came from the distribution itself, but the difference is not meaningful (and only arrises because of randomness in the distribution). Another thing you may have noticed is that the larger the sample size, the smaller the standard error. This leads to an important rule for working with samples: the bigger the better.

The last thing is another shortcut. We found out the 5% cut point and the 95% cut point by constructing the sampling distribution and then using quantile to tell us the actual cuts. You can also find those cut points just using the mean and the standard error. Two standard errors down from the mean is the 5% cut point and two standard errors up from the mean is the 95% cut point.

```
> StdError<-sd(USstatePops$V1)/sqrt(120)

> CutPoint5<-mean(USstatePops$V1)-(2 * StdError)

> CutPoint95<-mean(USstatePops$V1)+(2 * StdError)

> CutPoint5

[1] 4807951

> CutPoint95

[1] 7299717
```

You will notice again that these are slightly different from what we calculated with the quantile() function using the empirical distribution. The differences arise because of the randomness in the distribution that we constructed. We could easily reduce those discrepancies by using a larger sample size and by having more replications included in the sampling distribution.

To summarize, with a data set that includes 51 data points with the numbers of people in states, and a bit of work using R to construct a distribution of sampling means, we have learned the following:

- Run a statistical process a large number of times and you get a consistent pattern of results.
- Taking the means of a large number of samples and plotting them on a histogram shows that the sample means are fairly well normally distributed and that the center of the distribution is very, very close to the mean of the original raw data.
- This resulting distribution of sample means can be used as a basis for comparisons. By making cut points at the extreme low and high ends of the distribution, for example 5% and 95%, we have a way of comparing any new information we get.
- If we get a new sample mean, and we find that it is in the extreme zone defined by our cut points, we can tentatively conclude that the sample that made that mean is a different kind of thing than the samples that made the sampling distribution.
- A shortcut and more accurate way of figuring the cut points involves calculating the "standard error" based on the standard deviation of the original raw data.

We're not statisticians at this point, but the process of reasoning based on sampling distributions is at the heart of inferential statistics, so if you have followed the logic presented in this chapter, you have made excellent progress towards being a competent user of applied statistics.

Chapter Challenge

Collect a sample consisting of at least 20 data points and construct a sampling distribution. Calculate the standard error and use this to calculate the 5% and 95% distribution cut points. The data points you collect should represent instances of the same phenomenon. For instance, you could collect the prices of 20 textbooks, or count the number of words in each of 20 paragraphs.

Sources

http://en.wikipedia.org/wiki/Central_limit_theorem

http://en.wikipedia.org/wiki/Gerolamo_Cardano

http://en.wikipedia.org/wiki/Jacob_Bernoulli

http://en.wikipedia.org/wiki/Law_of_large_numbers

http://en.wikipedia.org/wiki/List_of_U.S._states_and_territories_by_population

http://www.khanacademy.org/math/statistics/v/central-limit-theorem

R Commands Used in This Chapter

length() - The number of elements in a vector

mean() - The arithmetic mean or average of a set of values

quantile() - Calculates cut points based on percents/proportions

replicate() - Runs an expression/calculation many times

sample() - Chooses elements at random from a vector

sd() - Calculates standard deviation

sqrt() - Calculates square root

summary() - Summarizes contents of a vector

CHAPTER 8

Big Data? Big Deal!

In 2011-2012 the technology press contained many headlines about big data. What makes data big, and why is this bigness important? In this chapter, we discuss some of the real issues behind these questions. Armed with information from the previous chapter concerning sampling, we can give more thought to how the size of a data set affects what we do with the data.

MarketWatch (a Wall Street Journal Service) recently published an article with the title, "Big Data Equals Big Business Opportunity Say Global IT and Business Professionals," and the subtitle, "70 Percent of Organizations Now Considering, Planning or Running Big Data Projects According to New Global Survey." The technology news has been full of similar articles for several years. Given the number of such articles it is hard to resist the idea that "big data" represents some kind of revolution that has turned the whole world of information and technology topsy-turvy. But is this really true? Does "big data" change everything?

Let's list a few things which are certainly pretty accurate:

1. The decline in the price of sensors (like barcode readers) and other technology over recent decades has made it cheaper and easier to collect a lot more data.
2. Similarly, the declining cost of storage has made it practical to keep lots of data hanging around, regardless of its quality or usefulness.
3. Many people's attitudes about privacy seem to have accommodated the use of Facebook and other platforms where we reveal lots of information about ourselves.
4. Researchers have made significant advances in the "machine learning" algorithms that form the basis of many data mining techniques.
5. When a data set gets to a certain size (into the range of thousands of rows), conventional tests of statistical significance are meaningless, because even the most tiny and trivial results (or effect sizes, as statisticians call them) are statistically significant.

Keeping these points in mind, there are also a number of things that have not changed throughout the years:

A. Garbage in, garbage out: The usefulness of data depends heavily upon how carefully and well it was collected. After data were collected, the quality depends upon how much attention was paid to suitable pre-processing: data cleaning and data screening.

B. Bigger equals weirder: If you are looking for anomalies - rare events that break the rules - then larger is better. Low frequency events often do not appear until a data collection goes on for a long time and/or encompasses a large enough group of instances to gran one of the bizarre cases.

C. Linking adds potential: Standalone datasets are inherently limited by whatever variables are available. But if those data can be linked to some other data, all of a sudden new vistas may open up. No guarantees, but the more you can connect records here to other records over there, the more potential findings you have.

Items on both of the lists above are considered pretty commonplace and uncontroversial. Taken together, however, they do shed some light on the question of how important "big data" might be. We have had lots of historical success using conventional statistics to examine modestly sized (i.e., 1000 rows or less) datasets for statistical regularities. Everyone's favorite basic statistic, the Student's t-test, is essential a test for differences in the central tendency of two groups. If the data contain regularities such that one group is notably different from another group, a t-test shows it to be so.

Big data does not help us with these kinds of tests. We don't even need a thousand records for many conventional statistical compari-

sons, and having a million or a hundred million records won't make our job any easier (it will just take more computer memory and storage).

On the other hand, if we are looking for needles in haystacks, it makes sense to look (as efficiently as possible) through the biggest possible haystack we can find, because it is much more likely that a big haystack will contain at least one needle and maybe more. Keeping in mind the advances in machine learning that have occurred over recent years, we begin to have an idea that good tools together with big data and interesting questions about unusual patterns could indeed provide some powerful new insights.

Let's couple this optimism with three very important cautions. The first caution is that the more complex our data are, the more difficult it will be to ensure that the data are "clean" and suitable for the purpose we plan for them. A dirty data set is worse in some ways than no data at all because we may put a lot of time and effort into finding an insight and find nothing. Even more problematic, we may put a lot of time and effort and find a result that is simply wrong!

The second caution is that rare and unusual events or patterns are almost always by their nature highly unpredictable. Even with the best data we can imagine and plenty of variables, we will almost always have a lot of trouble accurately enumerating all of the causes of an event. The data mining tools may show us a pattern, and we may even be able to replicate the pattern in some new data, but we may never be confident that we have understood the pattern to the point where we believe we can isolate, control, or understand the causes. Predicting the path of hurricanes provides a great example here: despite decades of advances in weather instrumentation, forecasting, and number crunching, meteorologists still have great difficulty predicting where a hurricane will make landfall or how hard the winds will blow when it gets there. The complexity and unpredictability of the forces at work make the task exceedingly difficult.

The third caution is about linking data sets. Item C above suggests that linkages may provide additional value. With every linkage to a new data set, however, we also increase the complexity of the data and the likelihood of dirty data and resulting spurious patterns. In addition, although many companies seem less and less concerned about the idea, the more we link data about living people (e.g., consumers, patients, voters, etc.) the more likely we are to cause a catastrophic loss of privacy. Even if you are not a big fan of the importance of privacy on principle, it is clear that security and privacy failures have cost companies dearly both in money and reputation. Today's data innovations for valuable and acceptable purposes maybe tomorrow's crimes and scams.

Putting this altogether, we can take a sensible position that **high quality** data, in abundance, together with tools used by intelligent analysts, may provide worthwhile benefits in the commercial sector, in education, in government, and in other areas. The focus of our efforts as data scientists, however, should not be on achieving the largest possible data sets, but rather on getting the right data and the right amount of data for the purpose we intend.

Sources

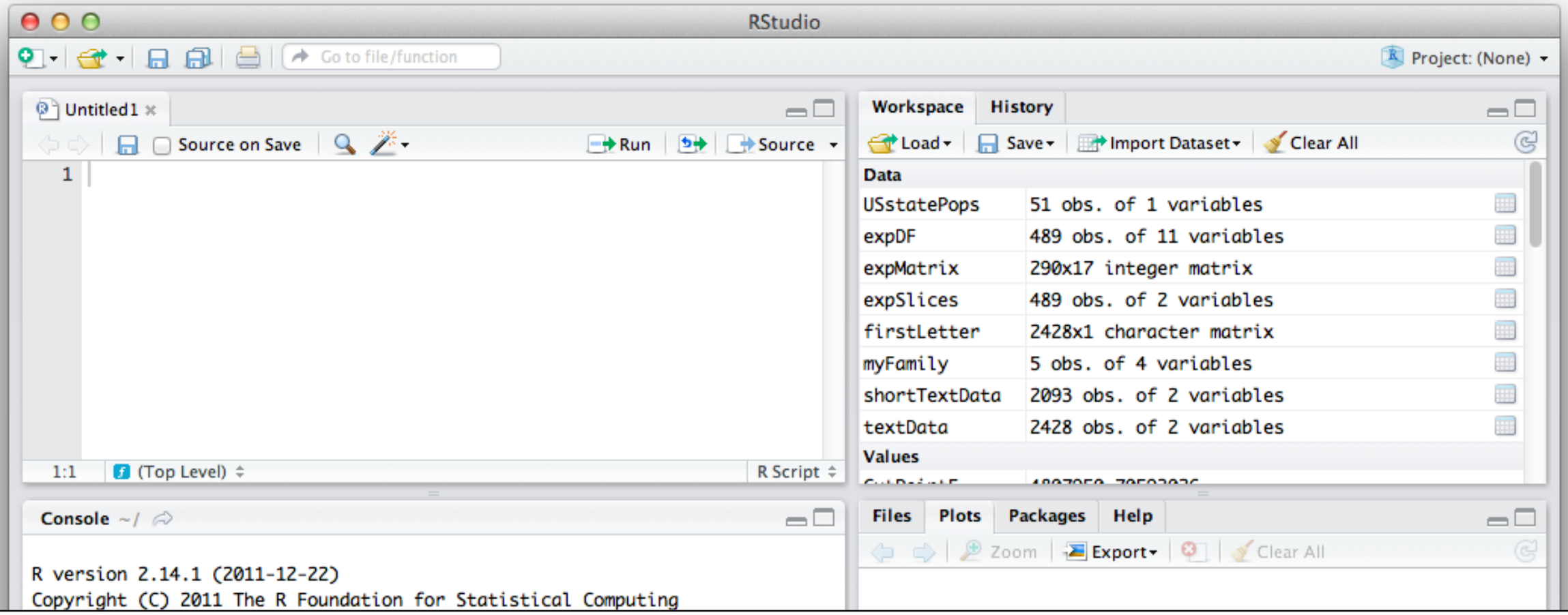
http://aqua.nasa.gov/doc/pubs/Wx_Forecasting.pdf

http://en.wikipedia.org/wiki/Big_data

http://www.marketwatch.com/story/big-data-equals-big-business-opportunity-say-global-it-and-business-professionals-2012-05-14

CHAPTER 9

Onward with R-Studio



As an open source program with an active user community, R enjoys constant innovation thanks to the dedicated developers who work on it. One useful innovation was the development of R-Studio, a beautiful frame to hold your copy of R. This chapter walks through the installation of R-Studio and introduces "packages," the key to the extensibility of R.

Joseph J. Allaire is a serial entrepreneur, software engineer, and the originator of some remarkable software products including "ColdFusion," which was later sold to the web media tools giant Macromedia and Windows Live Writer, a Microsoft blogging tool. Starting in 2009, Allaire began working with a small team to develop an open source program that enhances the usability and power of R.

As mentioned in previous chapters, R is an open source program, meaning that the source code that is used to create a copy of R to run on a Mac, Windows, or Linux computer is available for all to inspect and modify. As with many open source projects, there is an active community of developers who work on R, both on the basic program itself and the many pieces and parts that can be added onto the basic program. One of these add-ons is R-Studio. R-Studio is an Integrated Development Environment, abbreviated as IDE. Every software engineer knows that if you want to get serious about building something out of code, you must use an IDE. If you think of R as a piece of canvas rolled up and laying on the floor, R-Studio is like an elegant picture frame. R hangs in the middle of R studio, and like any good picture frame, enhances our appreciation of what is inside it.

The website for R-studio is http://www.rstudio.org/ and you can inspect the information there at any time. For most of the rest of this chapter, if you want to follow along with the installation and use of R-Studio, you will need to work on a Mac, Windows, or Linux computer.

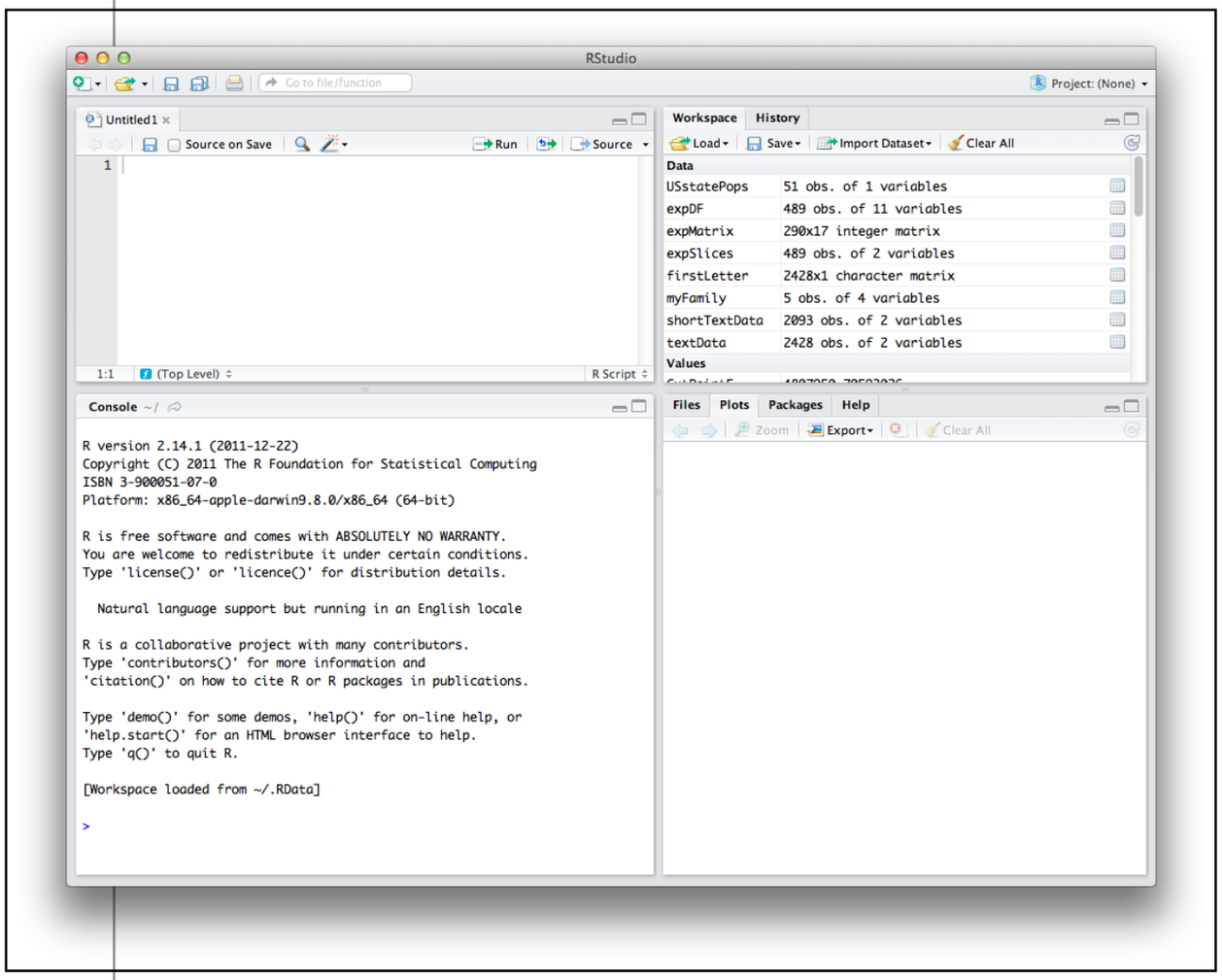
Before we start that, let's consider why we need an IDE to work with R. In the previous chapters, we have typed a variety of commands into R, using what is known as the "R console." Console is an old technology term that dates back to the days when computers were so big that they each occupied their own air conditioned room. Within that room there was often one "master control station" where a computer operator could do just about anything to control the giant computer by typing in commands. That station was known as the console. The term console is now used in many cases to refer to any interface where you can directly type in commands. We've typed commands into the R console in an effort to learn about the R language as well as to illustrate some basic principles about data structures and statistics.

If we really want to "do" data science, though, we can't sit around typing commands every day. First of all, it will become boring very fast. Second of all, whoever is paying us to be a data scientist will get suspicious when he or she notices that we are *retyping* some of the commands we typed yesterday. Third, and perhaps most important, it is way too easy to make a mistake - to create what computer scientists refer to as a bug - if you are doing every little task by hand. For these reasons, one of our big goals within this book is to create something that is reusable: where we can do a few clicks or type a couple of things and unleash the power of many processing steps. Using an IDE, we can build these kinds of reusable pieces. The IDE gives us the capability to open up the process of creation, to peer into the component parts when we need to, and to close the hood and hide them when we don't. Because we are working with data, we also need a way of closely inspecting the data, both its contents and its structure. As you probably noticed, it gets pretty tedious doing this at the R console, where almost every piece of output is a chunk of text and longer chunks scroll off the screen before you can see them. As an IDE for R, R-Studio allows us to control and

monitor both our code and our text in a way that supports the creation of reusable elements.

Before we can get there, though, we have to have R-Studio installed on a computer. Perhaps the most challenging aspect of installing R-Studio is having to install R first, but if you've already done that in chapter 2, then R-Studio should be a piece of cake. Make sure that you have the latest version of R installed before you begin with the installation of R-studio. There is ample documentation on the R-studio website, http://www.rstudio.org/, so if you follow the instructions there, you should have minimal difficulty. If you reach a page where you are asked to choose between installing R-studio server and installing R-studio as a desktop application on your computer, choose the latter. We will look into R-studio server a little later, but for now you want the desktop/single user version. If you run into any difficulties or you just want some additional guidance about R-studio, you may want to have a look at the book entitled, *Getting Started with R-studio*, by John Verzani (2011, Sebastopol, CA: O'Reilly Media). The first chapter of that book has a general orientation to R and R-studio as well as a guide to installing and updating R-studio. There is also a YouTube video that introduces R-studio here:
http://www.youtube.com/watch?v=7sAmqkZ3Be8
Be aware if you search for other YouTube videos that there is a disk recovery program as well a music group that share the R-Studio name: You will get a number of these videos if you search on "R-Studio" without any other search terms.

Once you have installed R-Studio, you can run it immediately in order to get started with the activities in the later parts of this chapter. Unlike other introductory materials, we will not walk through all of the different elements of the R-Studio screen. Rather, as we need each feature we will highlight the new aspect of the application. When you run R-Studio, you will see three or four sub-windows. Use the File menu to click "New" and in the sub-menu for "New" click "R Script." This should give you a screen that looks something like this:



The upper left hand "pane" (another name for a sub-window) displays a blank space under the tab title "Untitled1." Click in that pane and type the following:

```
MyMode <- function(myVector)

{

  return(myVector)

}
```

You have just created your first "function" in R. A function is a bundle of R code that can be used over and over again without having to retype it. Other programming languages also have functions. Other words for function are "procedure" and "subroutine," although these terms can have a slightly different meaning in other languages. We have called our function "MyMode." You may remember from a couple of chapters that the basic setup of R does not have a statistical mode function in it, even though it does have functions for the two other other common central tendency statistics, mean() and median(). We're going to fix that problem by creating our own mode function. Recall that the mode function should count up how many of each value is in a list and then return the value that occurs most frequently. That is the definition of the statistical mode: the most frequently occurring item in a vector of numbers.

A couple of other things to note: The first is the "myVector" in parentheses on the first line of our function. This is the "argument" or input to the function. We have seen arguments before when we called functions like mean() and median(). Next, note the curly braces that are used on the second and final lines. These curly braces hold together all of the code that goes in our function. Finally, look at the return() right near the end of our function. This return() is where we send back the result of what our function accomplished. Later on when we "call" our new function from the R console, the result that we get back will be whatever is in the parentheses in the return().

Based on that explanation, can you figure out what MyMode() does in this primitive initial form? All it does is return whatever we give it in myVector, completely unchanged. By the way, this is a common way to write code, by building up bit by bit. We can test out what we have each step of the way. Let's test out what we have accomplished so far. First, let's make a very small vector of data to work with. In the lower left hand pane of R-studio you will notice that we have a regular R console running. You can type commands into this console, just like we did in previous chapters just using R:

```
> tinyData <- c(1,2,1,2,3,3,3,4,5,4,5)

> tinyData

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

Then we can try out our new MyMode() function:

```
> MyMode(tinyData)

Error: could not find function "MyMode"
```

Oops! R doesn't know about our new function yet. We typed our MyMode() function into the code window, but we didn't tell R about it. If you look in the upper left pane, you will see the code for MyMode() and just above that a few small buttons on a tool bar. One of the buttons looks like a little right pointing arrow with

the word "Run" next to it. First, use your mouse to select all of the code for MyMode(), from the first M all the way to the last curly brace. Then click the Run button. You will immediately see the same code appear in the R console window just below. If you have typed everything correctly, there should be no errors or warnings. Now R knows about our MyMode() function and is ready to use it. Now we can type:

```
> MyMode(tinyData)

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

This did exactly what we expected: it just echoed back the contents of tinyData. You can see from this example how parameters work, too. in the command just above, we passed in tinyData as the input to the function. While the function was working, it took what was in tinyData and copied it into myVector for use inside the function. Now we are ready to add the next command to our function:

```
MyMode <- function(myVector)

{

  uniqueValues <- unique(myVector)

  return(uniqueValues)

}
```

Because we made a few changes, the whole function appears again above. Later, when the code gets a little more complicated, we will just provide one or two lines to add. Let's see what this code does. First, don't forget to select the code and click on the Run button. Then, in the R console, try the MyMode() command again:

```
> MyMode(tinyData)

[1] 1 2 3 4 5
```

Pretty easy to see what the new code does, right? We called the unique() function, and that returned a list of unique values that appeared in tinyData. Basically, unique() took out all of the redundancies in the vector that we passed to it. Now let's build a little more:

```
MyMode <- function(myVector)

{

  uniqueValues <- unique(myVector)

  uniqueCounts <- tabulate(myVector)

  return(uniqueCounts)

}
```

Don't forget to select all of this code and Run it before testing it out. This time when we pass tinyData to our function we get back another list of five elements, but this time it is the count of how many times each value occurred:

```
> MyMode(tinyData)

[1] 2 2 3 2 2
```

Now we're basically ready to finish our MyMode() function, but let's make sure we understand the two pieces of data we have in uniqueValues and uniqueCounts:

In the table below we have lined up a row of the elements of uniqueValues just above a row of the counts of how many of each of those values we have. Just for illustration purposes, in the top/

INDEX	1	2	3	4	5
uniqueValues	1	2	3	4	5
uniqueCounts	2	2	3	2	2

label row we have also shown the "index" number. This index number is the way that we can "address" the elements in either of the variables that are shown in the rows. For instance, element number 4 (index 4) for uniqueValues contains the number four, whereas element number four for uniqueCounts contains the number two. So if we're looking for the most frequently occurring item, we should look along the bottom row for the largest number. When we get there, we should look at the index of that cell. Whatever that index is, if we look in the same cell in uniqueValues, we will have the value that occurs most frequently in the original list. In R, it is easy to accomplish what was described in the last sentence with a single line of code:

```
uniqueValues[which.max(uniqueCounts)]
```

The which.max() function finds the index of the element of uniqueCounts that is the largest. Then we use that index to address uniqueValues with square braces. The square braces let us get at any of the elements of a vector. For example, if we asked for uniqueValues[5] we would get the number 5. If we add this one list of code to our return statement, our function will be finished:

```
MyMode <- function(myVector)
{
  uniqueValues <- unique(myVector)
  uniqueCounts <- tabulate(myVector)
  return(uniqueValues[which.max(uniqueCounts)])
}
```

We're now ready to test out our function. Don't forget to select the whole thing and run it! Otherwise R will still be remembering our old one. Let's ask R what tinyData contains, just to remind ourselves, and then we will send tinyData to our MyMode() function:

```
> tinyData
 [1] 1 2 1 2 3 3 3 4 5 4 5
> MyMode(tinyData)
[1] 3
```

Hooray! It works. Three is the most frequently occurring value in tinyData. Let's keep testing and see what happens:

```
> tinyData<-c(tinyData,5,5,5)
> tinyData
 [1] 1 2 1 2 3 3 3 4 5 4 5 5 5 5
> MyMode(tinyData)
[1] 5
```

It still works! We added three more fives to the end of the tinyData vector. Now tinyData contains five fives. MyMode() properly reports the mode as five. Hmm, now let's try to break it:

```
> tinyData

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

> MyMode(tinyData)

[1] 1
```

This is interesting: Now tinyData contains five ones and five fives. MyMode() now reports the mode as one. This turns out to be no surprise. In the documentation for which.max() it says that this function will return the first maximum it finds. So this behavior is to be expected. Actually, this is always a problem with the statistical mode: there can be more than one mode in a data set. Our MyMode() function is not smart enough to realize this, not does it give us any kind of warning that there are multiple modes in our data. It just reports the first mode that it finds.

Here's another problem:

```
> tinyData<-c(tinyData,9,9,9,9,9,9,9)

> MyMode(tinyData)

[1] NA

> tabulate(tinyData)

 [1] 5 2 3 2 5 0 0 0 7
```

In the first line, we stuck a bunch of nines on the end of tinyData. Remember that we had no sixes, sevens, or eights. Now when we run MyMode() it says "NA," which is R's way of saying that something went wrong and you are getting back an empty value. It is probably not obvious why things went whacky until we look at the last command above, tabulate(tinyData). Here we can see what happened: when it was run inside of the MyMode() function, tabulate() generated a longer list than we were expecting, because it added zeroes to cover the sixes, sevens, and eights that were not there. The maximum value, out at the end is 7, and this refers to the number of nines in tinyData. But look at what the unique() function produces:

```
> unique(tinyData)

[1]  1  2  3  4  5  9
```

There are only six elements in this list, so it doesn't match up as it should (take another look at the table on the previous page and imagine if the bottom row stuck out further than the row just above it). We can fix this with the addition of the match() function to our code:

```
MyMode <- function(myVector)

{

  uniqueValues <- unique(myVector)

  uniqueCounts <- tabulate( +
                 match(myVector,uniqueValues))


  return(uniqueValues[which.max(uniqueCounts)])

}
```

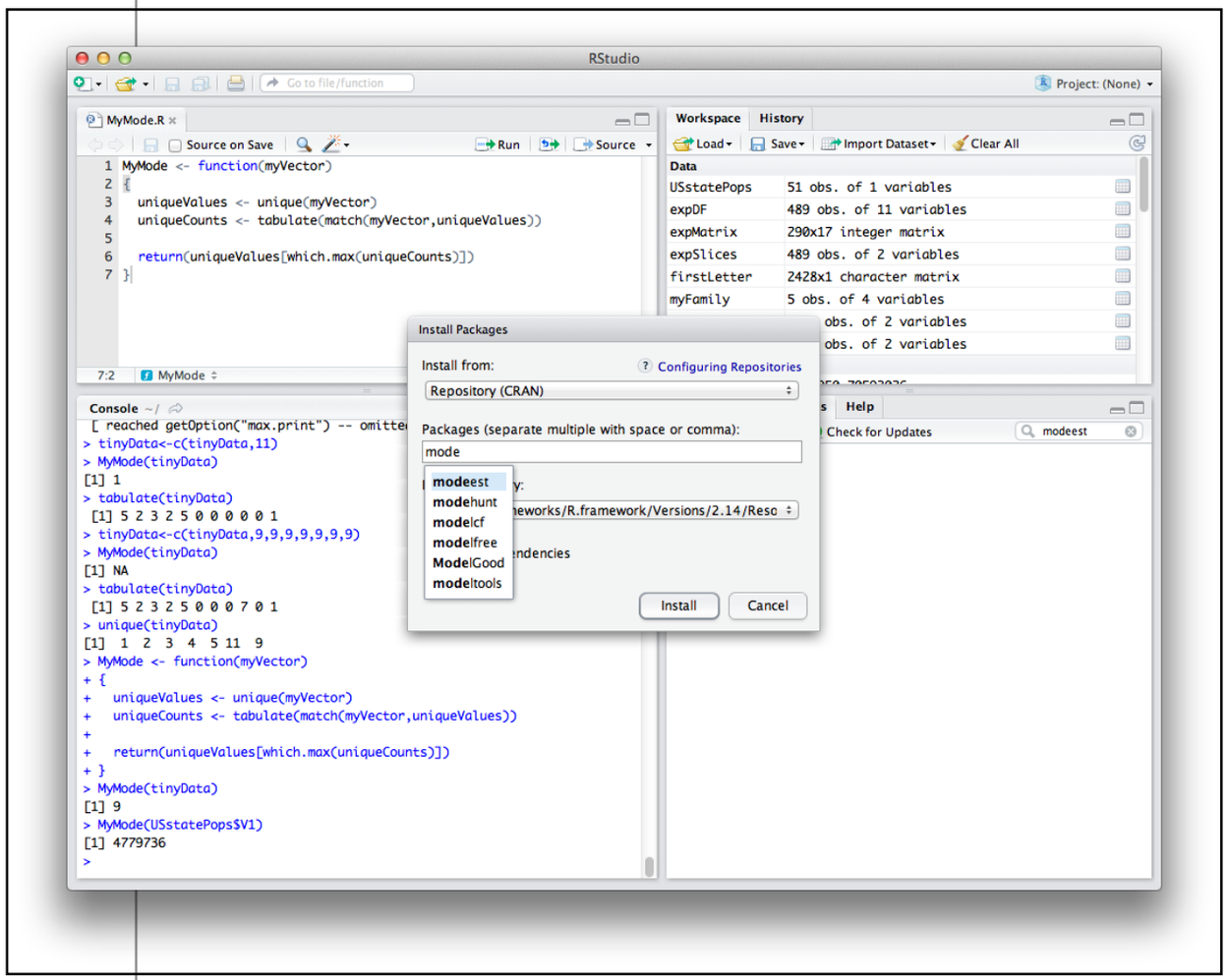
The new part of the code is in bold. Now instead of tabulating every possible value, including the ones for which we have no data, we only tabulate those items where there is a "match" between the list of unique values and what is in myVector. Now when we ask MyMode() for the mode of tinyData we get the correct result:

```
> MyMode(tinyData)

[1] 9
```

Aha, now it works the way it should. After our last addition of seven nines to the data set, the mode of this vector is correctly reported as nine.

Before we leave this activity, make sure to save your work. Click anywhere in the code window and then click on the File menu and then on Save. You will be prompted to choose a location and provide a filename. You can call the file MyMode, if you like. Note that R adds the "R" extension to the filename so that it is saved as MyMode.R. You can open this file at any time and rerun the MyMode() function in order to define the function in your current working version of R.

A couple of other points deserve attention. First, notice that when we created our own function, we had to do some testing and repairs to make sure it ran the way we wanted it to. This is a common situation when working on anything related to computers, including spreadsheets, macros, and pretty much anything else that requires precision and accuracy. Second, we introduced at least four new functions in this exercise, including unique(), tabulate(), match(), and which.max(). Where did these come from and how did we know? R has so many functions that it is very difficult to memorize them all. There's almost always more than one way to do something, as well. So it can be quite confusing to create a new function, if you don't know all of the ingredients and there's no one way to solve a particular problem. This is where the community comes in. Search online and you will find



dozens of instances where people have tried to solve similar problems to the one you are solving, and you will also find that they have posted the R code for their solutions. These code fragments are free to borrow and test. In fact, learning from other people's ex-

amples is a great way to expand your horizons and learn new techniques.

The last point leads into the next key topic. We had to do quite a bit of work to create our MyMode function, and we are still not sure that it works perfectly on every variation of data it might encounter. Maybe someone else has already solved the same problem. If they did, we might be able to find an existing "package" to add onto our copy of R to extend its functions. In fact, for the statistical mode, there is an existing package that does just about everything you could imagine doing with the mode. The package is called modeest, a not very good abbreviation for mode-estimator. To install this package look in the lower right hand pane of R-studio. There are several tabs there, and one of them is "Packages." Click on this and you will get a list of every package that you already have available in your copy of R (it may be a short list) with checkmarks for the ones that are ready to use. It is unlikely that modeest is already on this list, so click on the button that says "Install Packages. This will give a dialog that looks like what you see on the screenshot above. Type the beginning of the package name in the appropriate area, and R-studio will start to prompt you with matching choices. Finish typing modeest or choose it off of the list. There may be a check box for "Install Dependencies," and if so leve this checked. In some cases an R package will depend on other packages and R will install all of the necessary packages in the correct order if it can. Once you click the "Install" button in this dialog, you will see some commands running on the R console (the lower left pane). Generally, this works without a hitch and you should not see any warning messages. Once the installation is complete you will see modeest added to the list in the lower right pane (assuming you have clicked the "Packages" tab). One last step is to click the check box next to it. This runs the library() function on the package, which prepares it for further use.

Let's try out the mfv() function. This function returns the "most frequent value" in a vector, which is generally what we want in a mode function:

```
> mfv(tinyData)

[1] 9
```

So far so good! This seems to do exactly what our MyMode() function did, though it probably uses a different method. In fact, it is easy to see what strategy the authors of this package used just by typing the name of the function at the R command line:

```
> mfv

function (x, ...)

{

    f <- factor(x)

    tf <- tabulate(f)

    return(as.numeric(levels(f)[tf == max(tf)]))

}

<environment: namespace:modeest>
```

This is one of the great things about an open source program: you can easily look under the hood to see how things work. Notice that this is quite different from how we built MyMode(), although it too uses the tabulate() function. The final line, that begins with the word "environment" has importance for more complex feats of pro-

gramming, as it indicates which variable names mfv() can refer to when it is working. The other aspect of this function which is probably not so obvious is that it will correctly return a list of multiple modes when one exists in the data you send to it:

```
> multiData <- c(1,5,7,7,9,9,10)

> mfv(multiData)

[1] 7 9

> MyMode(multiData)

[1] 7
```

In the first command line above, we made a small new vector that contains two modes, 7 and 9. Each of these numbers occurs twice, while the other numbers occur only once. When we run mfv() on this vector it correctly reports both 7 and 9 as modes. When we use our function, MyMode(), it only reports the first of the two modes.

To recap, this chapter provided a basic introduction to R-studio, an integrated development environment (IDE) for R. An IDE is useful for helping to build reusable components for handling data and conducting data analysis. From this point forward, we will use R-studio, rather than plain old R, in order to save and be able to reuse our work. Among other things, R-studio makes it easy to manage "packages" in R, and packages are the key to R's extensibility. In future chapters we will be routinely using R packages to get access to specialized capabilities.

These specialized capabilities come in the form of extra functions that are created by developers in the R community. By creating our own function, we learn that functions take "arguments" as their inputs and provide a return value. A return value is a data object, so it could be a single number (technically a vector of length one) or it could be a list of values (a vector) or even a more complex data object. We can write and reuse our own functions, which we will do quite frequently later in the book, or we can use other people's functions by installing their packages and using the library() function to make the contents of the package available. Once we have used library() we can inspect how a function works by typing its name at the R command line. (Note that this works for many functions, but there are a few that were created in a different computer language, like C, and for those we will not be able to inspect the code as easily.)

Chapter Challenge

Write and test a new function called MySamplingDistribution() that creates a sampling distribution of means from a numeric input vector. You will need to integrate your knowledge of creating new functions from this chapter with your knowledge of creating sampling distributions from the previous chapter in order to create a working function. Make sure to give careful thought about the parameters you will need to pass to your function and what kind of data object your function will return.

Sources

http://en.wikipedia.org/wiki/R_(programming_language)

http://en.wikipedia.org/wiki/Joseph_J._Allaire

http://stats.lse.ac.uk/penzer/ST419materials/CSchpt3.pdf

http://www.use-r.org/downloads/Getting_Started_with_RStudio.pdf

http://www.statmethods.net/interface/packages.html

http://www.youtube.com/watch?v=7sAmqkZ3Be8

R Commands Used in this Chapter

function() - Creates a new function

return() - Completes a function by returning a value

tabulate() - Counts occurrences of integer-valued data in a vector

unique() - Creates a list of unique values in a vector

match() - Takes two lists and returns values that are in each

mfv() - Most frequent value (from the modeest package)

Review 9.1 Onward with R-Studio

Question 1 of 5

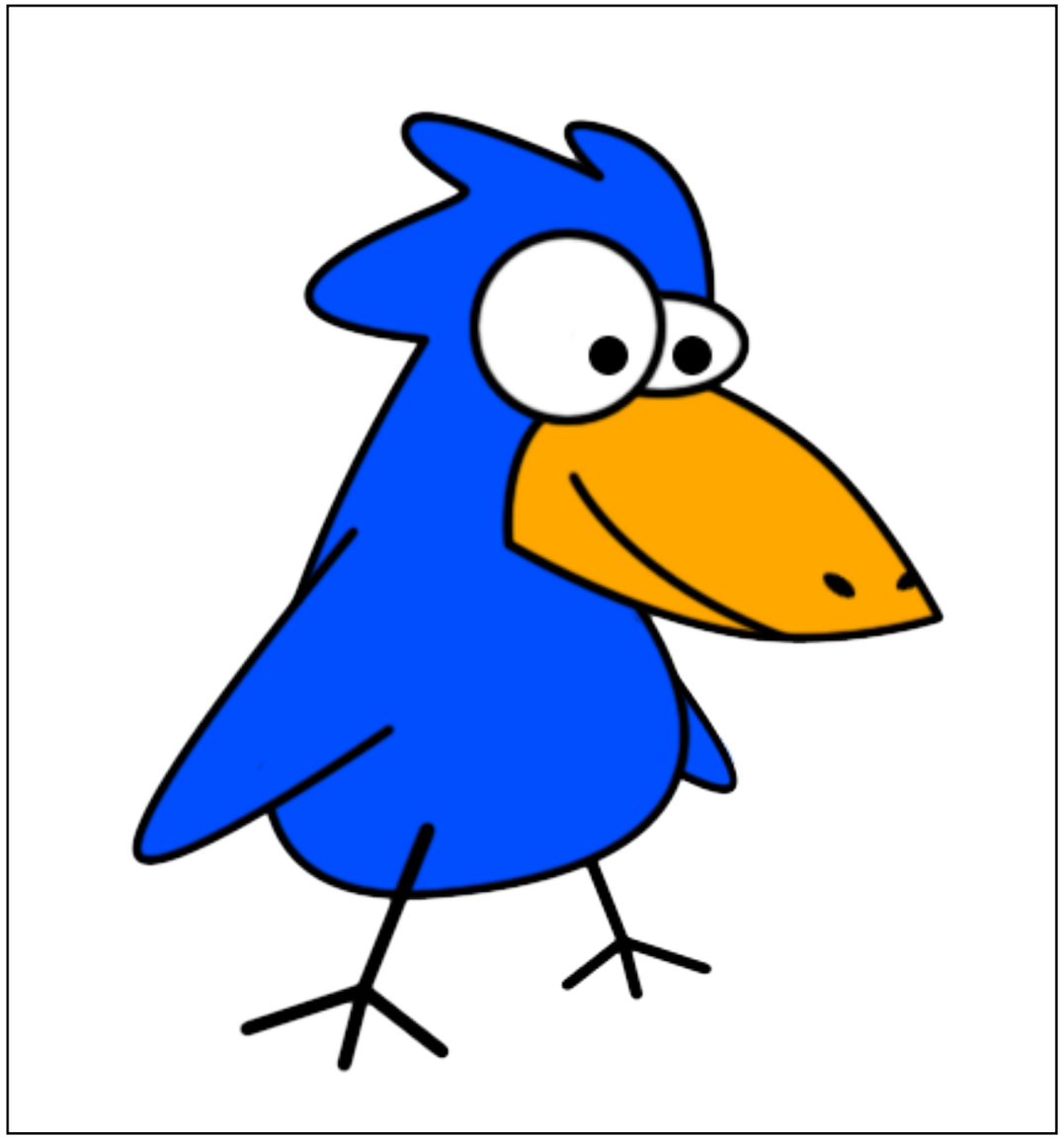
One common definition for the statistical mode is:

- **A.** The sum of all values divided by the number of values.
- ✓ **B.** The most frequently occurring value in the data.
- **C.** The halfway point through the data.
- **D.** The distance between the smallest value and the largest value.

Check Answer

CHAPTER 10

Tweet, Tweet!



We've come a long way already: Basic skills in controlling R, some exposure to R-studio, knowledge of how to manage add-on packages, experience creating a function, essential descriptive statistics, and a start on sampling distributions and inferential statistics. In this chapter, we use the social media service Twitter to grab some up-to-the minute data and begin manipulating it.

Prior to this chapter we only worked with toy data sets: some made up data about a fictional family and the census head counts for the 50 states plus the District of Columbia. At this point we have practiced a sufficient range of skills to work with some real data. There are data sets everywhere, thousands of them, many free for the taking, covering a range of interesting topics from psychology experiments to film actors. For sheer immediacy, though, you can't beat the Twitter social media service. As you may know from direct experience, Twitter is a micro-blogging service that allows people all over the world to broadcast brief thoughts (140 characters or less) that can then be read by their "followers" (other Twitter users who signed up to receive the sender's messages. The developers of Twitter, in a stroke of genius, decided to make these postings, called tweets, available to the general public through a web page on the Twitter.com site, and additional through what is known as an application programming interface or API.

Here's where the natural extensibility of R comes in. An individual named Jeff Gentry, who at this writing seems to be a data professional in the financial services industry, created an add-on package for R called twitteR (not sure how it is pronounced, but "twit-are" seems pretty close). The twitteR package provides an extremely simple interface for downloading a list of tweets directly from the Twitter service into R. Using the interface functions in twitteR, it is possible to search through Twitter to obtain a list of tweets on a specific topic. Every tweet contains the text of the posting that the author wrote as well as lots of other useful information such as the time of day when a tweet was posted. Put it all together and it makes a fun way of getting up-to-the-minute data on what people are thinking about a wide variety of topics.

The other great thing about working with twitteR is that we will use many, if not all of the skills that we have developed earlier in the book to put the interface to use.

To begin, launch your copy of R-studio. The first order of business is to create a new R-studio "project". A project in R-studio helps to keep all of the different pieces and parts of an activity together including the datasets and variables that you establish as well as the functions that you write. For professional uses of R and R-studio, it is important to have one project for each major activity: this keeps different data sets and variable names from interfering with each other. Click on the "Project" menu in R-studio and then click on "New Project." You will usually have a choice of three kinds of new projects, a brand new "clean" project, an existing directory of files that will get turned into a project folder, or a project that comes out of a version control system. (Later in the book we will look at version control, which is great for projects involving more than one person.) Choose "New Directory" to start a brand new project. You can call your project whatever you want, but because this project uses the twitteR package, you might want to just call the project "twitter". You also have a choice in the dialog box about where on your computer R-studio will create the new directory.

R-studio will respond by showing a clean console screen and most importantly an R "workspace" that does not contain any of the old variables and data that we created in previous chapters. In order to use twitteR, we need to load several packages that it depends aupon. These are called, in order "bitops", "RCurl", "RJSONIO", and once these are all in place "twitteR" itself. Rather than doing all of this by hand with the menus, lets create some functions that will assist us and make the activity more repeatable. First, here is a func-

tion that takes as input the name of a package. It tests whether the package has been downloaded - "installed" - from the R code repository. If it has not yet been downloaded/installed, the function takes care of this. Then we use a new function, called require(), to prepare the package for further use. Let's call our function "EnsurePackage" because it ensures that a package is ready for us to use. If you don't recall this step from the previous chapter, you should click the "File" menu and then click "New" to create a new file of R script. Then, type or copy/paste the following code:

```
EnsurePackage<-function(x)
{
  x <- as.character(x)
  if (!require(x,character.only=TRUE))
  {
    install.packages(pkgs=x,+
         repos="http://cran.r-project.org")
    require(x,character.only=TRUE)
  }
}
```

The require() function on the fourth line above does the same thing as library(), which we learned in the previous chapter, but it also returns the value "FALSE" if the package you requested in the argument "x" has not yet been downloaded. That same line of code also contains another new feature, the "if" statement. This is what computer scientists call a conditional. It tests the stuff inside the parentheses to see if it evaluates to TRUE or FALSE. If TRUE, the program continues to run the script in between the curly braces (lines 4 and 8). If FALSE, all the stuff in the curly braces is skipped. Also in the third line, in case you are curious, the arguments to the require() function include "x," which is the name of the package that was passed into the function, and "character.only=TRUE" which tells the require() function to expect x to be a character string. Last thing to notice about this third line: there is a "!" character that reverses the results of the logical test. Technically, it is the Boolean function NOT. It requires a bit of mental gyration that when require() returns FALSE, the "!" inverts it to TRUE, and that is when the code in the curly braces runs.

Once you have this code in a script window, make sure to select the whole function and click Run in the toolbar to make R aware of the function. There is also a checkbox on that same toolbar called, "Source on Save," that will keep us from having to click on the Run button all the time. If you click the checkmark, then every time you save the source code file, R-studio will rerun the code. If you get in the habit of saving after every code change you will always be running the latest version of your function.

Now we are ready to put EnsurePackage() to work on the packages we need for twitteR. We'll make a new function, "PrepareTwitter," that will load up all of our packages for us. Here's the code:

```
PrepareTwitter<-function()
{
  EnsurePackage("bitops")
  EnsurePackage("RCurl")
  EnsurePackage("RJSONIO")
```

```
  EnsurePackage("twitteR")
}
```

This code is quite straightforward: it calls the EnsurePackage() function we created before four times, once to load each of the packages we need. Make sure to save your script file once you have typed this new function in. You can give it any file name that make sense to you, such as "twitterSupport." Now is also a good time to start the habit of commenting: Comments are human readable messages that software developers leave for themselves and for others, so that everyone can remember what a piece of code is supposed to do. All computer languages have at least one "comment character" that sets off the human readable stuff from the rest of the code. In R, the comment character is `#`. For now, just put one comment line above each function you created, briefly describing it, like this:

EnsurePackage(x) - Installs and loads a package if necessary

and this:

PrepareTwitter() - Load packages for working with twitteR

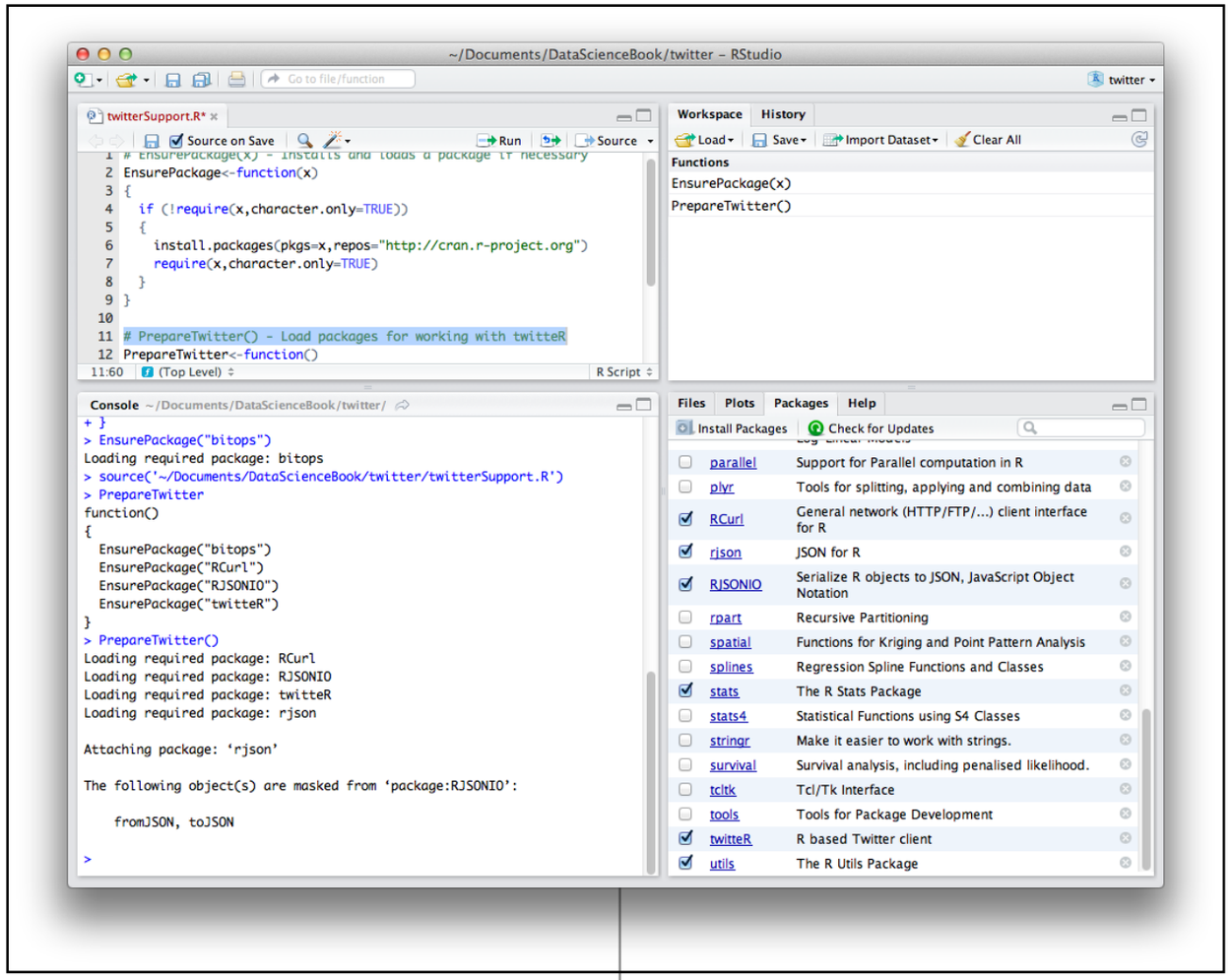
Later on we will do a better job a commenting, but this gives us the bare minimum we need to keep going with this project. Before we move on, you should run the PrepareTwitter() function on the console command line to actually load the packages we need:

```
> PrepareTwitter()
Loading required package: bitops
Loading required package: RCurl
Loading required package: RJSONIO
Loading required package: twitteR
Loading required package: rjson


Attaching package: 'rjson'


The following object(s) are masked from
'package:RJSONIO':


    fromJSON, toJSON
```

The command to run is shown in bold above. Note the parentheses after the function name, even though there is no argument to this function. What would happen if you left out the parentheses? Try it. You may get a lot more output than this, because your computer may need to download some or all of these packages. You may notice the warning message above, about objects being "masked." Generally speaking, this message refers to a variable or function that has become invisible because another variable or function with the same name has been loaded. Usually this is fine: the newer thing works the same as the older thing with the same name.

It is worth taking a look at a screen shot right now, just to make sure everything has gone smoothly. The screen shot below has four panes, each of which contains something of interest. The upper left pane is our code/script window, where we should have the code for our two new functions. The comment line at the head of the PrepareTwitter() function is highlighted. Note that the tab on this window says ""twitterSupport" which is the filename of the script file.

If yours says "untitled" you haven't saved your code yet. The lower left pane shows our R console with the results of the most recently run commands. You can see that the last command, in blue, is the PrepareTwitter() command. The output below it, in black, is what appeared on the previous page. The upper right pane contains our workspace and history of prior commands, with the tab currently set to workspace. As a reminder, in R parlance, workspace represents all of the currently available data objects include functions. Our two new functions which we have defined are listed there, indicating that they have each run at least once and R is now aware of them. In the lower right pane, we have files, plots, packages, and help, with the tab currently set to packages. This window is scrolled to the bottom to show that RCurl, RJSONIO, and twitteR are all loaded and "libraryed" meaning that they are ready to use from the command line or from functions.



Now let's get some data from Twitter. The twitteR package provides a function called searchTwitter() that allows us to retrieve some recent tweets based on a search term. Twitter users have invented a scheme for organizing their tweets based on subject matter. This system is called "hashtags" and is based on the use of the hashmark character (#) followed by a brief text tag. For example, fans of Oprah Winfrey use the tag #oprah to identify their tweets about her. We will use the searchTwitter() function to search for hashtags about global climate change. The website hashtags.org lists a variety of hashtags covering a range of contemporary topics. You can pick any hashtag you like, as long as there are a reasonable number of tweets that can be retrieved. The searchTwitter() function also requires specifying the maximum number of tweets that the call will return. For now we will use 500, although you may find that your request does not return that many. Here's the command:

```
tweetList <- searchTwitter("#climate", n=500)
```

Depending upon the speed of your Internet connection and the amount of traffic on Twitter's servers, this command may take a short while for R to process. Now we have a new data object, tweetList, that presumably contains the tweets we requested. But what is this data object? Let's use our R diagnostics to explore what we have gotten:

```
> mode(tweetList)
```

```
[1] "list"
```

Hmm, this is a type of object that we have not encountered before. In R, a list is an object that contains other data objects, and those objects may be a variety of different modes/types. Contrast this definition with a vector: A vector is also a kind of list, but with the requirement that all of the elements in the vector must be in the same mode/type. Actually, if you dig deeply into the definitions of R data objects, you may realize that we have already encountered one type of list: the dataframe. Remember that the dataframe is a list of vectors, where each vector is exactly the same length. So a dataframe is a particular kind of list, but in general lists do not have those two restrictions that dataframes have (i.e., that each element is a vector and that each vector is the same length).

So we know that tweetList is a list, but what does that list contain? Let's try using the str() function to uncover the structure of the list:

```
str(tweetList)
```

Whoa! That output scrolled right off the screen. A quick glance shows that it is pretty repetitive, with each 20 line block being quite similar. So let's use the head() function to just examine the first element of the list. The head() function allows you to just look at the first few elements of a data object. In this case we will look just at the first list element of the tweetList list. The command, also shown on the screen shot below is:

```
str(head(tweetList,1))
```

Looks pretty messy, but is simpler than it may first appear. Following the line "List of 1," there is a line that begins "$:Reference

Console ~/Documents/DataScienceBook/twitter/

```
> str(head(tweetList,1))
List of 1
 $ :Reference class 'status' [package "twitteR"] with 10 fields
  ..$ text        : chr "Get the real facts on gas prices from @ClimateDesk
http://t.co/Wx4pHcVq #climate #eco #nokxl @energyaction"
  ..$ favorited   : logi FALSE
  ..$ replyToSN   : chr(0)
  ..$ created     : POSIXct[1:1], format: "2012-04-05 14:10:01"
  ..$ truncated   : logi FALSE
  ..$ replyToSID  : chr(0)
  ..$ id          : chr "187904932609921025"
  ..$ replyToUID  : chr(0)
  ..$ statusSource: chr "&lt;a href=&quot;http://www.tweetdeck.com&quot;
rel=&quot;nofollow&quot;&gt;TweetDeck&lt;/a&gt;"
  ..$ screenName  : chr "citizensclimate"
  ..and 33 methods, of which 22 are possibly relevant:
  ..   getCreated, getFavorited, getId, getReplyToSID, getReplyToSN,
  ..   getReplyToUID, getScreenName, getStatusSource, getText,
  ..   getTruncated, initialize, setCreated, setFavorited, setId,
  ..   setReplyToSID, setReplyToSN, setReplyToUID, setScreenName,
  ..   setStatusSource, setText, setTruncated, toDataFrame
>
>
```

class" and then the word 'status' in single quotes. In Twitter terminology a "status" is a single tweet posting (it supposedly tells us the "status" of the person who posted it). So the author of the R twitteR package has created a new kind of data object, called a 'status' that itself contains 10 fields. The fields are then listed out. For each line that begins with "..$" there is a field name and then a mode or data type and then a taste of the data that that field contains.

So, for example, the first field, called "text" is of type "chr" (which means character/text data) and the field contains the string that

starts with, "Get the real facts on gas prices." You can look through the other fields and see if you can make sense of them. There are two other data types in there: "logi" stands for logical and that is the same as TRUE/FALSE; "POSIXct" is a format for storing the calendar date and time. (If you're curious, POSIX is an old unix style operating system, where the current date and time were stored as the number of seconds elapsed since 12 midnight on January 1, 1970.) You can see in the "created" field that this particular tweet was created on April 5, 2012 one second after 2:10 PM. It does not show what time zone, but a little detective work shows that all Twitter postings are coded with "coordinated universal time" or what is usually abbreviated with UTC.

One last thing to peek at in this data structure is about seven lines from the end, where it says, "and 33 methods..." In computer science lingo a "method" is an operation/activity/procedure that works on a particular data object. The idea of a method is at the heart of so called "object oriented programming." One way to think of it is that the data object is the noun, and the methods are all of the verbs that work with that noun. For example you can see the method "getCreated" in the list: If you use the method getCreated on an reference object of class 'status', the method will return the creation time of the tweet.

If you try running the command:

```
str(head(tweetList,2))
```

you will find that the second item in the tweetList list is structured exactly like the first time, with the only difference being the specific contents of the fields. You can also run:

```
length(tweetList)
```

to find out how many items are in your list. The list obtained for this exercise was a full 500 items long. Se we have 500 complex items in our list, but every item had exactly the same structure, with 10 fields in it and a bunch of other stuff too. That raises a thought: tweetList could be thought of as a 500 row structure with 10 columns! That means that we could treat it as a dataframe if we wanted to (and we do, because this makes handling these data much more convenient as you found in the "Rows and Columns" chapter).

Happily, we can get some help from R in converting this list into a dataframe. Here we will introduce four powerful new R functions: as(), lapply(), rbind(), and do.call(). The first of these, as(), performs a type coercion: in other words it changes one type to another type. The second of these, lapply(), applies a function onto all of the elements of a list. In the command below, lapply(tweetList, as.data.frame), applies the as.data.frame() coercion to each element in tweetList. Next, the rbind() function "binds" together the elements that are supplied to it into a row-by-row structure. Finally, the do.call() function executes a function call, but unlike just running the function from the console, allows for a variable number of arguments to be supplied to the function. The whole command we will use looks like this:

```
tweetDF <- do.call("rbind", lapply(tweetList, +
                        as.data.frame))
```

You might wonder a few things about this command. One thing that looks weird is "rbind" in double quotes. This is the required method of supplying the name of the function to do.call(). You might also wonder why we needed do.call() at all. Couldn't we have just called rbind() directly from the command line? You can

try it if you want, and you will find that it does provide a result, but not the one you want. The difference is in how the arguments to rbind() are supplied to it: if you call it directly, lapply() is evaluated first, and it forms a single list that is then supplied to rbind(). In contrast, by using do.call(), all 500 of the results of lapply() are supplied to rbind() as individual arguments, and this allows rbind() to create the nice rectangular dataset that we will need. The advantage of do.call() is that it will set up a function call with a variable number of arguments in cases where we don't know how many arguments will be supplied at the time when we write the code.

If you run the command above, you should see in the upper right hand pane of R-studio a new entry in the workspace under the heading of "Data." For the example we are running here, the entry says, "500 obs. of 10 variables." This is just what we wanted, a nice rectangular data set, ready to analyze. Later on, we may need more than one of these data sets, so let's create a function to accomplish the commands we just ran:

```
# TweetFrame() - Return a dataframe based on a
#                search of Twitter

TweetFrame<-function(searchTerm, maxTweets)

{

  twtList<-searchTwitter(searchTerm,n=maxTweets)

  return(do.call("rbind",+
                lapply(twtList,as.data.frame)))

}
```

There are three good things about putting this code in a function. First, because we put a comment at the top of the function, we will remember in the future what this code does. Second, if you test this function you will find out that the variable twtList that is created in the code above does not stick around after the function is finished running. This is the result of what computer scientists call "variable scoping." The variable twtList only exists while the TweetFrame() function is running. Once the function is done, twtList evaporates as if it never existed. This helps us to keep our workspace clean and avoid collecting lots of intermediate variables that are not reused.

The last and best thing about this function is that we no longer have to remember the details of the method for using do.call(), rbind(), lapply(), and as.data.frame() because we will not have to retype these commands again: we can just call the function whenever we need it. And we can always go back and look at the code later. In fact, this would be a good reason to put in a comment just above the return() function. Something like this:

```
# as.data.frame() coerces each list element into a row
# lapply() applies this to all of the elements in twtList
# rbind() takes all of the rows and puts them together
# do.call() gives rbind() all the rows as individual elements
```

Now, whenever we want to create a new data set of tweets, we can just call TweetFrame from the R console command line like this:

```
lgData <- TweetFrame("#ladygaga", 250)
```

This command would give us a new dataframe "lgData" all ready to analyze, based on the supplied search term and maximum number of tweets.

Let's start to play with the tweetDF dataset that we created before. First, as a matter of convenience, let's learn the attach() function. The attach() function saves us some typing by giving one particular dataframe priority over any others that have the same variable names. Normally, if we wanted to access the variables in our dataframe, we would have to use the $ notation, like this:

```
tweetDF$created
```

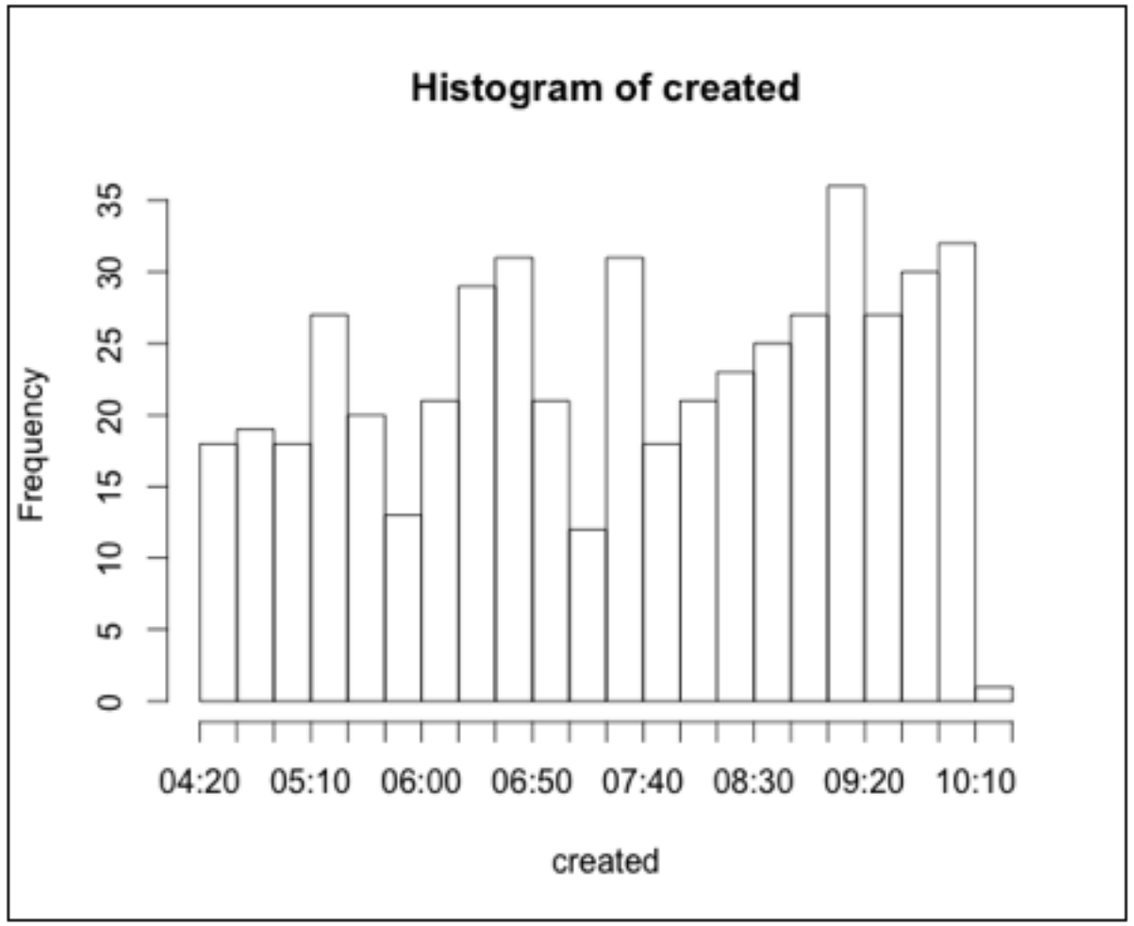
But if we run attach(tweetDF) first, we can then refer to created directly, without having to type the tweetDF$ before it:

```
> attach(tweetDF)

> head(created,4)

[1] "2012-04-05 14:10:01 UTC" "2012-04-05 14:09:21 UTC"

[3] "2012-04-05 14:08:15 UTC" "2012-04-05 14:07:12 UTC"
```

Let's visualize the creation time of the 500 tweets in our dataset. Wen working with time codes, the hist() function requires us to specify the approximate number of categories we want to see in the histogram:

```
hist(created, breaks=15, freq=TRUE)
```

This command yields the histogram that appears below. If we look along the x-axis (the horizontal), this string of tweets starts at about 4:20 AM and goes until about 10:10 AM, a span of roughly six hours. There are 22 different bars so each bar represents about 16 minutes - for casual purposes we'll call it a quarter of an hour. It looks like there are something like 20 tweets per bar, so we are looking at roughly 80 tweets per hour with the hashtag "#climate." This is obviously a pretty popular topic. This distribution does not really have a discernible shape, although it seems like there might be a bit of a growth trend as time goes on, particularly starting at about 7:40 AM.



Take note of something very important about these data: It doesn't make much sense to work with a measure of central tendency. Remember a couple of chapters ago when we were looking at the number of people who resided in different U.S. states? In that case it made sense to say that if State A had one million people and State B had three million people, then the average of these two states was two million people. When you're working with time

stamps, it doesn't make a whole lot of sense to say that one tweet arrived at 7 AM and another arrived at 9 AM so the average is 8 AM. Fortunately, there's a whole area of statistics concerned with "arrival" times and similar phenomena, dating back to a famous study by Ladislaus von Bortkiewicz of horsemen who died after being kicked by their horses. von Bortkiewicz studied each of 14 cavalry corps over a period of 20 years, noting when horsemen died each year. The distribution of the "arrival" of kick-deaths turns out to have many similarities to other arrival time data, such as the arrival of buses or subway cars at a station, the arrival of customers at a cash register, or the occurrence of telephone calls at a particular exchange. All of these kinds of events fit what is known as a "Poisson Distribution" (named after Simeon Denis Poisson, who published it about half a century before von Bortkiewicz found a use for it). Let's find out if the arrival times of tweets comprise a Poisson distribution.

Right now we have the actual times when the tweets were posted, coded as a POSIX date and time variable. Another way to think about these data is to think of each new tweet as arriving a certain amount of time after the previous tweet. To figure that out, we're going to have to "look back" a row in order to subtract the creation time of the previous tweet from the creation time of the current tweet. In order to be able to make this calculation, we have to make sure that our data are sorted in ascending order of arrival - in other words the earliest one first and the latest one last. To accomplish this, we will use the order() function together with R's built-in square bracket notation.

As mentioned briefly in the previous chapter, in R, square brackets allow "indexing" into a list, vector, or data frame. For example, myList[3] would give us the third element of myList. Keeping in mind that a a dataframe is a rectangular structure, really a two dimensional structure, we can address any element of a dataframe with both a row and column designator: myFrame[4,1] would give the fourth row and the first column. A shorthand for taking the whole column of a dataframe is to leave the row index empty: myFrame[, 6] would give every row in the sixth column. Likewise, a shorthand for taking a whole row of a dataframe is to leave the column index empty: myFrame[10,] would give every column in the tenth row. We can also supply a list of rows instead of just one row, like this: myFrame[c(1,3,5),] would return rows 1, 3, 5 (including the data for all columns, because we left the column index blank). We can use this feature to reorder the rows, using the order() function. We tell order() which variable we want to sort on, and it will give back a list of row indices in the order we requested. Putting it all together yields this command:

```
tweetDF[order(as.integer(created)), ]
```

Working our way from the inside to the outside of the expression above, we want to sort in the order that the tweets were created. We first coerce the variable "created" to integer - it will then truly be expressed in the number of seconds since 1970 - just in case there are operating system differences in how POSIX dates are sorted. We wrap this inside the order() function. The order() function will provide a list of row indices that reflects the time ordering we want. We use the square brackets notation to address the rows in tweetDF, taking all of the columns by leaving the index after the comma empty.

We have a choice of what to do with the dataframe that is returned from this command. We could assign it back to tweetDF, which

would overwrite our original dataframe with the sorted version. Or we could create a new sorted dataframe and leave the original data alone, like so:

```
sortweetDF<-tweetDF[order(as.integer(created)), ]
```

If you choose this method, make sure to detach() tweetDF and attach() sortweetDF so that later commands will work smoothly with the sorted dataframe:

```
> detach(tweetDF)

> attach(sortweetDF)
```

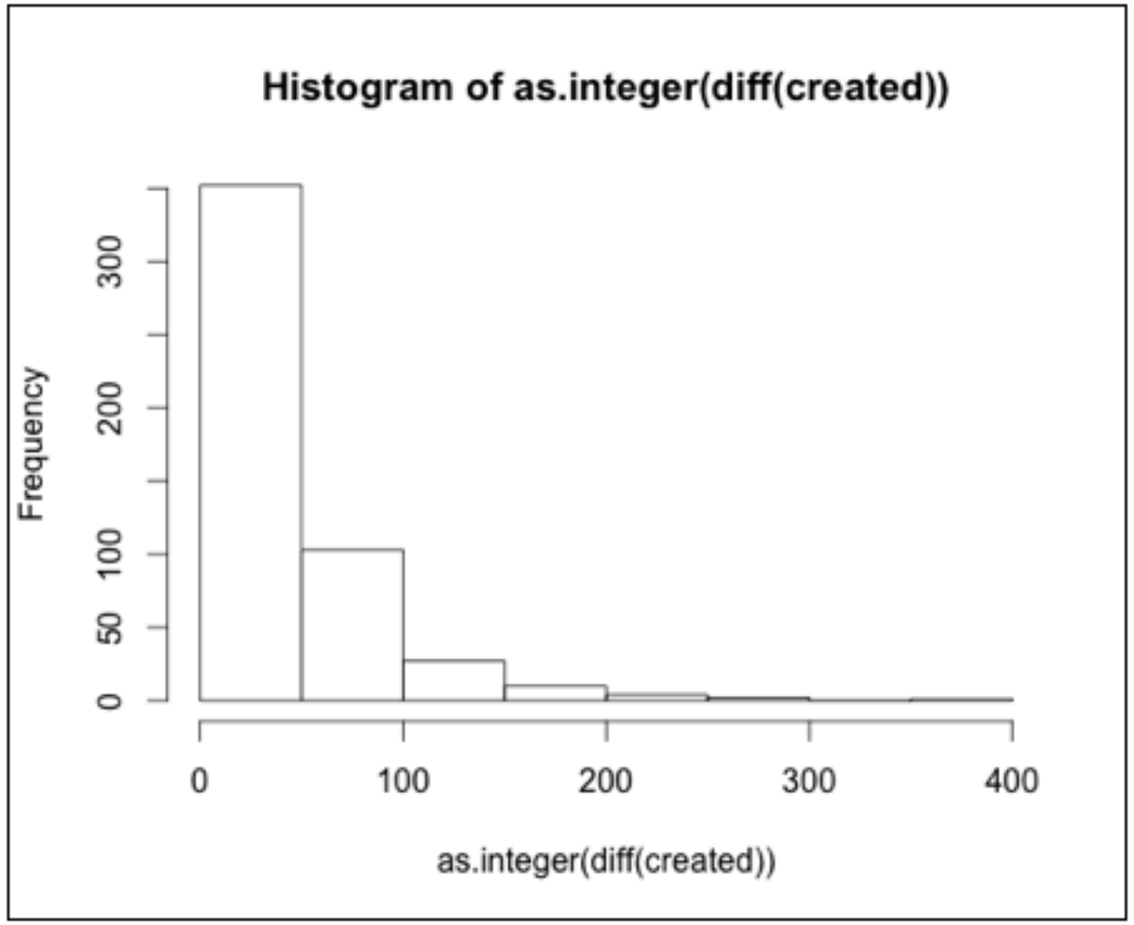
Another option, which seems better than creating a new dataframe, would be to build the sorting into the TweetFrame() function that we developed at the beginning of the chapter. Let's leave that to the chapter challenge. For now, we can keep working with sortweetDF.

Technically, what we have with our created variable now is a time series, and because statisticians like to have convenient methods for dealing with time series, R has a built-in function, called diff(), that allows us to easily calculate the difference in seconds between each pair of neighboring values. Try it:

```
diff(created)
```

You should get a list of time differences, in seconds, between neighboring tweets. The list will show quite a wide range of intervals, perhaps as long as several minutes, but with many intervals near or at zero. You might notice that there are only 499 values and not 500: This is because you cannot calculate a time difference for the very first tweet, because we have no data on the prior tweet. Let's visualize these data and see what we've got:

```
hist(as.integer(diff(created))
```



As with earlier commands, we use as.integer() to coerce the time differences into plain numbers, otherwise hist() does not know how to handle the time differences. This histogram shows that the majority of tweets in this group come within 50 seconds or less of the previous tweets. A much smaller number of tweets arrive within somewhere between 50 and 100 seconds, and so on down the line. This is typical of a Poisson arrival time distribution. Un-

like the raw arrival time data, we could calculate a mean on the time differences:

```
> mean(as.integer(diff(created)))
[1] 41.12826
```

We have to be careful though, in using measures of central tendency on this positively skewed distribution, that the value we get from the mean() is a sensible representation of central tendency. Remembering back to the previous chapter, and our discussion of the statistical mode (the most frequently occurring value), we learn that the mean and the mode are very different:

```
> library("modeest")
> mfv(as.integer(diff(created)))
[1] 0
```

We use the library() function to make sure that the add on package with the mfv() function is ready to use. The results of the mfv() function show that the most commonly occurring time interval between neighboring tweets is zero!

Likewise the median shows that half of the tweets have arrival times of under half a minute:

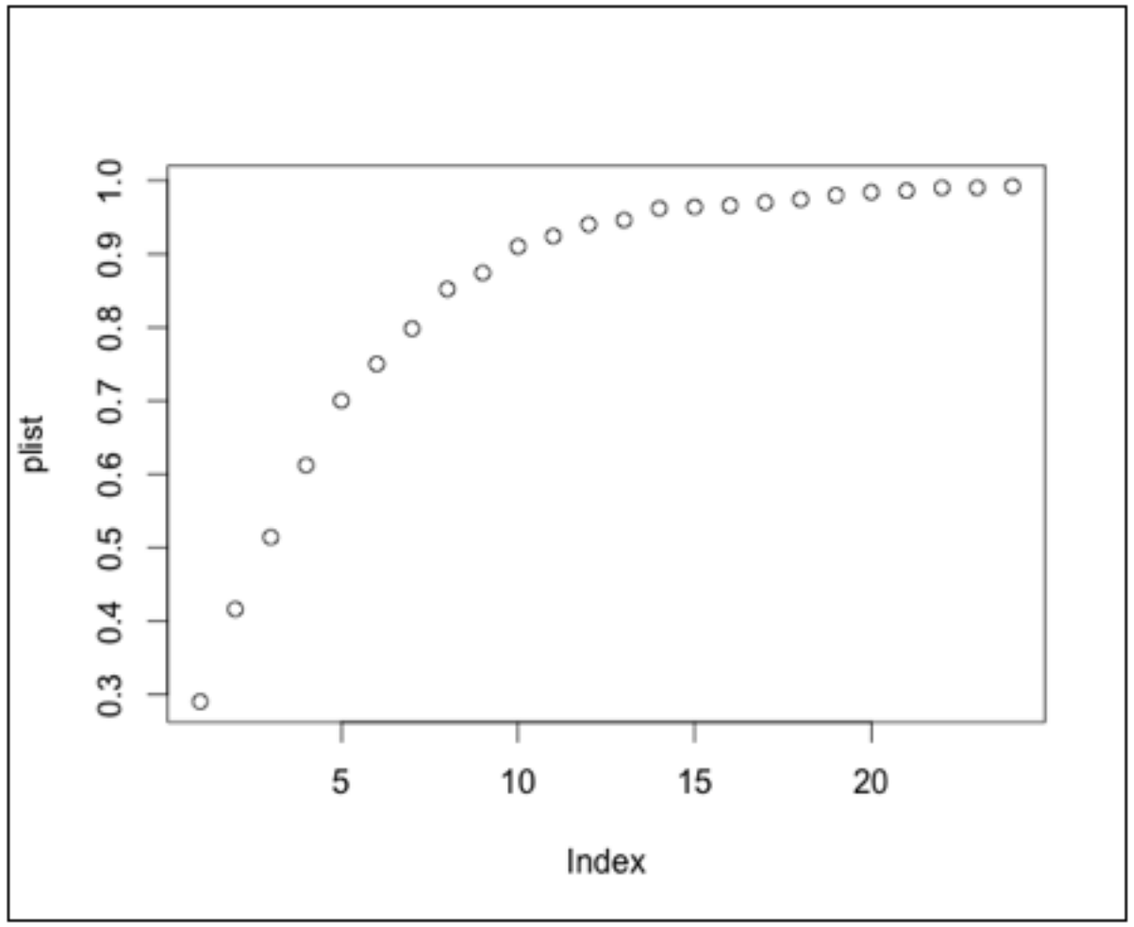
```
> median(as.integer(diff(created)))
[1] 28
```

In the next chapter we will delve more deeply into what it means when a set of data are shaped like a Poisson distribution and what that implies about making use of the mean.

One last way of looking at these data before we close this chapter. If we choose a time interval, such as 10 seconds, or 30 seconds, or 60 seconds, we can ask the question of how many of our tweet arrivals occurred within that time interval. Here's code that counts the number of arrivals that occur within certain time intervals:

```
> sum((as.integer(diff(created)))<60)
[1] 375
> sum((as.integer(diff(created)))<30)
[1] 257
> sum((as.integer(diff(created)))<10)
[1] 145
```

You could also think of these as ratios, for example 145/500 = 0.29. And where we have a ratio, we often can think about it as a probability: There is a 29% probability that the next tweet will arrive in 10 seconds or less. You could make a function to create a whole list of these probabilities. Some sample code for such a function appears at the end of the chapter. Some new scripting skills that we have not yet covered (for example, the "for loop") appear in this function, but try making sense out of it to stretch your brain. Output from this function created the plot that appears below.



This is a classic Poisson distribution of arrival probabilities. The x-axis contains 10 second intervals (so by the time you see the number 5 on the x-axis, we are already up to 50 seconds). This is called a cumulative probability plot and you read it by talking about the probability that the next tweet will arrive in the amount of time indicated on the x-axis or less. For example, the number five on the x-axis corresponds to about a 60% probability on the y-axis, so there is a 60% probability that the next tweet will arrive in 50 seconds or less. Remember that this estimate applies only to the data in this sample!

In the next chapter we will reexamine sampling in the context of Poisson and learn how to compare two Poisson distributions to find out which hashtag is more popular.

Let's recap what we learned from this chapter. First, we have begun to use the project features of R-studio to establish a clean environment for each R project that we build. Second, we used the source code window of R-studio to build two or three very useful functions, ones that we will reuse in future chapters. Third, we practiced the skill of installing packages to extend the capabilities of R. Specifically, we loaded Jeff Gentry's twitteR package and the other three packages it depends upon. Fourth, we put the twitteR package to work to obtain our own fresh data right from the web. Fifth, we started to condition that data, for example by creating a sorted list of tweet arrival times. And finally, we started to analyze and visualize those data, by conjecturing that this sample of arrival times fitted the classic Poisson distribution.

Chapter Challenge

Modify the TweetFrame() function created at the beginning of this chapter to sort the dataframe based on the creation time of the tweets. This will require taking the line of code from a few pages ago that has the order() function in it and adding this to the TweetFrame() function with a few minor modifications. Here's a hint: Create a temporary dataframe inside the function and don't attach it while you're working with it. You'll need to use the $ notation to access the variable you want to use to order the rows.

Sources

http://cran.r-project.org/web/packages/twitteR/twitteR.pdf

http://en.wikipedia.org/wiki/Ladislaus_Bortkiewicz

http://en.wikipedia.org/wiki/Poisson_distribution

http://hashtags.org/

http://www.khanacademy.org/math/probability/v/poisson-process-1

http://www.khanacademy.org/math/probability/v/poisson-process-2

https://support.twitter.com/articles/49309 (hashtags explained)

http://www.rdatamining.com/examples/text-mining

R Script - Create Vector of Probabilities From Arrival Times

```
# ArrivalProbability - Given a list of arrival times
# calculates the delays between them using lagged differences
# then computes a list of cumulative probabilities of arrival
# for the sequential list of time increments
# times - A sorted, ascending list of arrival times in POSIXct
# increment - the time increment for each new slot, e.g. 10 sec
# max - the highest time increment, e.g., 240 sec
#
# Returns - an ordered list of probabilities in a numeric vector
# suitable for plotting with plot()
ArrivalProbability<-function(times, increment, max)
{
  # Initialize an empty vector
  plist <- NULL

  # Probability is defined over the size of this sample
  # of arrival times
  timeLen <- length(times)

  # May not be necessary, but checks for input mistake
  if (increment>max) {return(NULL)}

  for (i in seq(increment, max, by=increment))
  {
    # diff() requires a sorted list of times
```

```
    # diff() calculates the delays between neighboring times
    # the logical test <i provides a list of TRUEs and FALSEs
    # of length = timeLen, then sum() counts the TRUEs.
    # Divide by timeLen to calculate a proportion
    plist<-c(plist,(sum(as.integer(diff(times))<i))/timeLen)
  }
  return(plist)
}
```

R Functions Used in This Chapter

attach() - Makes the variables of a dataset available without $

as.integer() - Coerces data into integers

detach() - Undoes an attach function

diff() - Calculates differences between neighboring rows

do.call() - Calls a function with a variable number of arguments

function() - Defines a function for later use

hist() - Plots a histogram from a list of data

install.packages() - Downloads and prepares a package for use

lapply() - Applies a function to a list

library() - Loads a package for use; like require()

mean() - Calculates the arithmetic mean of a vector

median() - Finds the statistical center point of a list of numbers

mfv() - Most frequent value; part of the modeest() package

mode() - Shows the basic data type of an object

order() - Returns a sorted list of index numbers

rbind() - Binds rows into a dataframe object

require() - Tests if a package is loaded and loads it if needed

searchTwitter() - Part of the twitteR package

str() - Describes the structure of a data object

sum() - Adds up a list of numbers

CHAPTER 11

Popularity Contest

In the previous chapter we found that arrival times of tweets on a given topic seem to fit a Poisson distribution. Armed with that knowledge we can now develop a test to compare two different Twitter topics to see which one is more popular (or at least which one has a higher posting rate). We will use our knowledge of sampling distributions to understand the logic of the test.

Which topic on Twitter is more popular, Lady Gaga or Oprah Winfrey? This may not seem like an important question, depending upon your view of popular culture, but if we can make the comparison for these two topics, we can make it for any two topics. Certainly in the case of presidential elections, or a corruption scandal in the local news, or an international crisis, it could be a worthwhile goal to be able to analyze social media in a systematic way. And on the surface, the answer to the question seems trivial: Just add up who has more tweets. Surprisingly, in order to answer the question in an accurate and reliable way, this won't work, at least not very well. Instead, one must consider many of the vexing questions that made inferential statistics necessary.

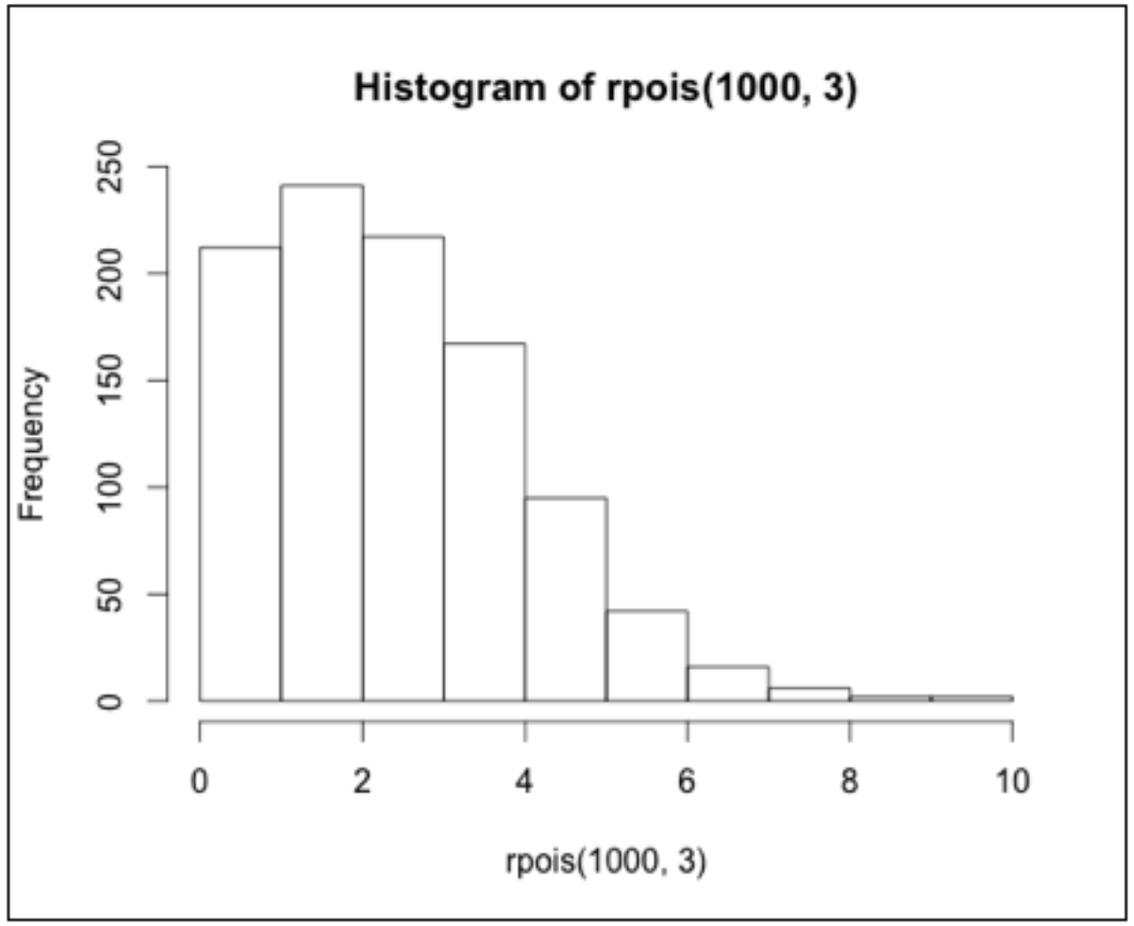
Let's say we retrieved one hour's worth of Lady Gaga tweets and a similar amount of Oprah Winfrey tweets and just counted them up. What if it just happened to be a slow news day for Oprah? It really wouldn't be a fair comparison. What if most of Lady Gaga's tweets happen at midnight or on Saturdays? We could expand our sampling time, maybe to a day or a week. This could certainly help: Generally speaking, the bigger the sample, the more representative it is of the whole population, assuming it is not collected in a biased way. This approach defines popularity as the number of tweets over a fixed period of time. Its success depends upon the choice of a sufficiently large period of time, that the tweets are collected for the two topics at the same time, and that the span of time chosen happens to be equally favorable for both two topics.

Another approach to the popularity comparison would build upon what we learned in the previous chapter about how arrival times (and the delays between them) fit into the Poisson distribution. In this alternative definition of the popularity of a topic, we could suggest that if the arrival curve is "steeper" for the first topic in contrast to the second topic, then the first topic is more active and therefore more popular. Another way of saying the same thing is that for the more popular topic, the likely delay until the arrival of the next tweet is shorter than for the less popular topic. You could also say that for a given interval of time, say ten minutes, the number of arrivals for the first topic would be higher than for the second topic. Assuming that the arrival delays fit a Poisson distribution, these are all equivalent ways of capturing the comparison between the two topics.

Just as we did in the chapter entitled, "Sample in a Jar," we can use a random number generator in R to illustrate these kinds of differences more concretely. The relevant function for the Poisson distribution is rpois(), "random poisson." The rpois() function will generate a stream of random numbers that roughly fit the Poisson distribution. The fit gets better as you ask for a larger and larger sample. The first argument to rpois() is how many random numbers you want to generate and the second number is the average delay between arrivals that you want the random number generator to try to come close to. We can look at a few of these numbers and then use a histogram function to visualize the results:

```
> rpois(10,3)

 [1] 5 4 4 2 0 3 6 2 3 3

> mean(rpois(100,3))

[1] 2.99

> var(rpois(100,3))

[1] 3.028182
```

```
> hist(rpois(1000,3))
```



In the first command above, we generate a small sample of n=10 arrival delays, with a hoped for mean of 3 seconds of delay, just to see what kind of numbers we get. You can see that all of the numbers are small integers, ranging from 0 to 6. In the second command we double check these results with a slightly larger sample of n=100 to see if rpois() will hit the mean we asked for. In that run it came out to 2.99, which was pretty darned close. If you run this command yourself you will find that your result will vary a bit each time: it will sometimes be slightly larger than three and occasionally a little less than three (or whatever mean you specify).

This is normal, because of the random number generator. In the third command we run yet another sample of 100 random data points, this time analyzing them with the var() function (which calculates the variance; see the chapter entitled "Beer, Farms, and Peas"). It is a curious fact of Poission distributions that the mean and the variance of the "ideal" (i.e., the theoretical) distribution are the same. In practice, for a small sample, they may be different.

In the final command, we ask for a histogram of an even larger sample of n=1000. The histogram shows the most common value hanging right around three seconds of delay with a nice tail that points rightwards and out to about 10 seconds of delay. You can think of this as one possible example of what you might observe of the average delay time between tweets was about three seconds. Note how similar the shape of this histogram is to what we observed with real tweets in the last chapter.

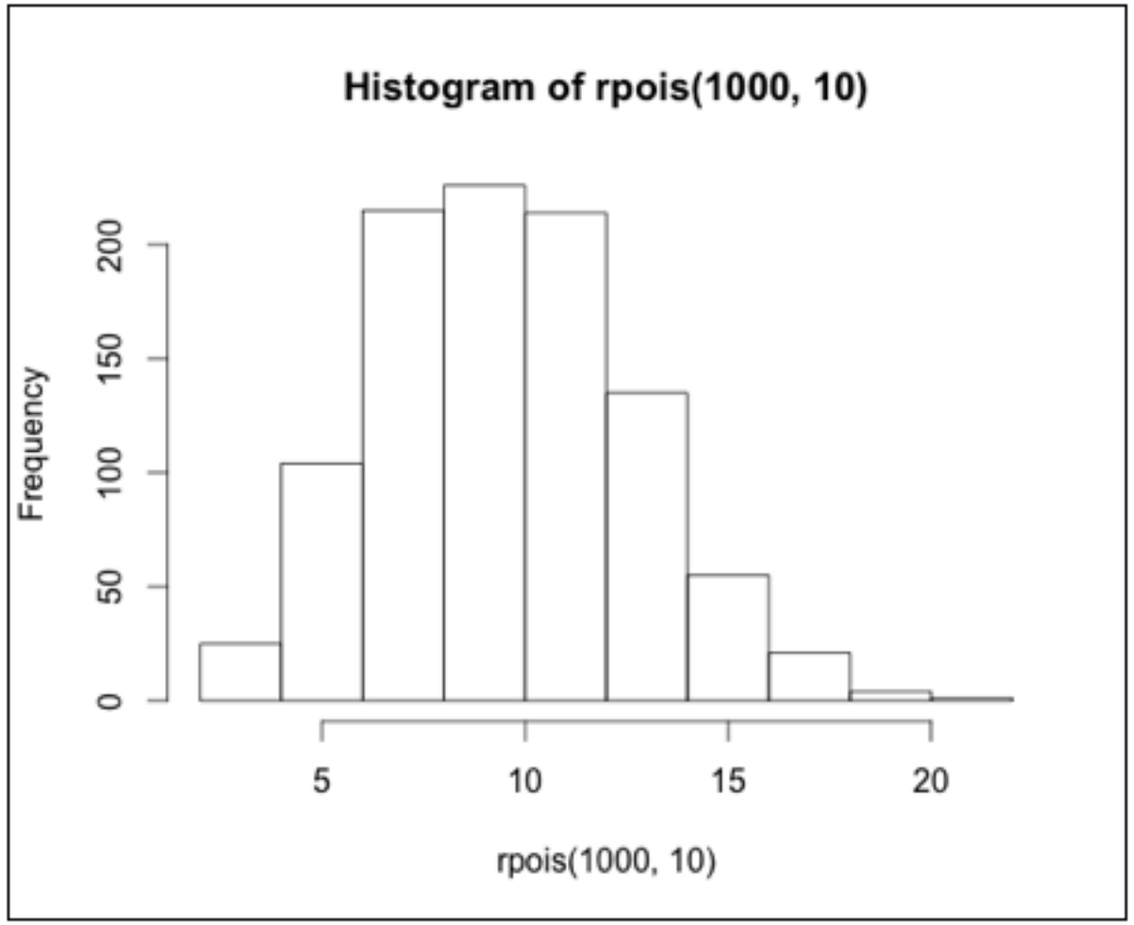
Compare the histogram on the previous page to the one on the next page that was generated with this command:

```
hist(rpois(1000,10))
```

It is pretty easy to see the different shape and position of this histogram, which has a mean arrival delay of about ten seconds. First of all, there are not nearly as many zero length delays. Secondly, the most frequent value is now about 10 (as opposed to two in the previous histogram). Finally, the longest delay is now over 20 seconds (instead of 10 for the previous histogram). One other thing to try is this:

```
> sum(rpois(1000,10)<=10)

[1] 597
```

This command generated 1000 new random numbers, following the Poisson distribution and also with a hoped-for mean of 10, just like in the histogram on the next page. Using the "<=" inequality test and the sum() function, we then counted up how many events were less than or equal to 12, and this turned out to be 597 events. As a fraction of the total of n=1000 data points that rpois() generated, that is 0.597, or 59.7%.



Review 11.1 Popularity Contest (Mid-Chapter Review)

Question 1 of 4

The Poisson distribution has a characteristic shape that would be described as:

- **A.** Negatively (left) skewed
- ✓ **B.** Positively (right) skewed
- **C.** Symmetric (not skewed)
- **D.** None of the above

Check Answer

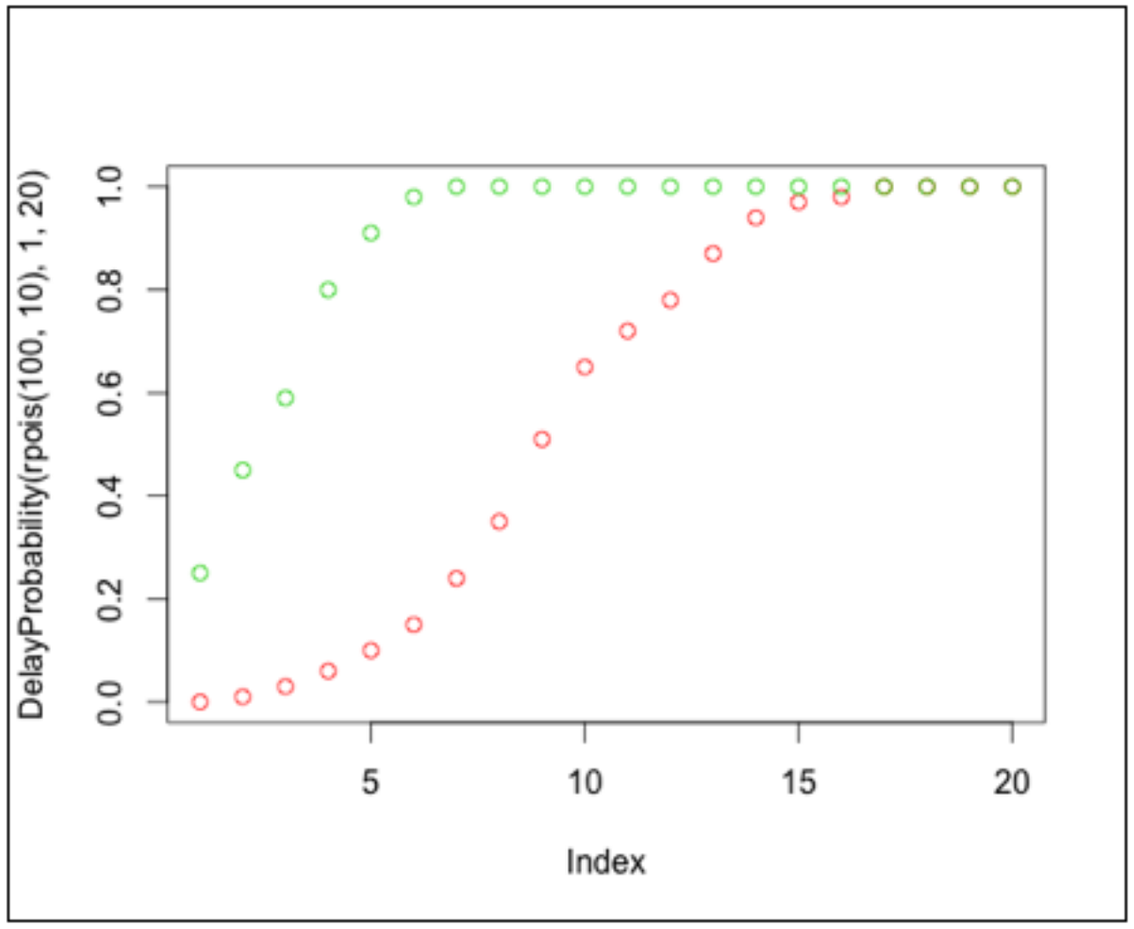
We can look at the same kind of data in terms of the probability of arrival within a certain amount of time. Because rpois() generates delay times directly (rather than us having to calculate them from neighboring arrival times), we will need a slightly different function than the ArrivalProbabilities() that we wrote and used in the previous chapter. We'll call this function "DelayProbability" (the code is at the end of this chapter):

```
> DelayProbability(rpois(100,10),1,20)

 [1] 0.00 0.00 0.00 0.03 0.06 0.09 0.21 0.33 0.48
0.61 0.73 0.82 0.92

[14] 0.96 0.97 0.98 0.99 1.00 1.00 1.00
```

At the heart of that command is the rpois() function, requesting 100 points with a mean of 10. The other two parameters are the increment, in this case one second, and the maximum delay time, in this case 20 seconds. The output from this function is a sorted list of cumulative probabilities for the times ranging from 1 second to 20 seconds. Of course, what we would really like to do is compare these probabilities to those we would get if the average delay was three seconds instead of ten seconds. We're going to use two cool tricks for creating this next plot. First, we will use the points() command to add points to an existing plot. Second, we will use the col= parameter to specify two different colors for the points that we plot. Here's the code that creates a plot and then adds more points to it:

```
> plot(DelayProbability(rpois(100,10),1,20), col=2)

> points(DelayProbability(rpois(100,3),1,20), col=3)
```

Again, the heart of each of these lines of code is the rpois() function that is generating random Poisson arrival delays for us. Our parameters for increment (1 second) and maximum (20 seconds) are the same for both lines. The first line uses col=2, which gives us red points, and the second gives us col=3, which yields green points:



This plot clearly shows that the green points have a "steeper" profile. We are more likely to have earlier arrivals for the 3-second delay data than we are for the 10-second data. If these were real tweets, the green tweets would be piling in much faster than the red tweets. Here's a reminder on how to read this plot: Look at a value on the X-axis, for example "5." Then look where the dot is

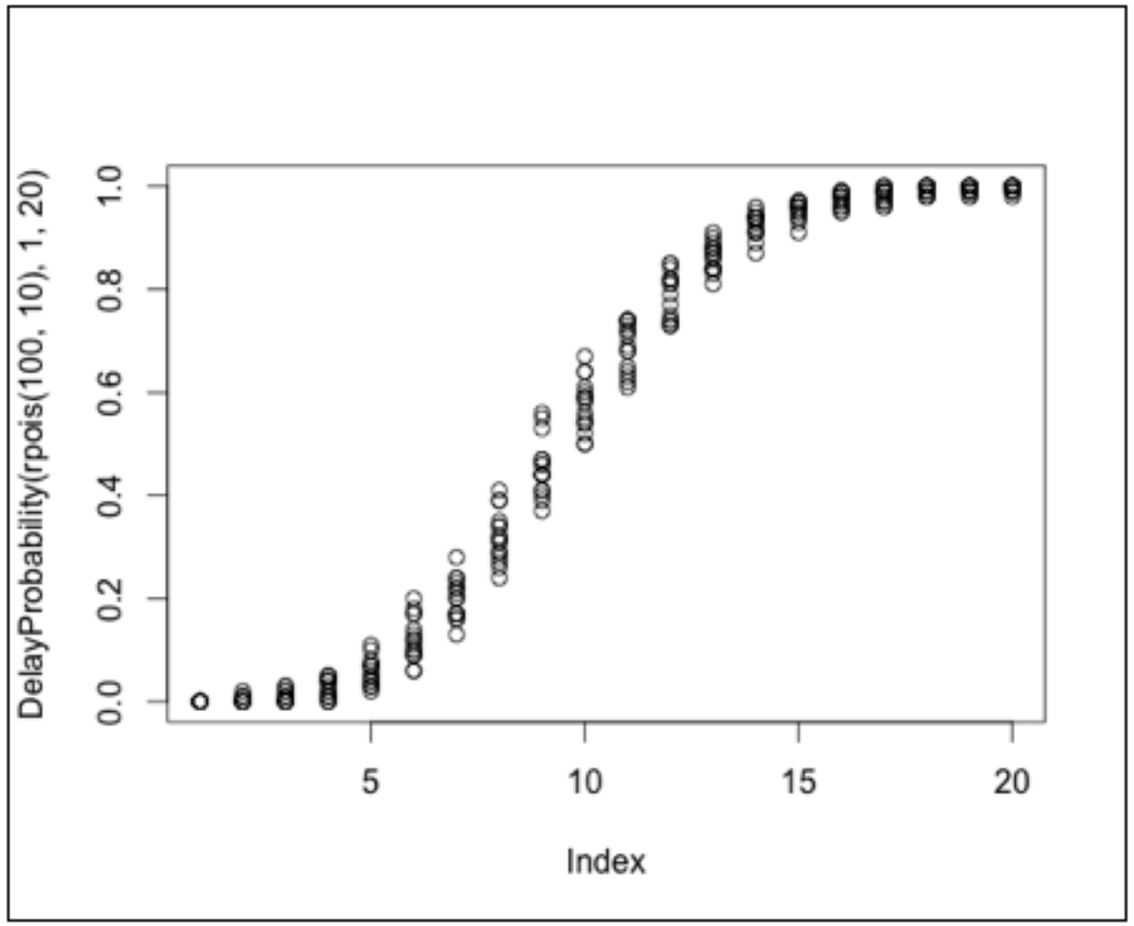
and trace leftward to the Y-axis. For the red dot, the probability value at time (x) equal 4 is about 0.10. So for the red data there is about a 10% chance that the next event will occur within five time units (we've been calling them seconds, but they could really be anything, as long as you use the units consistently throughout the whole example). For the green data there is about a 85% chance that the next event will occur within four time units. The fact that the green curve rises more steeply than the red curve means that *for these two samples only* the green stuff is arriving *much more often* than the red stuff.

These reason we emphasized the point "for these samples only" is that we know from prior chapters that every sample of data you collect varies by at least a little bit and sometimes by quite a lot. A sample is just a snapshot, after all, and things can and do change from sample to sample. We can illustrate this by running and plotting multiple samples, much as we did in the earlier chapter:

```
> plot(DelayProbability(rpois(100,10),1,20))

> for (i in 1:15) {points(DelayProbability(r-
pois(100,10),1,20))}
```

This is the first time we have used the "for loop" in R, so let's walk through it. A "for loop" is one of the basic constructions that computer scientists use to "iterate" or repeatedly run a chunk of code. In R, a for loop runs the code that is between the curly braces a certain number of times. The number of times R runs the code depends on the expression inside the parentheses that immediately follow the "for."

In the example above, the expression "i in 1:15" creates a new data object, called i, and then puts the number 1 in it. Then, the for loop keeps adding one to the value of i, until i reaches 15. Each time that it does this, it runs the code between the curly braces. The expression "in 1:15" tells R to start with one and count up to 15. The data object i, which is just a plain old integer, could also have been used within the curly braces if we had needed it, but it doesn't have to be used within the curly braces if it is not needed. In this case we didn't need it. The code inside the curly braces just runs a new random sample of 100 Poisson points with a hoped for mean of 10.



When you consider the two command lines on the previous page together you can see that we initiate a plot() on the first line of

code, using similar parameters to before (random poisson numbers with a mean of 10, fed into our probability calculator, which goes in increments of 1 second up to 20 seconds). In the second line we add more points to the same plot, by running exactly 15 additional copies of the same code. Using rpois() ensures that we have new random numbers each time:

Now instead of just one smooth curve we have a bunch of curves, and that these curves vary quite a lot. In fact, if we take the example of 10 seconds (on the X-axis), we can see that in one case the probability of a new event in 10 seconds could be as low as 0.50, while in another case the probability is as high as about 0.70.

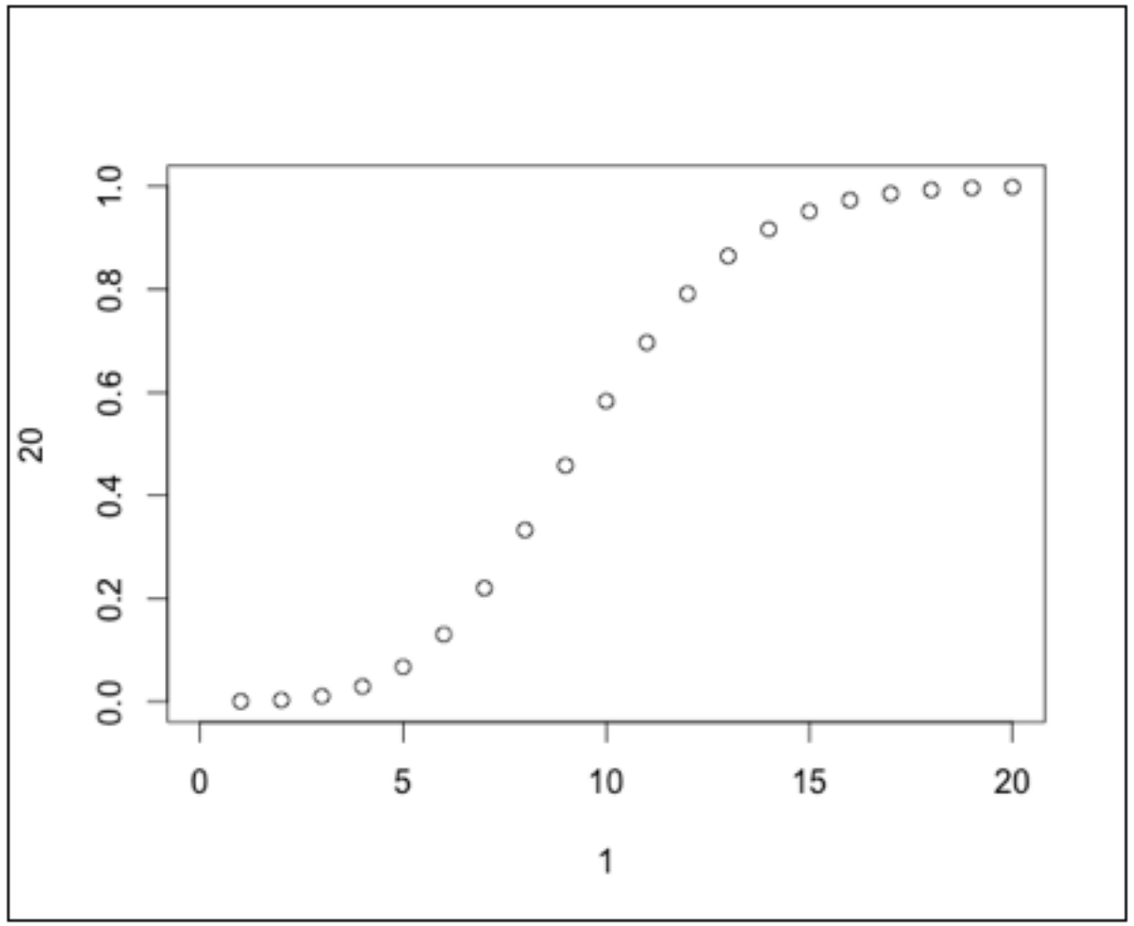
This shows why we can't just rely on one sample for making our judgments. We need to know something about the uncertainty that surrounds a given sample. Fortunately, R gives us additional tools to help us figure this situation out. First of all, even though we had loads of fun programming the DelayProbability() function, there is a quicker way to get information about what we ideally *expect* from a Poisson distribution. The function ppois() gives us the *theoretical* probability of observing a certain delay time, given a particular mean. For example:

```
> ppois(3, lambda=10)

[1] 0.01033605
```

So you can read this as: There is a 1% chance of observing a delay of 3 or less in a Poisson distribution with mean equal to 10. Note that in statistical terminology, "lambda" is the term used for the mean of a Poisson distribution. We've provided the named parameter "lambda=10" in the example above just to make sure that R does not get confused about what parameter we are controlling when we say "10." The ppois() function does have other parameters that we have not used here. Now, using a for loop, we could get a list of several of these theoretical probabilities:

```
> plot(1,20,xlim=c(0,20),ylim=c(0,1))

> for (i in 1:20) {points(i,ppois(i,lambda=10)) }
```

We are using a little code trick in the first command line above by creating a nearly empty set of axes with the plot() function, and then filling in the points in the second line using the points() function. This gives the following plot:



You may notice that this plot looks a lot like the ones earlier in this

chapter as well as somewhat similar to the probability plot in the previous chapter. When we say the "theoretical distribution" we are talking about the ideal Poisson distribution that would be generated by the complex equation that Mr. Poisson invented a couple of centuries ago. Another way to think about it is, instead just having a small sample of points, which we know has a lot of randomness in it, what if we had a truly humongous sample with zillions of data points? The curve in the plot above is just about what we would observe for a truly humongous sample (where most of the biases up or down cancel themselves out because the large number of points).

So this is the ideal, based on the mathematical theory of the Poisson distribution, or what we would be likely to observe if we created a really large sample. We know that real samples, of reasonable amounts of data, like 100 points or 1000 points or even 10,000 points, will not hit the ideal exactly, because some samples will come out a little higher and others a little lower.

We also know, from the histograms and output earlier in the chapter, that we can look at the mean of a sample, or the count of events less than or equal to the mean, or the arrival probabilities in the graph on this page, and in *each case we are looking at different versions of the same information*. Check out these five commands:

```
> mean(rpois(100000,10))
[1] 10.01009
> var(rpois(100000,10))
[1] 10.02214
> sum(rpois(100000,10)<=10)/100000
[1] 0.58638
> ppois(10,lambda=10)
[1] 0.58303
> qpois(0.58303,lambda=10)
[1] 10
```

In the first command, we confirm that for a very large random sample of n=100,000 with a desired mean of 10, the actual mean of the random sample is almost exactly 10. Likewise, for another large random sample with a desired mean of 10, the variance is 10. In the next command, we use the inequality test and the sum() function again to learn that the probability of observing a value of 10 or less in a very large sample is about 0.59 (note that the sum() function yielded 58,638 and we divided by 100,000 to get the reported value of 0.58638). Likewise, when we ask for the theoretical distribution with ppois() of observing 10 or less in a sample with a mean of 10, we get a probability of 0.58303, which is darned close to the empirical result from the previous command. Finally, if we ask qpois() what is the *threshold value* for a probability of 0.58303 is in a Poisson sample with mean of 10, we get back the answer: 10. You may see that qpois() does the reverse of what ppois() does. For fun, try this formula on the R command line:

```
qpois(ppois(10, lambda=10), lambda=10)
```

Here's one last point to cap off this thinking. Even with a sample of 100,000 there is some variation in samples. That's why the 0.58638 from the sum() function above does not exactly match the theoretical 0.58303 from the ppois() function above. We can ask R to tell us how much variation there is around one of these probabilities using the poisson.test() function like this:

```
> poisson.test(58638, 100000)

95 percent confidence interval:

 0.5816434 0.5911456
```

We've truncated a little of the output in the interests of space: What you have left is the upper and lower bounds on a 95% confidence interval. Here's what a confidence interval is: For 95% of the samples that we could generate using rpois(), using a sample size of 100,000, and a desired mean of 10, we will get a result that lies between 0.5816434 and 0.5911456 (remember that this resulting proportion is calculated as the total number of events whose delay time is 10 or less). So we know what would happen for 95% of the rpois() samples, but the assumption that statisticians also make is that *if* a natural phenomenon, like the arrival time of tweets, also fits the Poisson distribution, that this same confidence interval would be operative. So while we know that we got 0.58638 in one sample on the previous page, it is likely that future samples will vary by a little bit (about 1%). Just to get a feel for what happens to the confidence interval with smaller samples, look at these:

```
> poisson.test(5863, 10000)

95 percent confidence interval:

 0.5713874 0.6015033

> poisson.test(586, 1000)

95 percent confidence interval:

 0.5395084 0.6354261

> poisson.test(58, 100)

95 percent confidence interval:

 0.4404183 0.7497845
```

We've bolded the parameters that changed in each of the three commands above, just to emphasize that in each case we've reduced the sample size by a factor of 10. By the time we get to the bottom look how wide the confidence interval gets. With a sample of 100 events, of which 58 had delays of 10 seconds or less, the confidence interval around the proportion of 0.58 ranges from a low of 0.44 to a high of 0.75! That's huge! The confidence interval gets wider and wider as we get *less and less confident about the accuracy of our estimate.* In the case of a small sample of 100 events, the confidence interval is very wide, showing that we have a lot of uncertainty about our estimate that 58 events out of 100 will have arrival delays of 10 or less. Note that you can filter out the rest of the stuff that poisson.test() generates by asking specifically for the "conf.int" in the output that is returned:

```
> poisson.test(58, 100)$conf.int

[1] 0.4404183 0.7497845

attr(,"conf.level")

[1] 0.95
```

The bolded part of the command line above shows how we used the $ notation to get a report of just the bit of output that we wanted from poisson.test(). This output reports the exact same confidence interval that we saw on the previous page, along with a reminder in the final two lines that we are looking at a 95% confidence interval.

At this point we have all of the knowledge and tools we need to compare two sets of arrival rates. Let's grab a couple of sets of tweets and extract the information we need. First, we will use the function we created in the last chapter to grab the first set of tweets:

```
tweetDF <- TweetFrame("#ladygaga",500)
```

Next, we need to sort the tweets by arrival time, That is, of course, unless you accepted the Chapter Challenge in the previous chapter and built the sorting into your TweetFrame() function.

```
sortweetDF<-tweetDF[order(as.integer( +
                              tweetDF$created)), ]
```

Now, we'll extract a vector of the time differences. In the previous chapter the use of the diff() function occurred within the ArrivalProbability() function that we developed. Here we will use it directly and save the result in a vector:

```
eventDelays<- +
             as.integer(diff(sortweetDF$created))
```

Now we can calculate a few of the things we need in order to get a picture of the arrival delays for Lady Gaga's tweets:

```
> mean(eventDelays)

[1] 30.53707

> sum(eventDelays<=31)

[1] 333
```

So, for Lady Gaga tweets, the mean arrival delay for the next tweet is just short of 31 seconds. Another way of looking at that same statistic is that 333 out of 500 tweets (0.666, about two thirds) arrived within 31 seconds of the previous tweet. We can also ask poisson.test() to show us the confidence interval around that value:

```
> poisson.test(333,500)$conf.int

[1] 0.5963808 0.7415144

attr(,"conf.level")

[1] 0.95
```

So, this result suggests that for 95% of the Lady Gaga samples of tweets that we might pull from the Twitter system, the proportion arriving in 31 seconds or less would fall in this confidence band. In other words, we're not very likely to see a sample with a proportion under 59.6% or over 74.1%. That's a pretty wide band, so we do not have a lot of exactitude here.

Now let's get the same data for Oprah:

```
> tweetDF <- TweetFrame("#oprah",500)

> sortweetDF<-tweetDF[order( +
                  as.integer(tweetDF$created)), ]

> eventDelays<- +
             as.integer(diff(sortweetDF$created))

> mean(eventDelays)

[1] 423.01
```

Hmm, I guess we know who is boss here! Now let's finish the job:

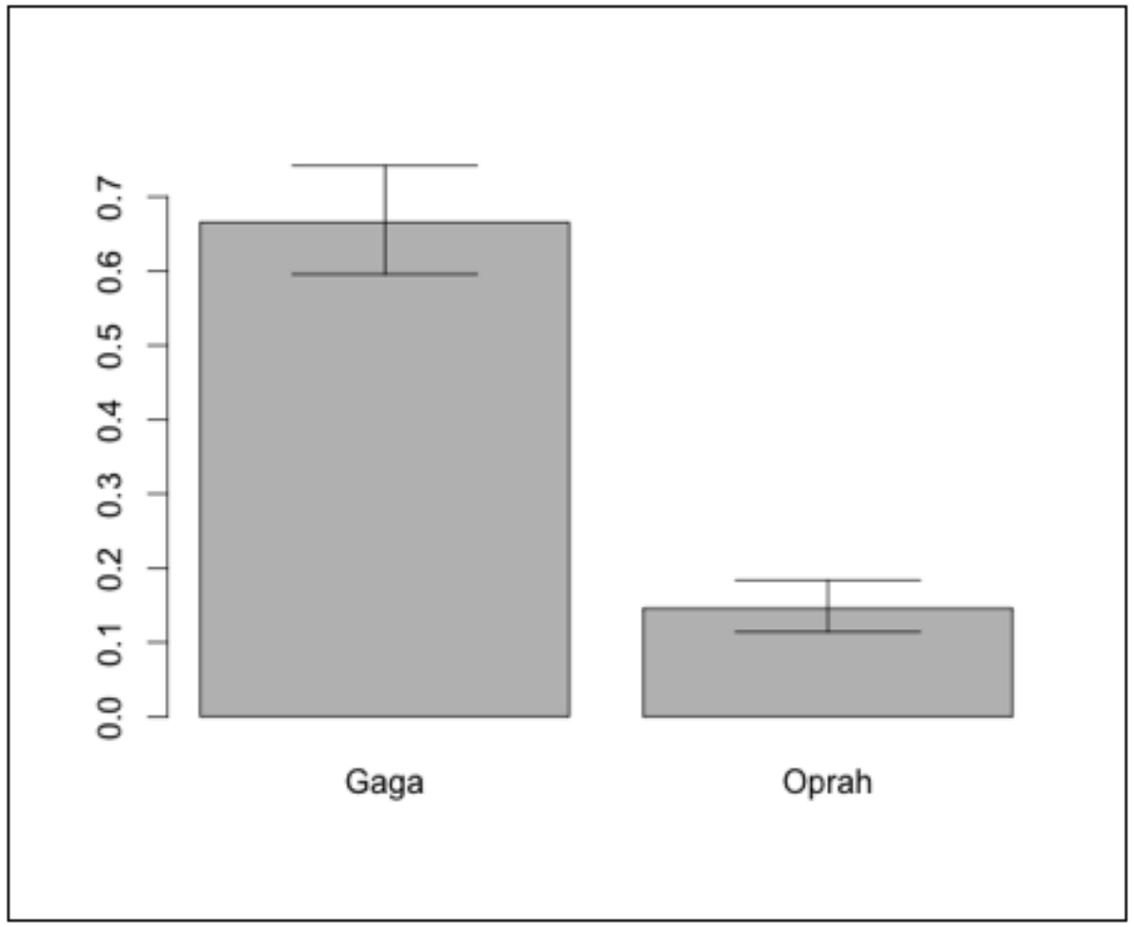
```
> sum(eventDelays<=31)

[1] 73
```

```
> poisson.test(73,500)$conf.int

[1] 0.1144407 0.1835731

attr(,"conf.level")

[1] 0.95
```

The sum() function, above, calculates that only 73 out of Oprah's sample of 500 tweets arrive in an interval of 31 or less. We use 31, the mean of the Lady Gaga sample, because we need to have a common basis of comparison. So for Oprah, the proportion of events that occur in the 31 second timeframe is, $73/500 = 0.146$, or about 14.6%. That's a lot lower than the 66.6% of Lady Gaga tweets, for sure, but we need to look at the confidence interval around that value. So the poisson.test() function just above for Oprah reports that the 95% confidence interval runs from about 11.4% to 18.4%. Note that this confidence interval does not overlap at all with the confidence interval for Lady Gaga, so we have a very strong sense that these two rates are statistically quite distinctive - in other words, this is a difference that was not caused by the random influences that sampling always creates. We can make a bar graph to summarize these differences. We'll use the barplot2() function, which is in a package called gplots(). If you created the EnsurePackage() function a couple of chapters ago, you can use that. Otherwise make sure to load gplots manually:

```
> EnsurePackage("gplots")

> barplot2(c(0.666,0.146), +
        ci.l=c(0.596,0.114), +
        ci.u=c(0.742,0.184), +
        plot.ci=TRUE, +
        names.arg=c("Gaga","Oprah"))
```

This is not a particularly efficient way to use the barplots() function, because we are supplying our data by typing it in, using the c() function to create short vectors of values on the command line. On the first line,, we supply a list of the means from the two samples, expressed as proportions. On the next two lines we first provide the lower limits of the confidence intervals and then the upper limits. The plot.ci=TRUE parameter asks barplot2() to put confidence interval whiskers on each bar. The final line provides labels to put underneath the bars. Here's what we get:



This is not an especially attractive bar plot, but it does represent the information we wanted to display accurately. And with the assistance of this plot, it is easy to see both the substantial difference between the two bars and the fact that the confidence intervals do not overlap.

For one final confirmation of our results, we can ask the poisson.text() function to evaluate our two samples together. This code provides the same information to poisson.test() as before, but now provides the event counts as short lists describing the two samples, with 333 events (under 31 seconds) for Lady Gaga and 73 events for Oprah, in both cases out of 500 events:

```
> poisson.test(c(333,73),c(500,500))

    Comparison of Poisson rates

data:  c(333, 73) time base: c(500, 500)

count1 = 333, expected count1 = 203, p-value <
2.2e-16

alternative hypothesis: true rate ratio is not
equal to 1

95 percent confidence interval:

 3.531401 5.960511

sample estimates:

rate ratio

  4.561644
```

Let's walk through this output line by line. Right after the command, we get a brief confirmation from the function that we're comparing two event rates in this test rather than just evaluating a single rate: "Comparison of Poisson rates." The next line confirms the data we provided. The next line, that begins with "count1 = 333" confirms the basis of of the comparison and then shows a "pooled" count that is the weighted average of 333 and 73. The p-value on that same line represents the position of a probability tail for "false positives." Together with the information on the next line, "alternative hypothesis," this constitutes what statisticians call a "null hypothesis significance test." Although this is widely used in academic research, it contains less useful information than confidence intervals and we will ignore it for now.

The next line, "95% confidence interval," is a label for the most important information, which is on the line that follows. The values of 3.53 and 5.96 represent the upper and lower limits of the 95% *confidence interval around the observed rate ratio* of 4.56 (reported on the final line). So, for 95% of samples that we might draw from twitter, the ratio of the Gaga/Oprah rates might be as low as 3.53 and as high as 5.96. So we can be pretty sure (95% confidence) that Lady Gaga gets tweets at least 3.5 times as fast as Oprah. Because the confidence interval does not include 1, which would be the same thing as saying that the two rates are identical, we can be pretty certain that the observed rate ratio of 4.56 is not a statistical fluke.

For this comparison, we chose two topics that had very distinctive event rates. As the bar chart on the previous page attests, there was a substantial difference between the two samples in the rates of arrival of new tweets. The statistical test confirmed this for us, and although the ability to calculate and visualize the confidence intervals was helpful, we probably could have guessed that such a large difference over a total of 1000 tweets was not a result due to sampling error.

With other topics and other comparisons, however, the results will not be as clear cut. After completing the chapter challenge on the next page, we compared the "#obama" hashtag to the "#romney" hashtag. Over samples of 250 tweets each, Obama had 159 events at or under the mean, while Romney had only 128, for a ratio of 1.24 in Obama's favor. The confidence interval told a different story, however: the lower bound of the confidence interval was 0.978, very close to, but slightly below one. This signifies that we can't rule out the possibility that the two rates are, in fact, equal and that the slightly higher rate (1.24 to 1) that we observed for Obama in this one sample might have come about due to sampling error. When a confidence interval overlaps the point where we consider something to be a "null result" (in this case a ratio of 1:1) we have to take seriously the possibility that peculiarities of the sample(s) we drew created the observed difference, and that a new set of samples might show the opposite of what we found this time.

Chapter Challenge

Write a function that takes two search strings as arguments and that returns the results of a Poisson rate ratio test on the arrival rates of tweets on the two topics. Your function should first run the necessary Twitter searches, then sort the tweets by ascending time of arrival and calculate the two vectors of time differentials. Use the mean of one of these vectors as the basis for comparison and for each vector, count how many events are at or below the mean. Use this information and the numbers of tweets requested to run the poisson.test() rate comparison.

Sources

Barplots

http://addictedtor.free.fr/graphiques/RGraphGallery.php?graph=54

http://biostat.mc.vanderbilt.edu/twiki/pub/Main/StatGraphCourse/graphscourse.pdf

http://rgm2.lab.nig.ac.jp/RGM2/func.php?rd_id=gplots:barplot2

Poisson Distribution

http://books.google.com/books?id=ZKswvkqhygYC&printsec=frontcover

http://www.khanacademy.org/math/probability/v/poisson-process-1

http://www.khanacademy.org/math/probability/v/poisson-process-2

http://stat.ethz.ch/R-manual/R-patched/library/stats/html/Poisson.html

http://stat.ethz.ch/R-manual/R-patched/library/stats/html/poisson.test.html

http://stats.stackexchange.com/questions/10926/how-to-calculate-confidence-interval-for-count-data-in-r

http://www.computing.dcu.ie/~mbezbradica/teaching/CA266/CA266_13_Poisson_Distribution.pdf

R Functions Used in this Chapter

as.integer() - Coerces another data type to integer if possible

barplot2() - Creates a bar graph

c() - Concatenates items to make a list

diff() - Calculates time difference on neighboring cases

EnsurePackage() - Custom function, install() and require() package

for() - Creates a loop, repeating execution of code

hist() - Creates a frequency histogram

mean() - Calculates the arithmetic mean

order() - Provides a list of indices reflecting a new sort order

plot() - Begins an X-Y plot

points() - Adds points to a plot started with plot()

poisson.test() - Confidence intervals for poisson events or ratios

ppois() - Returns a cumulative probability for particular threshold

qpois() - Does the inverse of ppois(): Probability into threshold

rpois() - Generates random numbers fitting a Poisson distribution

sum() - Adds together a list of numbers

TweetFrame() - Custom procedure yielding a dataset of tweets

var() - Calculates variance of a list of numbers

R Script - Create Vector of Probabilities From Delay Times

```
# Like ArrivalProbability, but works with unsorted list
# of delay times
DelayProbability<-function(delays, increment, max)
{
  # Initialize an empty vector
  plist <- NULL

  # Probability is defined over the size of this sample
  # of arrival times
  delayLen <- length(delays)

  # May not be necessary, but checks for input mistake
  if (increment>max) {return(NULL)}

  for (i in seq(increment, max, by=increment))
  {
    # logical test <=i provides list of TRUEs and FALSEs
    # of length = timeLen, then sum() counts the TRUEs
    plist<-c(plist,(sum(delays<=i)/delayLen))
  }
  return(plist)
}
```

CHAPTER 12

String Theory

Prior chapters focused on statistical analysis of tweet arrival times and built on earlier knowledge of samples and distributions. This chapter switches gears to focus on manipulating so-called "unstructured" data, which in most cases means natural language texts. Tweets are again a useful source of data for this because tweets are mainly a short (140 characters or less) character strings.

Yoiks, that last chapter was very challenging! Lots of numbers, lots of statistical concepts, lots of graphs. Let's take a break from all that (temporarily) and focus on a different kind of data for a while. If you think about the Internet, and specifically about the World Wide Web for a while, you will realize: 1) That there are zillions of web pages; and 2) That most of the information on those web pages is "unstructured," in the sense that it does not consist of nice rows and columns of numeric data with measurements of time or other attributes. Instead, most of the data spread out across the Internet is text, digital photographs, or digital videos. These last two categories are interesting, but we will have to postpone consideration of them while we consider the question of text.

Text is, of course, one of the most common forms of human communication, hence the label that researchers use sometimes: natural language. When we say natural language text we mean words created by humans and for humans. With our cool computer technology, we have collectively built lots of ways of dealing with natural language text. At the most basic level, we have a great system for representing individual characters of text inside of computers called "Unicode." Among other things Unicode provides for a binary representation of characters in most of the world's written languages, over 110,000 characters in all. Unicode supersedes ASCII (the American Standard Code for Information Interchange), which was one of the most popular standards (especially in the U.S.) for representing characters from the dawn of the computer age.

With the help of Unicode, most computer operating systems, and most application programs that handle text have a core strategy for representing text as lists of binary codes. Such lists are commonly referred to as "character strings" or in most cases just "strings."

One of the most striking things about strings from a computer programming perspective is that they seem to be changing their length all the time. You can't perform the usual mathematical operations on character strings the way you can with numbers - no multiplication or division - but it is very common to "split" strings into smaller strings, and to "add" strings together to form longer strings. So while we may start out with, "the quick brown fox," we may end up with "the quick brown" in one string and "fox" in another, or we may end up with something longer like, "the quick brown fox jumped over the lazy dog."

Fortunately, R, like most other data handling applications, has a wide range of functions for manipulating, keeping track of, searching, and even analyzing string data. In this chapter, we will use our budding skills working with tweet data to learn the essentials of working with unstructured text data. The learning goal here is simply to become comfortable with examining and manipulating text data. We need these basic skills before we can tackle a more interesting problem.

Let's begin by loading a new package, called "stringr". Although R has quite a few string functions in its core, they tend to be a bit disorganized. So Hadley Wickham, a professor of statistics at Rice University, created this "stringr" package to make a set of string manipulation functions a bit easier to use and more comprehensive. You can install() and library() this package using the point and click features of R-Studio (look in the lower right hand pane under the Packages tab), or if you created the EnsurePackage() function from a couple of chapters back, you can use that:

```
EnsurePackage("stringr")
```

Now we can grab a new set of tweets with our custom function TweetFrame() from a couple of chapters ago (if you need the code, look in the chapter entitled "Tweet, Tweet"; we've also pasted the enhanced function, that sorts the tweets into arrival order, into the end of this chapter):

```
tweetDF <- TweetFrame("#solar",100)
```

This command should return a data frame containing about 100 tweets, mainly having to do with solar energy. You can choose any topic you like - all of the string techniques we examine in this chapter are widely applicable to any text strings. We should get oriented by taking a look at what we retrieved. The head() function can return the first entries in any vector or list:

```
Console ~/Documents/DataScienceBook/twitter/
> head(tweetDF,1)

text
97 If your energy needs increase after you install a #solar system can you
answer! http://t.co/ims8gDWW
   favorited replyToSN             created truncated replyToSID
97     FALSE        NA 2012-04-13 17:00:54     FALSE       <NA>
                    id replyToUID
97 190847037594533888         NA

97 &lt;a href=&quot;http://www.hootsuite.com&quot; rel=&quot;nofollow&quot;
    screenName
97 solarponics
```

We provide a screen shot from R-Studio here just to preserve the formatting of this output. In the left hand margin, the number 97 represents R's indexing of the original order in which the tweet was received. The tweets were re-sorted into arrival order by our enhanced TweetFrame() function (see the end of the chapter for code). So this is the first element in our dataframe, but internally R has numbered it as 97 out of the 100 tweets we obtained. On the first line of the output, R has place the label "text" and this is the field name of the column in the dataframe that contains the texts of the tweets. Other dataframe fields that we will not be using in this chapter include: "favorited," "replyToSN," and "truncated." You may also recognize the field name "created" which contains the POSIX format time and date stamp that we used in previous chapters. Generally speaking, R has placed the example data (from tweet 97) that goes with the field name just underneath it, but the text justification can be confusing, and it makes this display very hard to read. For example, there is a really long number that starts with "1908" that is the unique numeric identifier (a kind of serial number) for this tweet. The field name "id" appears just above it, but is right justified (probably because the field is a number). The most important fact for us to note is that if we want to work with the text string that is the tweet itself, we need to use the field name "text." Let's see if we can get a somewhat better view if we use the head() function just on the text field. This command should provide just the first 2 entries in the "text" column of the dataframe:

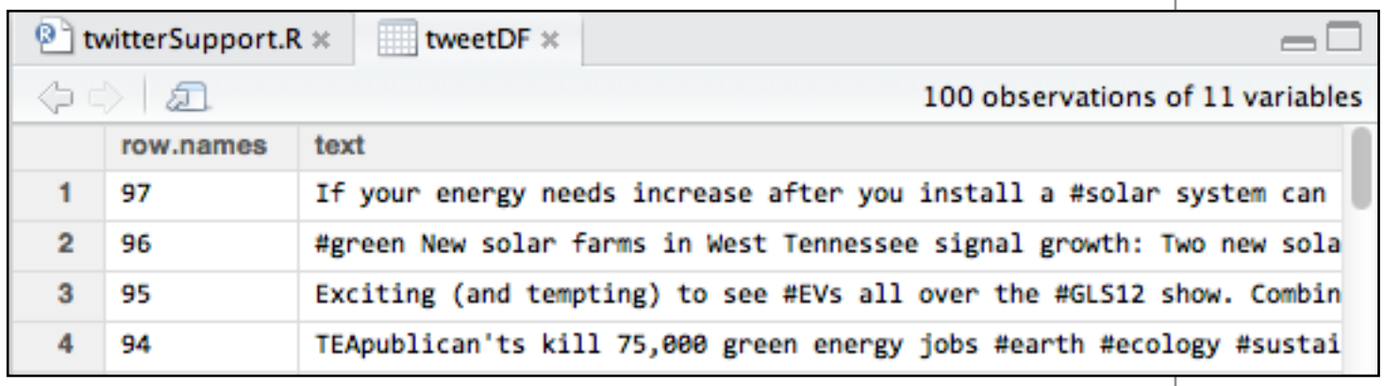
```
head(tweetDF$text,2)
```

```
 [1] "If your energy needs increase after you in-
stall a #solar system can you upgrade? Our ex-
perts have the answer! http://t.co/ims8gDWW"
```

```
 [2] "#green New solar farms in West Tennessee
signal growth: Two new solar energy farms produc-
ing electricity ... http://t.co/37PKAF3N #solar"
```

A couple of things which will probably seem obvious, but are nonetheless important to point out: The [1] and [2] are not part of the tweet, but are the typical line numbers that R uses in its output. The actual tweet text is between the double quotes. You can see the hashtag "#solar" appears in both tweets, which makes sense because this was our search term. There is also a second hashtag in the first tweet "#green" so we will have to be on the lookout for multiple hashtags. There is also a "shortened" URL in each of these tweets. If a Twitter user pastes in the URL of a website to which they want to refer people, the Twitter software automatically shortens the URL to something that begins with "http://t.co/" in order to save space in the tweet.

An even better way to look at these data, including the text and the other fields is to use the data browser that is built into R-Studio. If you look in the upper right hand pane of R-Studio, and make sure



that the Workspace tab is clicked, you should see a list of available dataframes, under the heading "Data." One of these should be "tweetDF." If you click on tweetDF, the data browser will open in the upper left hand pane of R-Studio and you should be able to see the first field or two of the first dozen rows. Here's a screen shot:

This screen shot confirms what we observed in the command line output, but gives us a much more appealing and convenient way of looking through our data. Before we start to manipulate our strings, let's attach() tweetDF so that we don't have to keep using the $ notation to access the text field. And before that, let's check what is already attached with the search() function:

```
> search()

 [1] ".GlobalEnv"        "sortweetDF"        "package:gplots"

 [4] "package:KernSmooth" "package:grid"      "package:caTools"
```

We've truncated this list to save space, but you can see on the first line "sortweetDF" left over from our work in a previous chapter. The other entries are all function packages that we want to keep active. So let's detach() sortweetDF and attach tweetDF:

```
> detach(sortweetDF)

> attach(tweetDF)
```

These commands should yield no additional output. If you get any messages about "The following object(s) are masked from..." you should run search() again and look for other dataframes that should be detached before proceeding. Once you can run attach("tweetDF") without any warnings, you can be sure that the fields in this dataframe are ready to use without interference.

The first and most basic thing to do with strings is to see how long they are. The stringr package gives us the str_length() function to accomplish this task:

```
> str_length(text)
 [1] 130 136 136 128  98  75 131 139  85 157 107  49  75 139 136 136
[17] 136  72  73 136 157 123 160 142 142 122 122 122 122 134  82  87
[33]  89 118  94  74 103  91 136 136 151 136 139 135  70 122 122 136
[49] 123 111  83 136 137  85 154 114 117  98 125 138 107  92 140 119
[65]  92 125  84  81 107 107  73  73 138  63 137 139 131 136 120 124
[81] 124 114  78 118 138 138 116 112 101  94 153  79  79 125 125 102
[97] 102 139 138 153
```

There are all of the string lengths of the texts, reported to the command line. It is interesting to find that there are a few of them (like the very last one) that are longer than 140 characters:

```
> tail(text,1)
[1] "RT @SolarFred: Hey, #solar & wind people. Tell @SpeakerBoehner and @Reuters that YOU have a green job and proud to be providing energy Independence to US"
```

As you can see, the tail() command works like the head() command except from the bottom up rather than the top down. So we have learned that under certain circumstances Twitter apparently does allow tweets longer than 140 characters. Perhaps the initial phrase "RT @SolarFred" does not count against the total. By the way "RT" stands for "retweet" and it indicates when the receiver of a tweet has passed along the same message to his or her followers.

We can glue the string lengths onto the respective rows in the dataframe by creating a new field/column:

```
tweetDF$textlen <- str_length(text)
```

After running this line of text, you should use the data browser in R-studio to confirm that the tweetDF now has a new column of data labeled "textlen". You will find the new column all the way on the rightmost side of the dataframe structure. One peculiarity of the way R treats attached data is that you will not be able to access the new field without the $ notation unless you detach() and then again attach() the data frame. One advantage of grafting this new field onto our existing dataframe is that we can use it to probe the dataframe structure:

```
> detach(tweetDF)
> attach(tweetDF)
> tweetDF[textlen>140, "text"]
[1] "RT @andyschonberger: Exciting (and tempting) to see #EVs all over the #GLS12 show. Combine EVs w #solar generation and we have a winner! http://t.co/NVsfq4G3"
```

We've truncated the output to save space, but in the data we are using here, there were nine tweets with lengths greater than 140. Not all of them had "RT" in them, though, so the mystery remains. An important word about the final command line above, though: We're using the square brackets notation to access the elements of tweetDF. In the first entry, "textlen>140", we're using a conditional expression to control which rows are reported. Only those rows where our new field "textlen" contains a quantity larger than 140

will be reported to the output. In the second entry within square brackets, "text" controls which columns are reported onto the output. The square bracket notation is extremely powerful and sometimes a little unpredictable and confusing, so it is worth experimenting with. For example, how would you change that last command above to report *all* of the columns/fields for the matching rows? Or how would you request the "screenName" column instead of the "text" column? What would happen if you substituted the number 1 in place of "text" on that command?

The next common task in working with strings is to count the number of words as well as the number of other interesting elements within the text. Counting the words can be accomplished in several ways. One of the simplest ways is to count the separators between the words - these are generally spaces. We need to be careful not to over count, if someone has mistakenly typed two spaces between a word, so let's make sure to take out doubles. The str_replace_all() function from stringr can be used to accomplish this:

```
> tweetDF$modtext <- str_replace_all(text,"  "," ")

> tweetDF$textlen2 <- str_length(tweetDF$modtext)

> detach(tweetDF)

> attach(tweetDF)

> tweetDF[textlen != textlen2,]
```

The first line above uses the str_replace_all() function to substitute the one string in place of another as many times as the matching string appears in the input. Three arguments appear on the function above: the first is the input string, and that is tweetDF$text (although we've referred to it just as "text because the dataframe is attached). The second argument is the string to look for and the third argument is the string to substitute in place of the first. Note that here we are asking to substitute one space any time that two in a row are found. Almost all computer languages have a function similar to this, although many of them only supply a function that replaces the *first* instance of the matching string.

In the second command we have calculated a new string length variable based on the length of the strings where the substitutions have occurred. We preserved this in a new variable/field/column so that we can compare it to the original string length in the final command. Note the use of the bracket notation in R to address a certain subset of rows based on where the inequality is true. So here we are looking for a report back of all of the strings whose lengths changed. In the tweet data we are using here, the output indicated that there were seven strings that had their length reduced by the elimination of duplicate spaces.

Now we are ready to count the number of words in each tweet using the str_count() function. If you give it some thought, it should be clear that generally there is one more word than there are spaces. For instance, in the sentence, "Go for it," there are two spaces but three words. So if we want to have an accurate count, we should add one to the total that we obtain from the str_count() function:

```
> tweetDF$wordCount<-(str_count(modtext," ") + 1)

> detach(tweetDF)

> attach(tweetDF)

> mean(wordCount)
```

```
[1] 14.24
```

In this last command, we've asked R to report the mean value of the vector of word counts, and we learn that on average a tweet in our dataset has about 14 words in it.

Next, let's do a bit of what computer scientists (and others) call "parsing." Parsing is the process of dividing a larger unit, like a sentence, into smaller units, like words, based on some kind of rule. In many cases, parsing requires careful use of pattern matching. Most computer languages accomplish pattern matching through the use of a strategy called "regular expressions." A regular expression is a set of symbols used to match patterns. For example, [a-z] is used to match any lowercase letter and the asterisk is used to represent a sequence of zero or more characters. So the regular expression "[a-z]*" means, "match a sequence of zero or more lowercase characters.

If we wanted to parse the retweet sequence that appears at the beginning of some of the tweets, we might use a regular expression like this: "RT @[a-z,A-Z]*: ". Each character up tot he square bracket is a "literal" that has to match exactly. Then the "[a-z,A-Z]*" lets us match any sequence of uppercase and lowercase characters. Finally, the ": " is another literal that matches the end of the sequence. You can experiment with it freely before you commit to using a particular expression, by asking R to echo the results to the command line, using the function str_match() like this:

```
str_match(modtext,"RT @[a-z,A-Z]*: ")
```

Once you are satisfied that this expression matches the retweet phrases properly, you can commit the results to a new column/field/variable in the dataframe:

```
> tweetDF$rt <- str_match(modtext,"RT @[a-z,A-Z]*: ")
> detach(tweetDF)
> attach(tweetDF)
```

Now you can review what you found by echoing the new variable "rt" to the command line or by examining it in R-studio's data browser:

```
> head(rt, 10)
      [,1]
 [1,] NA
 [2,] NA
 [3,] NA
 [4,] NA
 [5,] NA
 [6,] NA
 [7,] NA
 [8,] "RT @SEIA: "
 [9,] NA
[10,] "RT @andyschonberger: "
```

This may be the first time we have seen the value "NA." In R, NA means that there is no value available, in effect that the location is empty. Statisticians also call this missing data. These NAs appear in cases where there was no match to the regular expression that we provided to the function str_match(). So there is nothing wrong

here, this is an expected outcome of the fact that not all tweets were retweets. If you look carefully, though, you will see something else that is interesting.

R is trying to tell us something with the bracket notation. At the top of the list there is a notation of [,1] which signifies that R is showing us the first column of something. Then, each of the entries looks like [#,] with a row number in place of # and an empty column designator, suggesting that R is showing us the contents of a row, possibly across multiple columns. This seems a bit mysterious, but a check of the documentation for str_match() reveals that it returns a *matrix* as its result. This means that tweetDF$rt could potentially contain its own rectangular data object: In effect, the variable rt could itself contain than one column!

In our case, our regular expression is very simple and it contains just one chunk to match, so there is only one column of new data in tweetDF$rt that was generated form using str_match(). Yet the full capability of regular expressions allows for matching a whole sequence of chunks, not just one, and so str_match() has set up the data that it returns to prepare for the eventuality that each row of tweetDF$rt might actually have a whole list of results.

If, for some reason, we wanted to simplify the structure of tweetDF$rt so that each element was simply a single string, we could use this command:

```
tweetDF$rt <- tweetDF$rt[ ,1]
```

This assigns to each element of tweetDF$rt the contents of the first column of the matrix. If you run that command and reexamine tweetDF$rt with head() you will find the simplified structure: no more column designator.

For us to be able to make some use of the retweet string we just isolated, we probably should extract just the "screenname" of the individual whose tweet got retweeted. A screenname in Twitter is like a username, it provides a unique identifier for each person who wants to post tweets. An individual who is frequently retweeted by others may be more influential because their postings reach a wider audience, so it could be useful for us to have a listing of all of the screennames without the extraneous stuff. This is easy to do with str_replace(). Note that we used str_replace_all() earlier in the chapter, but we don't need it here, because we know that we are going to replace just one instance of each string:

```
tweetDF$rt<-str_replace(rt, "RT @","")

tweetDF$rt<-str_replace(rt,": ","")

> tail(rt, 1)

       [,1]

[100,] "SolarFred"

tweetDF$rt <- tweetDF$rt[ ,1]
```

In the first command, we substitute the empty string in place of the four character prefix "RT @", while in the second command we substitute the empty string in place of the two character suffix ": ". In each case we assign the resulting string back to tweetDF$rt. You may be wondering why sometimes we create a new column or field when we calculate some new data while other times we do not. The golden rule with data columns is never to mess with the

original data that was supplied. When you are working ona "derived" column, i.e., one that is calculated from other data, it may require several intermediate steps to get the data looking the way you want. In this case, rt is a derived column that we extracted from the text field of the tweet and our goal was to reduce it to the bare screenname of the individual whose post was retweeted. So these commands, which successfully overwrite rt with closer and closer versions of what we wanted, were fair game for modification.

You may also have noticed the very last command. It seems that one of our steps, probably the use of str_match() must have "matrix-ized" our data again, so we use the column trick that appeared earlier in this chapter to flatten the matrix back to a single column of string data.

```
Console ~/Documents/DataScienceBook/twitter/
> table(as.factor(rt))

  EarthTechling     FeedTheGrid      FirstSolar   GreenergyNews
              1               2               1               1
         RayGil            SEIA       SolarFred   SolarIndustry
              1               3               2               1
    SolarNovus andyschonberger deepgreendesign      gerdvdlogt
              1               2               1               2
           seia       solarfred  thesolsolution
              2               1               1
```

This would be a good point to visualize what we have obtained. Here we introduce two new functions, one which should seem familiar and one that is quite new:

```
table(as.factor(rt))
```

The as.factor() function is a type/mode coercion and just a new one in a family we have seen before. In previous chapters we used as.integer() and as.character() to perform other conversions. In R a factor is a collection of descriptive labels and corresponding unique identifying numbers. The identifying numbers are not usually visible in outputs. Factors are often used for dividing up a dataset into categories. In a survey, for instance, if you had a variable containing the gender of a participant, the variable would frequently be in the form of a factor with (at least) two distinct categories (or what statisticians call levels), male and female. Inside R, each of these categories would be represented as a number, but the corresponding label would usually be the only thing you would see as output. As an experiment, try running this command:

```
str(as.factor(rt))
```

This will reveal the "structure" of the data object after coercion.

Returning to the earlier table(as.factor(rt)) command, the table() function takes as input one or more factors and returns a so called contingency table. This is easy to understand for use with just one factor: The function returns a unique list of factor "levels" (unique: meaning no duplicates) along with a count of how many rows/instances there were of each level in the dataset as a whole.

The screen shot on this page shows the command and the output. There are about 15 unique screennames of Twitter users who were retweeted. The highest number of times that a screenname appeared was three, in the case of SEIA. The table() function is used more commonly to create two-way (two dimensional) contingency

tables. We could demonstrate that here if we had two factors, so let's create another factor.

Remember earlier in the chapter we noticed some tweets had texts that were longer than 140 characters. We can make a new variable, we'll call it longtext, that will be TRUE if the original tweet was longer than 140 characters and FALSE if it was not:

```
> tweetDF$longtext <- (textlen>140)

> detach(tweetDF)

> attach(tweetDF)
```

The first command above has an inequality expression on the right hand side. This is tested for each row and the result, either TRUE or FALSE, is assigned to the new variable longtext. Computer scientists sometimes call this a "flag" variable because it flags whether or not a certain attribute is present in the data. Now we can run the table() function on the two factors:

```
> table(as.factor(rt),as.factor(longtext))


                  FALSE TRUE

  EarthTechling       0    1

  FeedTheGrid         2    0

  FirstSolar          1    0

  GreenergyNews       1    0

  RayGil              0    1

  SEIA                3    0

  SolarFred           0    2

  SolarIndustry       1    0

  SolarNovus          1    0

  andyschonberger     0    2

  deepgreendesign     0    1

  gerdvdlogt          2    0

  seia                2    0

  solarfred           1    0

  thesolsolution      1    0
```

For a two-way contingency table, the first argument you supply to table() is used to build up the rows and the second argument is used to create the columns. The command and output above give us a nice compact display of which retweets are longer than 140 characters (the TRUE column) and which are not (the FALSE column). It is easy to see at a glance that there are many in each category. So, while doing a retweet may contribute to having an extra long tweet, there are also many retweets that are 140 characters or less. It seems a little cumbersome to look at the long list of retweet screennames, so we will create another flag variable that indicates whether a tweet text contains a retweet. This will just provide a more compact way of reviewing which tweets have retweets and which do not:

```
> tweetDF$hasrt <- !(is.na(rt))

> detach(tweetDF)
```

```
> attach(tweetDF)

> View(tweetDF)
```

The first command above uses a function we have not encountered bevfore: is.na(). A whole family of functions that start with "is" exists in R (as well as in other programming languages) and these functions provide a convenient way of testing the status or contents of a data object or of a particular element of a data object. The is.na() function tests whether an element of the input variable has the value NA, which we know from earlier in the chapter is R's way of showing a missing value (when a particular data element is empty). So the expression, is.na(rt) will return TRUE if a particular cell of tweetDF$rt contains the empty value NA, and false if it contains some real data. If you look at the name of our new variable, however, which we have called "hasrt" you may see that we want to reverse the sense of the TRUE and FALSE that is.na() returns. To do that job we use the "!" character, which computers scientists may either call "bang" or more accurately, "not." Using "not" is more accurate because the "!" character provides the Boolean NOT function, which changes a TRUE to a FALSE and vice versa. One last little thing is that the View() command causes R-Studio to freshen the display of the dataframe in its upper left hand pane. Let's look again at retweets and long tweet texts:

```
> table(hasrt,longtext)

       longtext

hasrt   FALSE TRUE

  FALSE    76    2

  TRUE     15    7
```

There are more than twice as many extra long texts (7) when a tweet contains a retweet than when it does not.

Let's now follow the same general procedure for extracting the URLs from the tweet texts. As before the goal is to create a new string variable/column on the original dataframe that will contain the URLs for all of those tweets that have them. Additionally, we will create a flag variable that signifies whether or not each tweet contains a URL. Here, as before, we follow a key principle: Don't mess with your original data. We will need to develop a new regular expression in order to locate an extract the URL string from inside of the tweet text. Actually, if you examine your tweet data in the R-Studio data browser, you may note that some of the tweets have more than one URL in them. So we will have to choose our function call carefully and be equally careful looking at the results to make sure that we have obtained what we need.

At the time when this was written, Twitter had imposed an excellent degree of consistency on URLs, such that they all seem to start with the string "http://t.co/". Additionally, it seems that the compacted URLs all contain exactly 8 characters after that literal, composed of upper and lower case letters and digits. We can use str_match_all() to extract these URLs using the following code:

```
str_match_all(text,"http://t.co/[a-z,A-Z,0-9]{8}")
```

We feed the tweetDF$text field as input into this function call (we don't need to provide the tweetDF$ part because this dataframe is attached). The regular expression begins with the 12 literal characters ending with a forward slash. Then we have a regular expression pattern to match. The material within the square brackets matches any upper or lowercase letter and any digit. The numeral

8 between the curly braces at the end say to match the previous pattern exactly eight times. This yields output that looks like this:

```
[[6]]

     [,1]

[1,] "http://t.co/w74X9jci"


[[7]]

     [,1]

[1,] "http://t.co/DZBUoz5L"

[2,] "http://t.co/gmtEdcQI"
```

This is just an excerpt of the output, but there are a couple of important things to note. First, note that the first element is preceded by the notation [[6]]. In the past when R has listed out multiple items on the output, we have seen them with index numbers like [1] and [2]. In this case, however, that could be confusing because each element in the output could have multiple rows (as item [[7]] above clearly shows). So R is using double bracket notation to indicate the ordinal number of each chunk of data in the list, where a given chunk may itself contain multiple elements.

Confusing? Let's go at it from a different angle. Look at the output under the [[7]] above. As we noted a few paragraphs ago, some of those tweets have multiple URLs in them. The str_match_all() function handles this by creating, *for every single row in the tweet data*, a data object that itself contains exactly one column but one or possibly more than one row - one row for each URL that appears in the tweet. So, just as we saw earlier in the chapter, we are getting back from a string function a complex matrix-like data object that requires careful handling if we are to make proper use of it.

The only other bit of complexity is this: What if a tweet contained no URLs at all? Your output from running the str_match_all() function probably contains a few elements that look like this:

```
[[30]]

character(0)


[[31]]

character(0)
```

So elements [[30]] and [[31]] of the data returned from str_match_all() each contain a zero length string. No rows, no columns, just character(0), the so-called null character, which in many computer programming languages is used to "terminate" a string.

Let's go ahead and store the output from str_match_all() into a new vector on tweetDF and then see what we can do to tally up the URLs we have found:

```
> tweetDF$urlist<-str_match_all(text,+
                 "http://t.co/[a-z,A-Z,0-9]{8}")

> detach(tweetDF)

> attach(tweetDF)

> head(tweetDF$urlist,2)

[[1]]
```

```
     [,1]
[1,] "http://t.co/ims8gDWW"


[[2]]
     [,1]
[1,] "http://t.co/37PKAF3N"
```

Now we are ready to wrestle with the problem of how to tally up the results of our URL parsing. Unlike the situation with retweets, where there either was or was not a single retweet indication in the text, we have the possibility of zero, one or more URLs within the text of each tweet. Our new object "urlist" is a multi-dimensional object that contains a single null character, one row/column of character data, or one column with more than one row of character data. The key to summarizing this is the length() function, which will happily count up the number of elements in an object that you supply to it:

```
> length(urlist[[1]])
[1] 1
> length(urlist[[5]])
[1] 0
> length(urlist[[7]])
[1] 2
```

Here you see that double bracket notation again, used as an index into each "chunk" of data, where the chunk itself may have some internal complexity. In the case of element [[1]] above, there is one row, and therefore one URL. For element [[5]] above, we see a zero, which means that length() is telling us that this element has no rows in it at all. Finally, for element [[7]] we see 2, meaning that this element contains two rows, and therefore two URLs.

In previous work with R, we've gotten used to leaving the inside of the square brackets empty when we want to work with a whole list of items, but that won't work with the double brackets:

```
> length(urlist[[]])
Error in urlist[[]] : invalid subscript type 'symbol'
```

The double brackets notation is designed to reference just a single element or component in a list, so empty double brackets does not work as a shorthand for every element in a list. So what we must do if we want to apply the length() function to each element in urlist is to loop. We could accomplish this with a for loop, as we did in the last chapter, using an index quantity such as "i" and substituting i into each expression like this: urlist[[i]]. But let's take this opportunity to learn a new function in R, one that is generally more efficient for looping. The rapply() function is part of the "apply" family of functions, and it stands for "recursive apply." Recursive in this case means that the function will dive down into the complex, nested structure of urlist and repetitively run a function for us, in this case the length() function:

```
> tweetDF$numurls<-rapply(urlist,length)
> detach(tweetDF)
> attach(tweetDF)
> head(numurls,10)
```

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

Excellent! We now have a new field on tweetDF that counts up the number of URLs. As a last step in examining our tweet data, let's look at a contingency table that looks at the number of URLs together with the flag indicating an extra long tweet. Earlier in the chapter, we mentioned that the table() function takes factors as its input. In the command below we have supplied the numurls field to the table() function without coercing it to a factor. Fortunately, the table() function has some built in intelligence that will coerce a numeric variable into a factor. In this case because numurls only takes on the values of 0, 1, or 2, it makes good sense to allow table() to perform this coercion:

```
> table(numurls,longtext)

       longtext

numurls FALSE TRUE

      0    16    3

      1    72    6

      2     3    0
```

This table might be even more informative if we looked at it as proportions, so here is a trick to view proportions instead of counts:

```
> prop.table(table(numurls,longtext))

       longtext

numurls FALSE TRUE

      0  0.16 0.03

      1  0.72 0.06

      2  0.03 0.00
```

That looks familiar! Now, of course, we remember that we had exactly 100 tweets, so each of the counts could be considered a percentage with no further calculation. Still, prop.table() is a useful function to have when you would rather view your contingency tables as percentages rather than counts. We can see from these results that six percent of the tweets have one URL, but only three percent have no URLS.

So, before we close out this chapter, let's look at a three way contingency table by putting together our two flag variables and the number of URLs:

```
> table(numurls,hasrt,longtext)

, , longtext = FALSE

       hasrt

numurls FALSE TRUE

      0    15    1

      1    58   14

      2     3    0


, , longtext = TRUE

       hasrt

numurls FALSE TRUE

      0     0    3
```

```
1     2     4
2     0     0
```

Not sure this entirely solves the mystery, but if we look at the second two-way table above, where longtext = TRUE, it seems that extra long tweets either have a retweet (3 cases), or a single URL (2 cases) or both (4 cases).

When we said we would give statistics a little rest in this chapter, we lied just a tiny bit. Check out these results:

```
> mean(textlen[hasrt&longtext])
[1] 155
> mean(textlen[!hasrt&longtext])
[1] 142
```

In both commands we have requested the mean of the variable textlen, which contains the length of the original tweet (the one without the space stripped out). In each command we have also used the bracket notation to choose a particular subset of the cases. Inside the brackets we have a logical expression. The only cases that will be included in the calculation of the mean are those where the expression inside the brackets evaluates to TRUE. In the first command we ask for the mean tweet length for those tweets that have a retweet AND are extra long (the ampersand is the Boolean AND operator). In the second command we use the logical NOT (the "!" character) to look at only those cases that have extra long text but do not have a retweet. The results are instructive. The really long tweets, with a mean length of 155 characters, are those that have retweets. It seems that Twitter does not penalize an individual who retweets by counting the number of characters in the "RT @SCREENNAME:" string. If you have tried the web interface for Twitter you will see why this makes sense: Retweeting is accomplished with a click, and the original tweet - which after all may already be 140 characters - appears underneath the screenname of the originator of the tweet. The "RT @" string does not even appear in the text of the tweet at that point.

Looking back over this chapter, we took a close look at some of the string manipulation functions provided by the package "stringr". These included some of the most commonly used actions such as finding the length of a string, finding matching text within a string, and doing search and replace operations on a string. We also became aware of some additional complexity in nested data structures. Although statisticians like to work with nice, well-ordered rectangular datasets, computer scientists often deal with much more complex data structures - although these are built up out of parts that we are familiar with such as lists, vectors, and matrices.

Twitter is an excellent source of string data, and although we have not yet done much in analyzing the contents of tweets or their meanings, we have looked at some of the basic features and regularities of the text portion of a tweet. In the next chapter we will become familiar with a few additional text tools and then be in a position to manipulate and analyze text data

Chapter Challenges

Create a function that takes as input a dataframe of tweets and returns as output a list of all of the retweet screennames. As an extra challenge, see if you can reduce that list of screennames to a unique set (i.e., no duplicates) while also generating a count of the number of times that each retweet screenname appeared.

Once you have written that function, it should be a simple matter to copy and modify it to create a new function that extracts a unique list of *hashtags* from a dataframe of tweets. Recall that hashtags begin with the "#" character and may contain any combination of upper and lowercase characters as well as digits. There is no length limit on hashtags, so you will have to assume that a hashtag ends when there is a space or a punctuation mark such as a comma, semicolon, or period.

Sources

http://cran.r-project.org/web/packages/stringr/index.html

http://en.wikipedia.org/wiki/ASCII

http://en.wikipedia.org/wiki/Regular_expression

http://en.wikipedia.org/wiki/Unicode

http://had.co.nz/ (Hadley Wickham)

http://mashable.com/2010/08/14/twitter-140-bug/

http://stat.ethz.ch/R-manual/R-devel/library/base/html/search.html

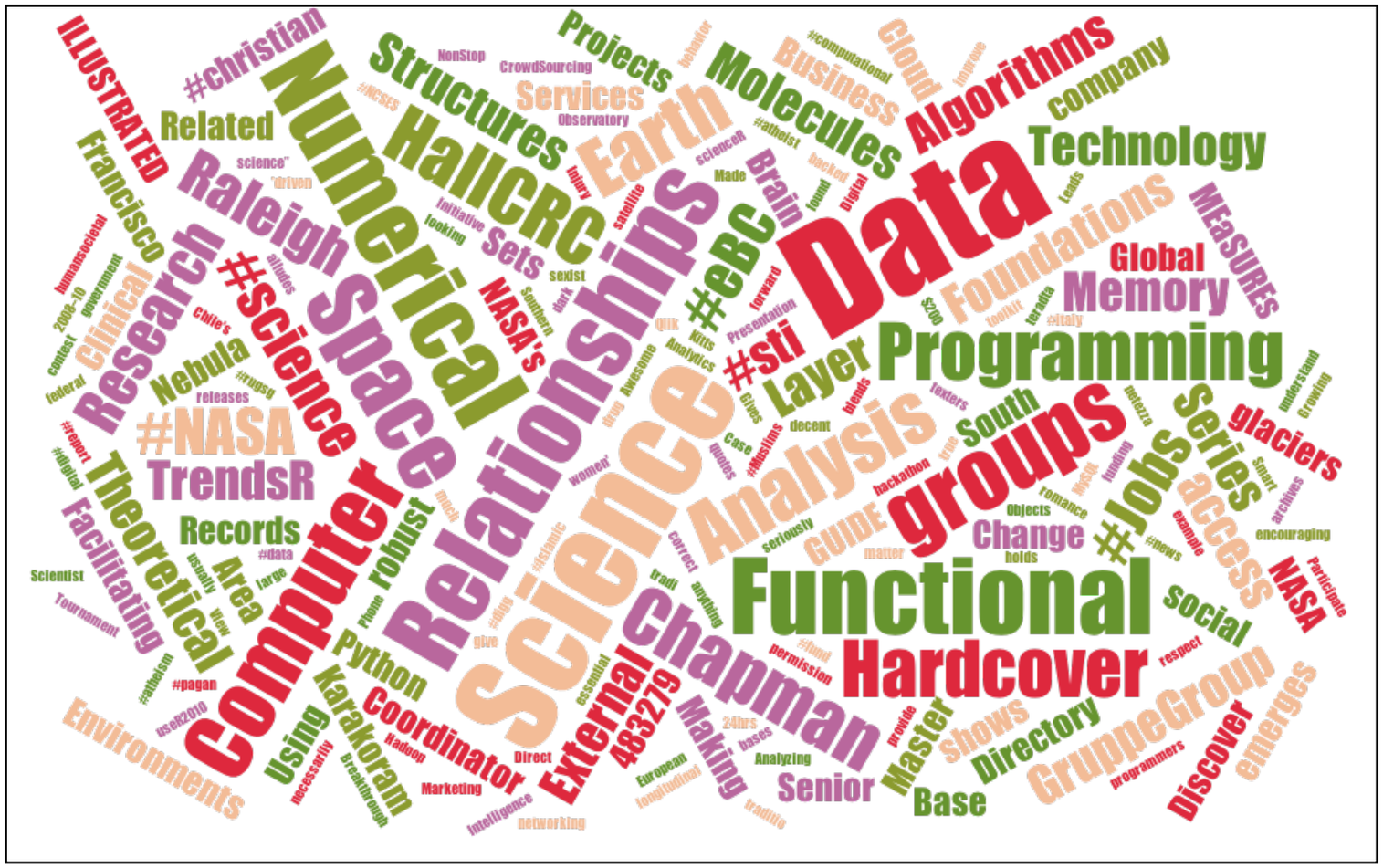
http://stat.ethz.ch/R-manual/R-devel/library/base/html/table.html

R Code for TweetFrame() Function

```
# TweetFrame() - Return a dataframe based on a search of Twit-
ter

TweetFrame<-function(searchTerm, maxTweets)

{

  tweetList <- searchTwitter(searchTerm, n=maxTweets)


  # as.data.frame() coerces each list element into a row

  # lapply() applies this to all of the elements in twtList

  # rbind() takes all of the rows and puts them together

  # do.call() gives rbind() all rows as individual elements

  tweetDF<- do.call("rbind", lapply(tweetList,as.data.frame))


  # This last step sorts the tweets in arrival order

  return(tweetDF[order(as.integer(tweetDF$created)), ])

}
```

CHAPTER 13

Word Perfect



In the previous chapter we mastered some of the most basic and important functions for examining and manipulating text. Now we are in a position to analyze the actual words that appear in text documents. Some of the most basic functions of the Internet, such as keyword search, are accomplished by analyzing the "content" i.e., the words in a body of text.

The picture at the start of this chapter is a so called "word cloud" that was generated by examining all of the words returned from a Twitter search of the term "data science" (using a web application at http://www.jasondavies.com) These colorful word clouds are fun to look at, but they also do contain some useful information. The geometric arrangement of words on the figure is partly random and partly designed and organized to please the eye. Same with the colors. The font size of each word, however, conveys some measure of its importance in the "corpus" of words that was presented to the word cloud graphics program. Corpus, from the Latin word meaning "body," is a word that text analysts use to refer to a body of text material, often consisting of one or more documents. When thinking about a corpus of textual data, a set of documents could really be anything: web pages, word processing documents on your computer, a set of Tweets, or government reports. In most cases, text analysts think of a collection of documents, each of which contains some natural language text, as a corpus if they plan to analyze all the documents together.

The word cloud on the previous page shows that "Data" and "Science" are certainly important terms that came from the search of Twitter, but there are dozens and dozens of less important, but perhaps equally interesting, words that the search results contained. We see words like algorithms, molecules, structures, and research, all of which could make sense in the context of data science. We also see other terms, like #christian, Facilitating, and Coordinator, that don't seem to have the same obvious connection to our original search term "data science." This small example shows one of the fundamental challenges of natural language processing and the closely related area of search: ensuring that the analysis of text produces results that are relevant to the task that the user has in mind.

In this chapter we will use some new R packages to extend our abilities to work with text and to build our own word cloud from data retrieved from Twitter. If you have not worked on the chapter "String Theory" that precedes this chapter, you should probably do so before continuing, as we build on the skills developed there.

Depending upon where you left off after the previous chapter, you will need to retrieve and pre-process a set of tweets, using some of the code you already developed, as well as some new code. At the end of the previous chapter, we have provided sample code for the TweetFrame() function, that takes a search term and a maximum tweet limit and returns a time-sorted dataframe containing tweets. Although there are a number of comments in that code, there are really only three lines of functional code thanks to the power of the twitteR package to retrieve data from Twitter for us. For the activities below, we are still working with the dataframe that we retrieved in the previous chapter using this command:

```
tweetDF <- TweetFrame("#solar",100)
```

This yields a dataframe, tweetDF, that contains 100 tweets with the hashtag #solar, presumably mostly about solar energy and related "green" topics. Before beginning our work with the two new R packages, we can improve the quality of our display by taking out a lot of the junk that won't make sense to show in the word cloud. To accomplish this, we have authored another function that strips out extra spaces, gets rid of all URL strings, takes out the retweet header if one exists in the tweet, removes hashtags, and eliminates references to other people's tweet handles. For all of these transformations, we have used string replacement functions from the stringr package that was introduced in the previous chapter. As an example of one of these transformations, consider this command,

which appears as the second to last line of the CleanTweet() function:

```
tweets <- str_replace_all(tweets,"@[a-z,A-Z]*","")
```

You should feel pretty comfortable reading this line of code, but if not, here's a little more practice. The left hand side is easy: we use the assignment arrow to assign the results of the right hand side expression to a data object called "tweets." Note that when this statement is used inside the function as shown at the end of the chapter, "tweets" is a temporary data object, that is used just within CleanTweets() after which it disappears automatically.

The right hand side of the expression uses the str_replace_all() function from the stringr package. We use the "all" function rather than str_replace() because we are expecting multiple matches within each individual tweet. There are three arguments to the str_replace_all() function. The first is the input, which is a vector of character strings (we are using the temporary data object "tweets" as the source of the text data as well as its destination), the second is the regular expression to match, and the third is the string to use to replace the matches, in this case the empty string as signified by two double quotes with nothing between them. The regular expression in this case is the at sign, "@", followed by zero or more upper and lowercase letters. The asterisk, "*", after the stuff in the square brackets is what indicates the zero or more. That regular expression will match any screenname referral that appears within a tweet.

If you look at a few tweets you will find that people refer to each other quite frequently by their screennames within a tweet, so @SolarFred might occur from time to time within the text of a tweet.

Here's something you could investigate on your own: Can screennames contain digits as well as letters? If so, how would you have to change the regular expression in order to also match the digits zero through nine as part of the screenname? On a related note, why did we choose to strip these screen names out of our tweets? What would the word cloud look like if you left these screennames in the text data?

Whether you typed in the function at the end of this chapter or you plan to enter each of the cleaning commands individually, let's begin by obtaining a separate vector of texts that is outside the original dataframe:

```
> cleanText<-tweetDF$text

> head(cleanText, 10)
```

There's no critical reason for doing this except that it will simplify the rest of the presentation. You could easily copy the tweetDF$text data into another column in the same dataframe if you wanted to. We'll keep it separate for this exercise so that we don't have to worry about messing around with the rest of the dataframe. The head() command above will give you a preview of what you are starting with. Now let's run our custom cleaning function:

```
> cleanText<-CleanTweets(cleanText)

> head(cleanText, 10)
```

Note that we used our "cleanText" data object in the first command above as both the source and the destination. This is an old computer science trick for cutting down on the number of temporary variables that need to be used. In this case it will do exactly what we want, first evaluating the right hand side of the expres-

sion by running our CleanTweets() function with the cleanText object as input and then taking the result that is returned by CleanTweets() and assigning it back into cleanText, thus overwriting the data that was in there originally. Remember that we have license to do whatever we want to cleanText because it is a copy of our original data, and we have left the original data intact (i.e., the text column inside the tweetDF dataframe).

The head() command should now show a short list of tweets with much of the extraneous junk filtered out. If you have followed these steps, cleanText is now a vector of character strings (in this example exactly 100 strings) ready for use in the rest of our work below. We will now use the "tm" package to process our texts. The "tm" in this case refers to "text mining," and is a popular choice among the many text analysis packages available in R. By the way, text mining refers to the practice of extracting useful analytic information from corpora of text (corpora is the plural of corpus). Although some people use text mining and natural language processing interchangeably, there are probably a couple subtle differences worth considering. First, the "mining" part of text mining refers to an area of practice that looks for unexpected patterns in large data sets, or what some people refer to as knowledge discovery in databases. In contrast, natural language processing reflects a more general interest in understanding how machines can be programmed (or learn on their own) how to digest and make sense of human language. In a similar vein, text mining often focuses on statistical approaches to analyzing text data, using strategies such as counting word frequencies in a corpus. In natural language processing, one is more likely to hear consideration given to linguistics, and therefore to the processes of breaking text into its component grammatical pieces such as nouns and verbs. In the case of the "tm" add on package for R, we are definitely in the statistical camp, where the main process is to break down a corpus into sequences of words and then to tally up the different words and sequences we have found.

To begin, make sure that the tm package is installed and "libraryed" in your copy of R and R-Studio. You can use the graphic interface in R-Studio for this purpose or the EnsurePackage() function that we wrote in a previous chapter. Once the tm package is ready to use, you should be able to run these commands:

```
> tweetCorpus<-Corpus(VectorSource(cleanText))

> tweetCorpus

A corpus with 100 text documents

> tweetCorpus<-tm_map(tweetCorpus, tolower)

> tweetCorpus<-tm_map(tweetCorpus, removePunctuation)

> tweetCorpus<-tm_map(tweetCorpus,removeWords,+
                                    stopwords('english'))
```

In the first step above , we "coerce" our cleanText vector into a custom "Class" provided by the tm package and called a "Corpus," storing the result in a new data object called "tweetCorpus." This is the first time we have directly encountered a "Class." The term "class" comes from an area of computer science called "object oriented programming." Although R is different in many ways from object-oriented languages such as Java, it does contain many of the most fundamental features that define an object oriented language. For our purposes here, there are just a few things to know about a class. First, a class is nothing more or less than a definition for the structure of a data object. Second, classes use basic data types, such

as numbers, to build up more complex data structures. For example, if we made up a new "Dashboard" class, it could contain one number for "Miles Per Hour," another number for "RPM," and perhaps a third one indicating the remaining "Fuel Level." That brings up another point about Classes: users of R can build their own. In this case, the author of the tm package, Ingo Feinerer, created a new class, called Corpus, as the central data structure for text mining functions. (Feinerer is a computer science professor who works at the Vienna University of Technology in the Database and Artificial Intelligence Group.) Last, and most important for this discussion, a Class not only contains definitions about the structure of data, it also contains references to functions that can work on that Class. In other words, a Class is a data object that carries with it instructions on how to do operations on it, from simple things like add and subtract all the way up to complicated operations such as graphing.

In the case of the tm package, the Corpus Class defines the most fundamental object that text miners care about, a corpus containing a collection of documents. Once we have our texts stored in a Corpus, the many functions that the tm package provides to us are available. The last three commands in the group above show the use of the tm_map() function, which is one of the powerful capabilities provided by tm. In each case where we call the tm_map() function, we are providing tweetCorpus as the input data, and then we are providing a command that undertakes a transformation on the corpus. We have done three transformations here, first making all of the letters lowercase, then removing the punctuation, and finally taking out the so called "stop" words.

The stop words deserve a little explanation. Researchers who developed the early search engines for electronic databases found that certain words interfered with how well their search algorithms worked. Words such as "the," "a," and "at" appeared so commonly in so many different parts of the text that they were useless for differentiating between documents. The unique and unusual nouns, verbs, and adjectives that appeared in a document did a much better job of setting a document apart from other documents in a corpus, such that researchers decided that they should filter out all of the short, commonly used words. The term "stop words" seems to have originated in the 1960s to signify words that a computer processing system would throw out or "stop using" because they had little meaning in a data processing task. To simplify the removal of stop words, the tm package contains lists of such words for different languages. In the last command on the previous page we requested the removal of all of the common stop words.

At this point we have processed our corpus into a nice uniform "bag of words" that contains no capital letters, punctuation, or stop words. We are now ready to conduct a kind of statistical analysis of the corpus by creating what is known as a "term-document matrix." The following command from the tm package creates the matrix:

```
> tweetTDM<-TermDocumentMatrix(tweetCorpus)

> tweetTDM

A term-document matrix (375 terms, 100 documents)

Non-/sparse entries: 610/36890

Sparsity           : 98%
```

```
Maximal term length: 21

Weighting          : term frequency (tf)
```

A term-document matrix, also sometimes called a document-term matrix, is a rectangular data structure with terms as the rows and documents as the columns (in other uses you may also make the terms as columns and documents as rows). A term may be a single word, for example, "biology," or it could also be a compound word, such as "data analysis." The process of determining whether words go together in a compound word can be accomplished statistically by seeing which words commonly go together, or it can be done with a dictionary. The tm package supports the dictionary approach, but we have not used a dictionary in this example. So if a term like "data" appears once in the first document, twice in the second document, and not at all in the third document, then the column for the term data will contain 1, 2, 0.

The statistics reported when we ask for tweetTDM on the command line give us an overview of the results. The TermDocumentMatrix() function extracted 375 different terms from the 100 tweets. The resulting matrix mainly consists of zeros: Out of 37,500 cells in the matrix, only 610 contain non-zero entries, while 36,890 contain zeros. A zero in a cell means that that particular term did not appear in that particular document. The maximal term length was 21 words, which an inspection of the input tweets indicates is also the maximum word length of the input tweets. Finally, the last line, starting with "Weighting" indicates what kind of statistic was stored in the term-document matrix. In this case we used the default, and simplest, option which simply records the count of the number of times a term appears across all of the documents in the corpus. You can peek at what the term-document matrix contains by using the inspect function:

```
inspect(tweetTDM)
```

Be prepared for a large amount of output. Remember the term "sparse" in the summary of the matrix? Sparse refers to the overwhelming number of cells that contain zero - indicating that the particular term does not appear in a given document. Most term document matrices are quite sparse. This one is 98% sparse because 36890/37500 = 0.98. In most cases we will need to cull or filter the term-document matrix for purposes of presenting or visualizing it. The tm package provides several methods for filtering out sparsely used terms, but in this example we are going to leave the heavy lifting to the word cloud package.

As a first step we need to install and library() the "wordcloud" package. As with other packages, either use the package interface in R-Studio or the EnsurePackage() function that we wrote a few chapters ago. The wordcloud package was written by freelance statistician Ian Fellows, who also developed the "Deducer" user interface for R. Deducer provides a graphical interface that allows users who are more familiar with SPSS or SAS menu systems to be able to use R without resorting to the command line.

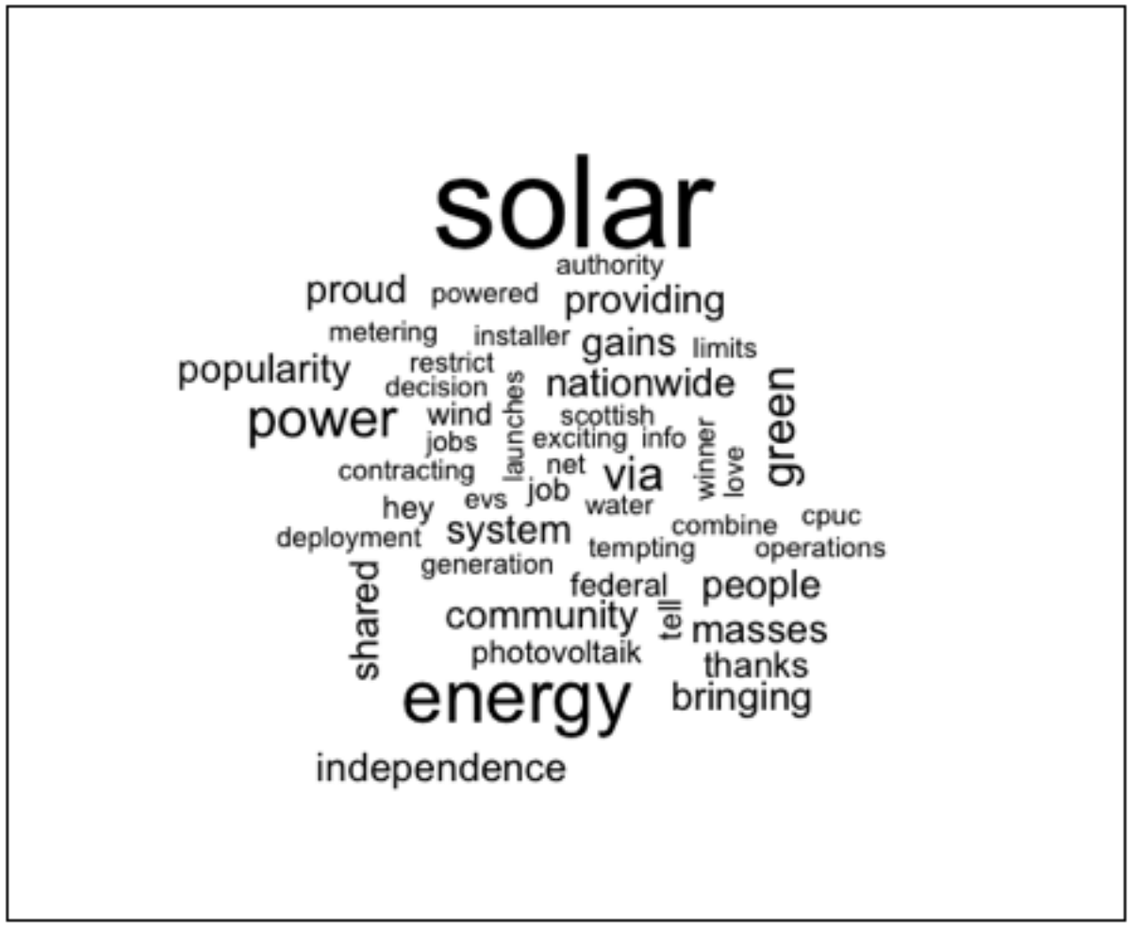
Once the wordcloud package is loaded, we need to do a little preparation to get our data ready to submit to the word cloud generator function. That function expects two vectors as input arguments, the first a list of the terms, and the second a list of the frequencies of occurrence of the terms. The list of terms and frequencies must be sorted with the most frequent terms appearing first. To accomplish this we first have to coerce our tweet data back into

a plain data matrix so that we can sort it by frequency. The first command below accomplishes this:

```
> tdMatrix <- as.matrix(tweetTDM)

> sortedMatrix<-sort(rowSums(tdMatrix),+
                              decreasing=TRUE)

> cloudFrame<-data.frame( +
   word=names(sortedMatrix),freq=sortedMatrix)

> wordcloud(cloudFrame$word,cloudFrame$freq)
```

In the next command above, we are accomplishing two things in one command: We are calculating the sums across each row, which gives us the total frequency of a term across all of the different tweets/documents. We are also sorting the resulting values with the highest frequencies first. The result is a named list: Each item of the list has a frequency and the name of each item is the term to which that frequency applies.

In the second to last command above, we are extracting the names from the named list in the previous command and binding them together into a dataframe with the frequencies. This dataframe, "cloudFrame", contains exactly the same information as the named list. "sortedMatrix," but cloudFrame has the names in a separate column of data. This makes it easier to do the final command above, which is the call to the wordcloud() function. The wordcloud() function has lots of optional parameters for making the word cloud more colorful, controlling its shape, and controlling how frequent an item must be before it appears in the cloud, but we have used the default settings for all of these parameters for the sake of simplicity. We pass to the wordcloud() function the term list and frequency list that we bound into the dataframe and wordcloud() produces the nice graphic that you see below.



If you recall the Twitter search that we used to retrieve those tweets (#solar) it makes perfect sense that "solar" is the most frequent term (even though we filtered out all of the hashtags. The next most popular term is "energy" and after that there are a variety of related words such as "independence," "green," "wind," and "metering."

Chapter Challenge

Develop a function that builds upon previous functions we have developed, such as TweetFrame() and CleanTweets(), to take a search term, conduct a Twitter search, clean up the resulting texts, formulate a term-document matrix, and submit resulting term frequencies to the wordcloud() function. Basically this would be a "turnkey" package that would take a Twitter search term and produce a word cloud from it, much like the Jason Davies site described at the beginning of this chapter.

Sources Used in This Chapter

http://cran.r-project.org/web/packages/wordcloud/wordcloud.pdf

http://www.dbai.tuwien.ac.at/staff/feinerer/

http://en.wikipedia.org/wiki/Document-term_matrix

http://en.wikipedia.org/wiki/Stop_words

http://en.wikipedia.org/wiki/Text_mining

http://stat.ethz.ch/R-manual/R-devel/library/base/html/colSums.html

http://www.jasondavies.com/wordcloud/

R Code for CleanTweets() Function

```
# CleanTweets() - Takes the junk out of a vector of
# tweet texts
CleanTweets<-function(tweets)
{
  # Remove redundant spaces
  tweets <- str_replace_all(tweets,"  "," ")
  # Get rid of URLs
  tweets <- str_replace_all(tweets, +
              "http://t.co/[a-z,A-Z,0-9]{8}","")
  # Take out retweet header, there is only one
  tweets <- str_replace(tweets,"RT @[a-z,A-Z]*: ","")
  # Get rid of hashtags
  tweets <- str_replace_all(tweets,"#[a-z,A-Z]*","")
  # Get rid of references to other screennames
  tweets <- str_replace_all(tweets,"@[a-z,A-Z]*","")
  return(tweets)
}
```

CHAPTER 14

Storage Wars

Before now we have only used small amount of data that we typed in ourselves, or somewhat larger amounts that we extracted from Twitter. The world is full of other sources of data, however, and we need to examine how to get them into R, or at least how to make them accessible for manipulation in R. In this chapter, we examine various ways that data are stored, and how to access them.

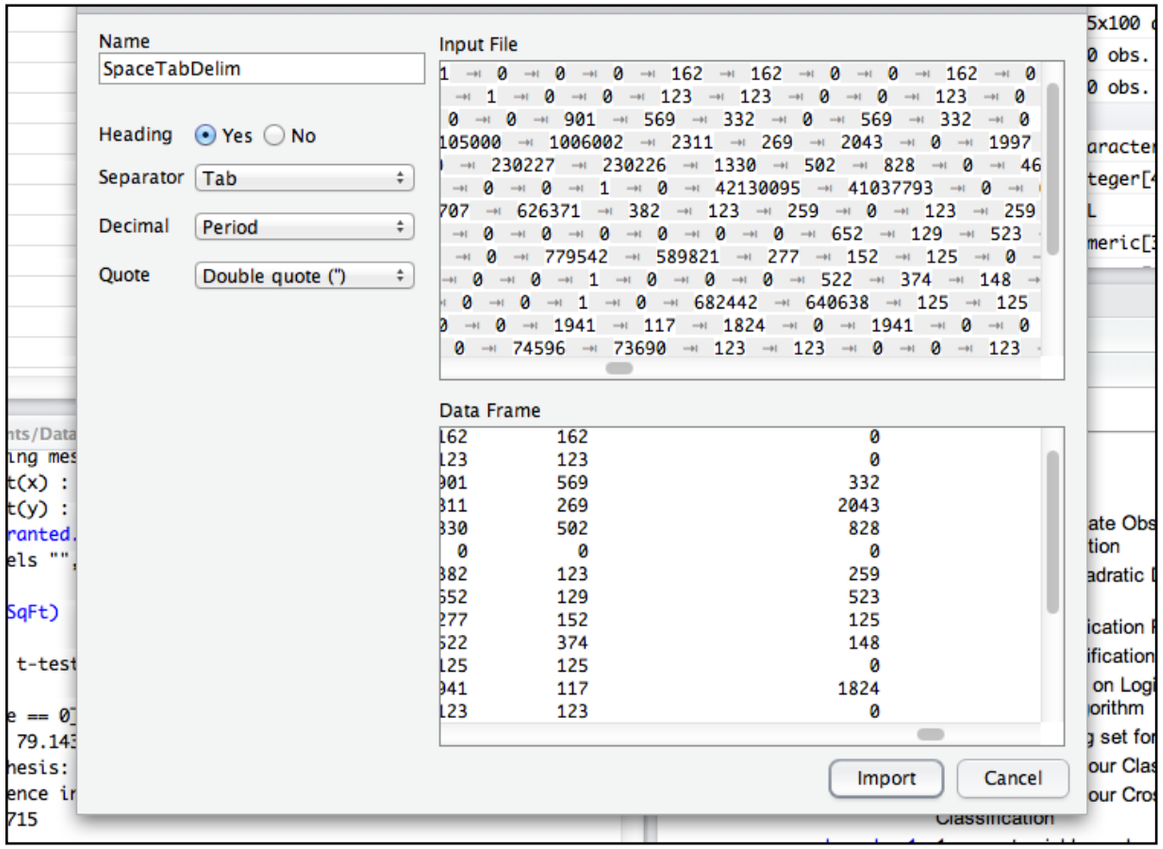
Most people who have watched the evolution of technology over recent decades remember a time when storage was expensive and it had to be hoarded like gold. Over the last few years, however, the accelerating trend of Moore's Law has made data storage almost "too cheap to meter" (as they used to predict about nuclear power). Although this opens many opportunities, it also means that people keep data around for a long time, since it doesn't make sense to delete anything, and they may keep data around in many different formats. As a result, the world is full of different data formats, some of which are proprietary - designed and owned by a single company such as SAS - and some of which are open, such as the lowly but infinitely useful "comma separated variable," or CSV format.

In fact, one of the basic dividing lines in data formats is whether data are human readable or not. Formats that are not human readable, often called binary formats, are very efficient in terms of how much data they can pack in per kilobyte, but are also squirrelly in the sense that it is hard to see what is going on inside of the format. As you might expect, human readable formats are inefficient from a storage standpoint, but easy to diagnose when something goes wrong. For high volume applications, such as credit card processing, the data that is exchanged between systems is almost universally in binary formats. When a data set is archived for later reuse, for example in the case of government data sets available to the public, they are usually available in multiple formats, at least one of which is a human readable format.

Another dividing line, as mentioned above is between proprietary and open formats. One of the most common ways of storing and sharing small datasets is as Microsoft Excel spreadsheets. Although this is a proprietary format, owned by Microsoft, it has also become a kind of informal and ubiquitous standard. Dozens of different software applications can read Excel formats (there are several different formats that match different versions of Excel). In contrast, the OpenDocument format is an open format, managed by a standards consortium, that anyone can use without worrying what the owner might do. OpenDocument format is based on XML, which stands for Extensible markup language. XML is a whole topic in and of itself, but briefly it is a data exchange format designed specifically to work on the Internet and is both human and machine readable. XML is managed by the W3C consortium, which is responsible for developing and maintaining the many standards and protocols that support the web.

As an open source program with many contributors, R offers a wide variety of methods of connecting with external data sources. This is both a blessing and a curse. There is a solution to almost any data access problem you can imagine with R, but there is also a dizzying array of options available such that it is not always obvious what to choose. We'll tackle this problem in two different ways. In the first half of this chapter we will look at methods for importing existing datasets. These may exist on a local computer or on the Internet but the characteristic they share in common is that they are contained (usually) within one single file. The main trick here is to choose the right command to import that data into R. In the second half of the chapter, we will consider a different strategy, namely linking to a "source" of data that is not a file. Many data sources, particularly databases, exist not as a single discrete file, but rather as a system. The system provides methods or calls to "query" data from the system, but from the perspective of the user (and of R) the data never really take the form of a file.

The first and easiest strategy for getting data into R is to use the data import dialog in R-Studio. In the upper right hand pane of R-Studio, the "Workspace" tab gives views of currently available data objects, but also has a set of buttons at the top for managing the work space. One of the choices there is the "Import Dataset" button: This enables a drop down menu where one choice is to import, "From Text File..." If you click this option and choose an appropriate file you will get a screen like this:



The most important stuff is on the left side. Heading controls whether or not the first line of the text file is treated as containing variable names. The separator drop down gives a choice of different characters that separate the fields/columns in the data. R-Studio tries to guess the appropriate choice here based on a scan of the data. In this case it guessed right by choosing "tab-delimited." As mentioned above, tab-delimited and comma-delimited are the two most common formats used for interchange between data programs.The next drop down is for "Decimal" and this option accounts for the fact the a dot is used in the U.S. while a comma may be used for the decimal point in Europe and elsewhere. Finally, the "Quote" drop down controls which character is used to contain quoted string/text data. The most common method is double quotes.

Of course, we skipped ahead a bit here because we assumed that an appropriate file of data was available. It might be useful to see some examples of human readable data:

Name, Age, Gender
"Fred",22,"M"
"Ginger",21,"F"

The above is a very simple example of a comma delimited file where the first row contains a "header," i.e. the information about the names of variables. The second and subsequent rows contain actual data. Each field is separated by a comma, and the text strings are enclosed in double quotes. The same file tab-delimited might look like this:

Name	Age	Gender
"Fred"	22	"M"
"Ginger"	21	"F"

Of course you can't see the tab characters on the screen, but there is one tab character in between each pair of values. In each case, for both comma- and tab-delimited, one line equals one row. The end

of a line is marked, invisibly, with a so called "newline" character. On occasion you may run into differences between different operating systems on how this end of line designation is encoded.

Files containing comma or tab delimited data are ubiquitous across the Internet, but sometimes we would like to gain direct access to binary files in other formats. There are a variety of packages that one might use to access binary data. A comprehensive access list appears here:

http://cran.r-project.org/doc/manuals/R-data.html

This page shows a range of methods for obtaining data from a wide variety of programs and formats. Because Excel is such a widely used program for small, informal data sets, we will use it as an example here to illustrate both the power and the pitfalls of accessing binary data with R. Down near the bottom of the page mentioned just above there are several paragraphs of discussion of how to access Excel files with R. In fact, the first sentence mentions that one of the most commonly asked data questions about R is how to access Excel data.

Interestingly, this is one area where Mac and Linux users are at a disadvantage relative to Windows users. This is perhaps because Excel is a Microsoft product, originally written to be native to Windows, and as a result it is easier to create tools that work with Windows. One example noted here is the package called RODBC. The acronym ODBC stands for Open Database Connection, and this is a Windows facility for exchanging data among Windows programs. Although there is a proprietary ODBC driver available for the Mac, most Mac users will want to try a different method for getting access to Excel data.

Another Windows-only package for R is called xlsReadWrite. This package provides convenient one-command calls for importing data directly from Excel spreadsheets or exporting it directly to spreadsheets. There are also more detailed commands that allow manipulating individual cells.

Two additional packages, xlsx and XLConnect, supposedly will work with the Mac, but at the time of this writing both of these packages had version incompatibilities that made it impossible to install the packages directly into R. Note that the vast majority of packages provide the raw "source code" and so it is theoretically possible, but generally highly time consuming, to "compile" your own copies of these packages to create your own installation.

Fortunately, a general purpose data manipulation package called gdata provides essential facilities for importing spreadsheet files. In the example that follows, we will use a function from gdata to read Excel data directly from a website. The gdata package is a kind of "Swiss Army Knife" package containing many different functions for accessing and manipulating data. For example, you may recall that R uses the value "NA" to represent missing data. Frequently, however, it is the case that data sets contain other values, such as 999, to represent missing data. The gdata package has several functions that find and transform these values to be consistent with R's strategy for handling missing data.

Begin by using install.package() and library() functions to prepare the gdata package for use:

```
> install.packages("gdata")

# ... lots of output here
```

```
> library("gdata")

gdata: read.xls support for 'XLS' (Excel 97-2004)
files

gdata: ENABLED.

gdata: read.xls support for 'XLSX' (Excel 2007+)
files ENABLED.
```

Of course, you could also use the EnsurePackage() function that we developed in an earlier chapter, but it was important here to see the output from the library() function. Note that the gdata package reported some diagnostics about the different versions of Excel data that it supports. Note that this is one of the major drawbacks of binary data formats, particularly proprietary ones: you have to make sure that you have the right software to access the different versions of data that you might encounter. In this case it looks like we are covered for the early versions of Excel (97-2004) as well as later versions of Excel (2007+). We must always be on the lookout, however, for data that is stored in even newer versions of Excel that may not be supported by gdata or other packages.

Now that gdata is installed, we can use the read.xls() function that it provides. The documentation for the gdata package and the read.xls() function is located here:

http://cran.r-project.org/web/packages/gdata/gdata.pdf

A review of the documentation reveals that the only required argument to this function is the location of the XLS file, and that this location can be a pathname, a web location with http, or an Internet location with ftp (file transfer protocol, a way of sending and receiving files without using a web browser). If you hearken back to a very early chapter in this book, you may remember that we accessed some census data that had population counts for all the different U.S. states. For this example, we are going to read the Excel file containing that data directly into a dataframe using the read.xls() function:

```
> testFrame<-read.xls( +
"http://www.census.gov/popest/data/state/totals/2011/
tables/NST-EST2011-01.xls")

trying URL
'http://www.census.gov/popest/data/state/totals/2011/
tables/NST-EST2011-01.xls'

Content type 'application/vnd.ms-excel' length 31232
bytes (30 Kb)

opened URL

==================================================

downloaded 30 Kb
```

The command in the first three lines above provides the URL of the Excel file to the read.xls() function. The subsequent lines of output show the function attempting to open the URL, succeeding, and downloading 30 kilobytes of data.

Next, let's take a look at what we got back. In R-Studio we can click on the name of the dataframe in the upper right hand data pane or we can use this command:

```
> View(testFrame)
```

Either method will show the contents of the dataframe in the upper left hand window of R-Studio. Alternatively, we could use the str() function to create a summary of the structure of testFrame:

```
> str(testFrame)

'data.frame':   65 obs. of  10 variables:

 $
table.with.row.headers.in.column.A.and.column.hea
ders.in.rows.3.through.4...leading.dots.indicate.
sub.parts.: Factor w/ 65 levels "",".Alabama",..:
62 53 1 64 55 54 60 65 2 3 ...

 $ X
: Factor w/ 60 levels "","1,052,567",..: 1 59 60
27 38 47 10 49 32 50 ...

 $ X.1
: Factor w/ 59 levels "","1,052,567",..: 1 1 59
27 38 47 10 49 32 50 ...

 $ X.2
: Factor w/ 60 levels "","1,052,528",..: 1 60 21
28 39 48 10 51 33 50 ...

 $ X.3
: Factor w/ 59 levels "","1,051,302",..: 1 1 21
28 38 48 10 50 33 51 ...

 $ X.4
: logi  NA NA NA NA NA NA ...

 $ X.5
: logi  NA NA NA NA NA NA ...

 $ X.6
: logi  NA NA NA NA NA NA ...

 $ X.7
: logi  NA NA NA NA NA NA ...

 $ X.8
: logi  NA NA NA NA NA NA ...
```

The last few lines are reminiscent of that late 60s song entitled, “"Na Na Hey Hey Kiss Him Goodbye." Setting aside all the NA NA NA NAs, however, the overall structure is 65 observations of 10 variables, signifying that the spreadsheet contained 65 rows and 10 columns of data. The variable names that follow are pretty bizarre. The first variable name is:

“table.with.row.headers.in.column.A.and.column.headers.in.rows.3.through.4...leading.dots.indicate.sub.parts.”

What a mess! It is clear that read.xls() treated the upper leftmost cell as a variable label, but was flummoxed by the fact that this was really just a note to human users of the spreadsheet (the variable labels, such as they are, came on lower rows of the spreadsheet). Subsequent variable names include X, X.1, and X.2: clearly the read.xls() function did not have an easy time getting the variable names out of this file.

The other worrisome finding from str() is that all of our data are “factors.” This indicates that R did not see the incoming data as numbers, but rather as character strings that it interpreted as factor data. Again, this is a side effect of the fact that some of the first cells that read.xls() encountered were text rather than numeric. The numbers came much later in the sheet. This also underscores the

idea that it is much better to export a data set in a more regular, structured format such as CSV rather than in the original spreadsheet format. Clearly we have some work to do if we are to make use of these data as numeric population values.

First, we will use an easy trick to get rid of stuff we don't need. The Census bureau put in three header rows that we can eliminate like this:

```
> testFrame<-testFrame[-1:-3,]
```

The minus sign used inside the square brackets refers to the index of rows that should be eliminated from the dataframe. So the notation -1:-3 gets rid of the first three rows. We also leave the column designator empty so that we can keep all columns for now. So the interpretation of all of the notation within the square brackets is that rows 1 through 3 should be dropped, all other rows should be included, and all columns should be included. We assign the result back to the same data object thereby replacing the original with our new, smaller, cleaner version.

Next, we know that of the ten variables we got from read.xls(), only the first five are useful to us (the last five seem to be blank). So this command keeps the first five columns of the dataframe:

```
> testFrame<-testFrame[,1:5]
```

In the same vein, the tail() function shows us that the last few rows just contained some census bureau notes:

```
> tail(testFrame,5)
```

So we can safely eliminate those like this:

```
> testFrame<-testFrame[-58:-62,]
```

If you're alert you will notice that we could have combined some of these commands, but for the sake of clarity we have done each operation individually. The result (which you can check in the upper right hand pane of R-Studio) is a dataframe with 57 rows and five observations. Now we are ready to perform a couple of data transformations. Before we start these, let's give our first column a more reasonable name:

```
> testFrame$region <- testFrame[,1]
```

We've used a little hack here to avoid typing out the ridiculously long name of that first variable/column. We've used the column notation in the square brackets on the right hand side of the expression to refer to the first column (the one with the ridiculous name) and simply copied the data into a new column entitled "region." Let's also remove the offending column with the stupid name so that it does not cause us problems later on:

```
> testFrame<-testFrame[,-1]
```

Next, we can change formats and data types as needed. We can remove the dots from in front of the state names very easily with str_replace():

```
> testFrame$region <- str_replace( +
                        testFrame$region,"\\.","")
```

Don't forget that str_replace() is part of the stringr package, and you will have to use install.packages() and library() to load it if it is not already in place. The two backslashes in the string expression above are called "escape characters" and they force the dot that follows to be treated as a literal dot rather than as a wildcard charac-

ter. The dot on its own is a wildcard that matches one instance of any character.

Next, we can use str_replace_all() and as.numeric() to convert the data contained in the population columns to usable numbers. Remember that those columns are now represented as R "factors" and what we are doing is taking apart the factor labels (which are basically character strings that look like this: "308,745,538") and making them into numbers. This is sufficiently repetitive that we could probably benefit by created our own function call to do it:

```
# Numberize() - Gets rid of commas and other junk and

# converts to numbers

# Assumes that the inputVector is a list of data that

# can be treated as character strings

Numberize <- function(inputVector)

{

  # Get rid of commas

  inputVector<-str_replace_all(inputVector,",","")

  # Get rid of spaces

  inputVector<-str_replace_all(inputVector," ","")


  return(as.numeric(inputVector))

}
```

This function is flexible in that it will deal with both unwanted commas and spaces, and will convert strings into numbers whether they are integers or not (i.e., possibly with digits after the decimal point). So we can now run this a few times to create new vectors on the dataframe that contain the numeric values we wanted:

```
testFrame$april10census <-Numberize(testFrame$X)

testFrame$april10base <-Numberize(testFrame$X.1)

testFrame$july10pop <-Numberize(testFrame$X.2)

testFrame$july11pop <-Numberize(testFrame$X.3)
```

By the way, the choice of variable names for the new columns in the dataframe was based on an examination of the original data set that was imported by read.xls(). You can (and should) confirm that the new columns on the dataframe are numeric. You can use str() to accomplish this.

We've spent half a chapter so far just looking at one method of importing data from an external file (either on the web or local storage). A lot of our time was spent conditioning the data we got in order to make it usable for later analysis. Herein lies a very important lesson (or perhaps two). An important, and sometimes time consuming aspect of what data scientists do is to make sure that data are "fit for the purpose" to which they are going to be put. We had the convenience of importing a nice data set directly from the web with one simple command, and yet getting those data actually ready to analyze took several additional steps.

A related lesson is that it is important and valuable to try to automate as many of these steps as possible. So when we saw that numbers had gotten stored as factor labels, we moved immediately to create a general function that would convert these to numbers. Not

only does this save a lot of future typing, it prevents mistakes from creeping into our processes.

Now we are ready to consider the other strategy for getting access to data: querying it from external databases. Depending upon your familiarity with computer programming and databases, you may notice that the abstraction is quite a bit different here. Earlier in the chapter we had a file (a rather messy one) that contained a complete copy of the data that we wanted, and we read that file into R and stored it in our local computer's memory (and possibly later on the hard disk for safekeeping). This is a good and reasonable strategy for small to medium sized datasets, let's say just for the sake of argument anything up to 100 megabytes.

But what if the data you want to work with is really large - too large to represent in your computer's memory all at once and too large to store on your own hard drive. This situation could occur even with smaller datasets if the data owner did not want people making complete copies of their data, but rather wanted everyone who was using it to work from one "official" version of the data. Similarly, if data do need to be shared among multiple users, it is much better to have them in a database that was designed for this purpose: For the most part R is a poor choice for maintaining data that must be used simultaneously by more than one user. For these reasons, it becomes necessary to do one or both of the following things:

1. Allow R to send messages to the large, remote database asking for summaries, subsets, or samples of the data.
2. Allow R to send computation requests to a distributed data processing system asking for the results of calculations performed on the large remote database.

Like most contemporary programming languages, R provides several methods for performing these two tasks. The strategy is the same across most of these methods: a package for R provides a "client" that can connect up to the database server. The R client supports sending commands - mostly in SQL, structured query language - to the database server. The database server returns a result to the R client, which places it in an R data object (typically a data frame) for use in further processing or visualization.

The R community has developed a range of client software to enable R to connect up with other databases. Here are the major databases for which R has client software:

- RMySQL - Connects to MySQL, perhaps the most popular open source database in the world. MySQL is the M in "LAMP" which is the acronym for Linux, Apache, MySQL, and PHP. Together, these four elements provide a complete solution for data driven web applications.
- ROracle - Connects with the widely used Oracle commercial database package. Oracle is probably the most widely used commercial database package. Ironically, Oracle acquired Sun Microsystems a few years ago and Sun developers predominate in development and control of the open source MySQL syste,
- RPostgreSQL - Connects with the well-developed, full featured PostgreSQL (sometimes just called Postgres) database system. PostgreSQL is a much more venerable system than MySQL and

has a much larger developer community. Unlike MySQL, which is effectively now controlled by Oracle, PostgreSQL has a developer community that is independent of any company and a licensing scheme that allows anybody to modify and reuse the code.

- RSQlite - Connects with SQlite, another open source, independently developed database system. As the name suggests, SQlite has a very light "code footprint" meaning that it is fast and compact.
- RMongo - Connects with the MongoDB system, which is the only system here that does not use SQL. Instead, MongoDB uses JavaScript to access data. As such it is well suited for web development applications.
- RODBC - Connects with ODBC compliant databases, which include Microsoft's SQLserver, Microsoft Access, and Microsoft Excel, among others. Note that these applications are native to Windows and Windows server, and as such the support for Linux and Mac OS is limited.

For demonstration purposes, we will use RMySQL. This requires installing a copy of MySQL on your computer. Use your web browser to go to this page:

http://dev.mysql.com/downloads/

Then look for the "MySQL Community Server." The term community in this context refers to the free, open source developer community version of MySQL. Note that there are also commercial versions of SQL developed and marketed by various companies including Oracle. Download the version of MySQL Community Server that is most appropriate for your computer's operating system and install it. Note that unlike user applications, such as a word processor, there is no real user interface to server software like the MySQL Community Server. Instead, this software runs in the "background" providing services that other programs can use. This is the essence of the client-server idea. In many cases the server is on some remote computer to which we do not have physical access. In this case, we will run the server on our local computer so that we can complete the demonstration.

On the Mac installation used in preparation of this chapter, after installing the MySQL server software, it was also important to install the "MySQL Preference Pane," in order to provide a simple graphical interface for turning the server on and off. Because we are just doing a demonstration here, and we want to avoid future security problems, it is probably sensible to turn MySQL server off when we are done with the demonstration. In Windows, you can use MySQL Workbench to control the server settings on your local computer.

Returning to R, use install.packages() and library() to prepare the RMySQL package for use. If everything is working the way it should, you should be able to run the following command from the command line:

```
> con <- dbConnect(dbDriver("MySQL"), dbname = "test")
```

The dbConnect() function establishes a linkage or "connection" between R and the database we want to use. This underscores the point that we are connecting to an "external" resource and we must therefore manage the connection. If there were security controls involved, this is where we would provide the necessary infor-

mation to establish that we were authorized users of the database. In this case, because we are on a "local server" of MySQL, we don't need to provide these. The dbDriver() function provided as an argument to dbConnect specifies that we want to use a MySQL client. The database name - specified as dbname="test" - is just a placeholder at this point. We can use the dbListTables() function to see what tables are accessible to us (for our purposes, a table is just like a dataframe, but it is stored inside the database system):

```
> dbListTables(con)
```

```
character(0)
```

The response "character(0)" means that there is an empty list, so no tables are available to us. This is not surprising, because we just installed MySQL and have not used it for anything yet. Unless you have another database available to import into MySQL, we can just use the census data we obtained earlier in the chapter to create a table in MySQL:

```
> dbWriteTable(con, "census", testFrame, +
               overwrite = TRUE)
```

```
[1] TRUE
```

Take note of the arguments supplied to the dbWriteTable() function. The first argument provides the database connection that we established with the dbConnect() function. The "census" argument gives our new table in MySQL a name. We use testFrame as the source of data - as noted above a dataframe and a relational database table are very similar in structure. Finally, we provide the argument overwrite=TRUE, which was not really needed in this case - because we know that there were no existing tables - but could be important in other operations where we need to make sure to replace any old table that may have been left around from previous work. The function returns the logical value TRUE to signal that it was able to finish the request that we made. This is important in programming new functions because we can use the signal of success or failure to guide subsequent steps and provide error or success messages.

Now if we run dbListTables() we should see our new table:

```
> dbListTables(con)
```

```
[1] "census"
```

Now we can run an SQL query on our table:

```
> dbGetQuery(con, "SELECT region, july11pop FROM
census WHERE july11pop<1000000")
```

```
                region july11pop
1               Alaska    722718
2             Delaware    907135
3 District of Columbia    617996
4              Montana    998199
5         North Dakota    683932
6         South Dakota    824082
7              Vermont    626431
8              Wyoming    568158
```

Note that the dbGetQuery() call shown above breaks onto two lines, but the string starting with SELECT has to be typed all on

one line. The capitalized words in that string are the SQL commands. It is beyond the scope of this chapter to give an SQL tutorial, but, briefly, SELECT chooses a subset of the table and the fields named after select are the ones that will appear in the result. The FROM command choose the table(s) where the data should come from. The WHERE command specified a condition, in this case that we only wanted rows where the July 2011 population was less than one million. SQL is a powerful and flexible language and this just scratches the surface.

In this case we did not assign the results of dbGetQuery() to another data object, so the results were just echoed to the R console. But it would be easy to assign the results to a dataframe and then use that dataframe for subsequent calculations or visualizations.

To emphasize a point made above, the normal motivation for accessing data through MySQL or another database system is that a large database exists on a remote server. Rather than having our own complete copy of those data, we can use dbConnect(), dbGetQuery() and other database functions to access the remote data through SQL. We can also use SQL to specify subsets of the data, to preprocess the data with sorts and other operations, and to create summaries of the data. SQL is also particularly well suited to "joining" data from multiple tables to make new combinations. In the present example, we only used one table, it was a very small table, and we had created it ourselves in R from an Excel source, so none of these were very good motivations for storing our data in MySQL, but this was only a demonstration.

The next step beyond remote databases is toward distributed computing across a "cluster" of computers. This combines the remote access to data that we just demonstrated with additional computational capabilities. At this writing, one of the most popular systems for large scale distributed storage and computing is "Hadoop" (named after the toy elephant of the young son of the developer).

Hadoop is not a single thing, but is rather a combination of pieces of software called a library. Hadoop is developed and maintained by the same people who maintain the Apache open source web server. There are about a dozen different parts of the Hadoop framework, but the Hadoop Distributed Files System (HDFS) and Hadoop MapReduce framework are two of the most important frameworks.

HDFS is easy to explain. Imagine your computer and several other computers at your home or workplace. If we could get them all to work together, we could call them a "cluster" and we could theoretically get more use out of them by taking advantage of all of the storage and computing power they have as a group. Running HDFS, we can treat this cluster of computers as one big hard drive. If we have a really large file - too big to fit on any one of the computers - HDFS can divide up the file and store its different parts in different storage areas without us having to worry about the details. With a proper configuration of computer hardware, such as an IT department could supply, HDFS can provide an enormous amount of "throughput" (i.e., a very fast capability for reading and writing data) as well as redundancy and failure tolerance.

MapReduce is a bit more complicated, but it follows the same logic of trying to divide up work across multiple computers. The term MapReduce is used because there are two big processes involved: map and reduce. For the map operation, a big job is broken up into

lots of separate parts. For example, if we wanted to create a search index for all of the files on a company's intranet servers, we could break up the whole indexing task into a bunch of separate jobs. Each job might take care of indexing the files on one server.

In the end, though, we don't want dozens or hundreds of different search indices. We want one big one that covers all of the files our company owns. This is where the reduce operation comes in. As all of the individual indexing jobs finish up, a reduce operation combines them into one big job. This combining process works on the basis of a so called "key." In the search indexing example, some of the small jobs might have found files that contained the word "fish." As each small job finishes, it mentioned whether or not fish appeared in a document and perhaps how many times fish appeared. The reduce operation uses fish as a key to match up the results from all of the different jobs, thus creating an aggregated summary listing all of the documents that contained fish. Later, if anyone searched on the word fish, this list could be used to direct them to documents that contained the word.

In short, "map" takes a process that the user specifies and an indication of which data it applies to, and divides the processing into as many separate chunks as possible. As the results of each chunk become available, "reduce" combines them and eventually creates and returns one aggregated result.

Recently, an organization called RevolutionAnalytics has developed an R interface or "wrapper" for Hadoop that they call RHadoop. This package is still a work in progress in the sense that it does not appear in the standard CRAN package archive, not because there is anything wrong with it, but rather because RevolutionAnalytics wants to continue to develop it without having to provide stable versions for the R community. There is a nice tutorial here:

https://github.com/RevolutionAnalytics/RHadoop/wiki/Tutorial

We will break open the first example presented in the tutorial just to provide further illustration of the use of MapReduce. As with our MySQL example, this is a rather trivial activity that would not normally require the use of a large cluster of computers, but it does show how MapReduce can be put to use.

The tutorial example first demonstrates how a repetitive operation is accomplished in R without the use of MapReduce. In prior chapters we have used several variations of the apply() function. The lapply() or list-apply is one of the simplest. You provide an input vector of values and a function to apply to each element, and the lapply() function does the heavy lifting. The example in the RHadoop tutorial squares each integer from one to 10. This first command fills a vector with the input data:

```
> small.ints <- 1:10

> small.ints

 [1]  1  2  3  4  5  6  7  8  9 10
```

Next we can apply the "squaring function" (basically just using the ^ operator) to each element of the list:

```
> out <- lapply(small.ints, function(x) x^2)

> out
```

```
[[1]]

[1] 1


[[2]]

[1] 4

... (shows all of the values up to [[10]] [1] 100)
```

In the first command above, we have used lapply() to perform a function on the input vector small.ints. We have defined the function as taking the value x and returning the value x^2. The result is a list of ten vectors (each with just one element) containing the squares of the input values. Because this is such a small problem, R was able to accomplish it in a tiny fraction of a second.

After installing both Hadoop and RHadoop - which, again, is not an official package, and therefore has to be installed manually - we can perform this same operation with two commands:

```
> small.ints <- to.dfs(1:10)

> out <- mapreduce(input = small.ints, +
             map = function(k,v) keyval(v, v^2))
```

In the first command, we again create a list of integers from one to ten. But rather than simply storing them in a vector, we are using the "distributed file system" or dfs class that is provided by RHadoop. Note that in most cases we would not need to create this ourselves because our large dataset would already exist on the HDFS (Hadoop Distributed FIle System). We would have connected to HDFS and selected the necessary data much as we did earlier in this chapter with dbConnect().

In the second command, we are doing essentially the same thing as we did with lapply(). We provide the input data structure (which, again is a dfs class data object, a kind of pointer to the data stored by Hadoop in the cluster). We also provide a "map function" which is the process that we want to apply to each element in our data set. Notice that the function takes two arguments, k and v. The k refers to the "key" that we mentioned earlier in the chapter. We actually don't need the key in this example because we are not supplying a reduce function. There is in fact, no aggregation or combining activity that needs to occur because our input list (the integers) and the output list (the squares of those integers) are lists of the same size. If we had needed to aggregate the results of the map function, say by creating a mean or a sum, we would have had to provide a "reduce function" that would do the job.

The keyval() function, for which there is no documentation at this writing, is characterized as a "helper" function in the tutorial. In this case it is clear that the first argument to keyval, "v" is the intger to which the process must be applied, and the second argument, "v^2" is the squaring function that is applied to each argument. The data returned by mapreduce() is functionally equivalent to that returned by lapply(), i.e., a list of the squares of the integers from 1 to 10.

Obviously there is no point in harnessing the power of a cluster of computers to calculate something that could be done with a pencil and a paper in a few seconds. If, however, the operation was more complex and the list of input data had millions of elements, the use

of lapply() would be impractical as it would take your computer quite a long time to finish the job. On the other hand, the second strategy of using mapreduce() could run the job in a fraction of a second, given a sufficient supply of computers and storage.

On a related note, Amazon, the giant online retailer, provides virtual computer clusters that can be used for exactly this kind of work. Amazon's product is called the Elastic Compute Cloud or EC2, and at this writing it is possible to create a small cluster of Linux computers for as little as eight cents per hour.

To summarize this chapter, although there are many analytical problems that require only a small amount of data, the wide availability of larger data sets has added new challenges to data science. As a single user program running on a local computer, R is well suited for work by a single analyst on a data set that is small enough to fit into the computer's memory. We can retrieve these small datasets from individual files stored in human readable 9e.g., CSV) or binary (e.g., XLS) formats.

To be able to tackle the larger data sets, however, we need to be able to connect R with either remote databases or remote computational resources or both. A variety of packages is available to connect R to mature database technologies such as MySQL. In fact, we demonstrated the use of MySQL by installing it on a local machine and then using the RMySQL package to create a table and query it. The more cutting edge technology of Hadoop is just becoming available for R users. This technology, which provides the potential for both massive storage and parallel computational power, promises to make very large datasets available for processing and analysis in R.

Chapter Challenge

Hadoop is a software framework designed for use with Apache, which is first and foremost a Linux server application. Yet there are development versions of Hadoop available for Windows and Mac as well. These are what are called single node instances, that is they use a single computer to simulate the existence of a large cluster of computers. See if you can install the appropriate version of Hadoop for your computer's operating system.

As a bonus activity, if you are successful in installing Hadoop, then get a copy of the RHadoop package from RevolutionAnalytics and install that. If you are successful with both, you should be able to run the MapReduce code presented in this chapter.

Sources

http://cran.r-project.org/doc/manuals/R-data.html

http://cran.r-project.org/doc/manuals/R-data.pdf

http://cran.r-project.org/web/packages/gdata/gdata.pdf

http://dev.mysql.com/downloads/

http://en.wikipedia.org/wiki/Comparison_of_relational_database_management_systems

http://en.wikipedia.org/wiki/Mapreduce

https://github.com/RevolutionAnalytics/RHadoop/wiki/Tutorial

R Functions Used in this Chapter

as.numeric() - Convert another data type to a number

dbConnect() - Connect to an SQL database

dgGetQuery() - Run an SQL query and return the results

dbListTables() - Show the tables available in a connection

dbWriteTable() - Send a data table to an SQL systems

install.packages() - Get the code for an R package

lapply() - Apply a function to elements of a list

library() - Make an R package available for use

Numberize() - A custom function created in this chapter

read.xls() - Import data from a binary R file; part of the gdata package

return() - Used in functions to designate the data returned by the function

str_replace() - Replace a character string with another

str_replace_all() - Replace multiple instances of a character string with another

CHAPTER 15

Map MashUp

Much of what we have accomplished so far has focused on the standard rectangular dataset: one neat table with rows and columns well defined. Yet much of the power of data science comes from bringing together difference sources of data in complementary ways. In this chapter we combine different sources of data to make a unique product that transcends any one source.

Mashup is a term that originated in the music business decades ago related to the practice of overlaying one music recording on top of another one. The term has entered general usage to mean anything that brings together disparate influences or elements. In the application development area, mashup often refers to bringing together various sources of data to create a new product with unique value. There's even a non-profit group called the Open Mashup Alliance that develops standards and methods for creating new mashups.

One example of a mashup is http://www.housingmaps.com/ , a web application that grabs apartment rental listings from the classified advertising service Craigslist and plots them on an interactive map that shows the location of each listing. If you have ever used Craigslist you know that it provides a very basic text-based interface and that the capability to find listings on a map would be welcome.

In this chapter we tackle a similar problem. Using some address data from government records, we call the Google geocoding API over the web to find the latitude and longitude of each address. Then we plot these latitudes and longitudes on a map of the U.S. This activities reuses skills we learned in the previous chapter for reading in data files, adds some new skills related to calling web APIs, and introduces us to a new type of data, namely the shapefiles that provide the basis for electronic maps.

Let's look at the new stuff first. The Internet is full of shapefiles that contain mapping data. Shapefiles are a partially proprietary, partially open format supported by a California software company called ESRI. Shapefile is actually an umbrella term that covers several different file types. Because the R community has provided some packages to help deal with shapefiles, we don't need to much information about the details. The most important thing to know is that shapefiles contain points, polygons, and "polylines." Everyone knows what a point and a polygon are, but a polyline is a term used by computer scientist to refer to a multi-segment line. In many graphics applications it is much easier to approximate a curved line with many tiny connected straight lines than it is to draw a truly curved line. If you think of a road or a river on a map, you will have a good idea of a polyline.

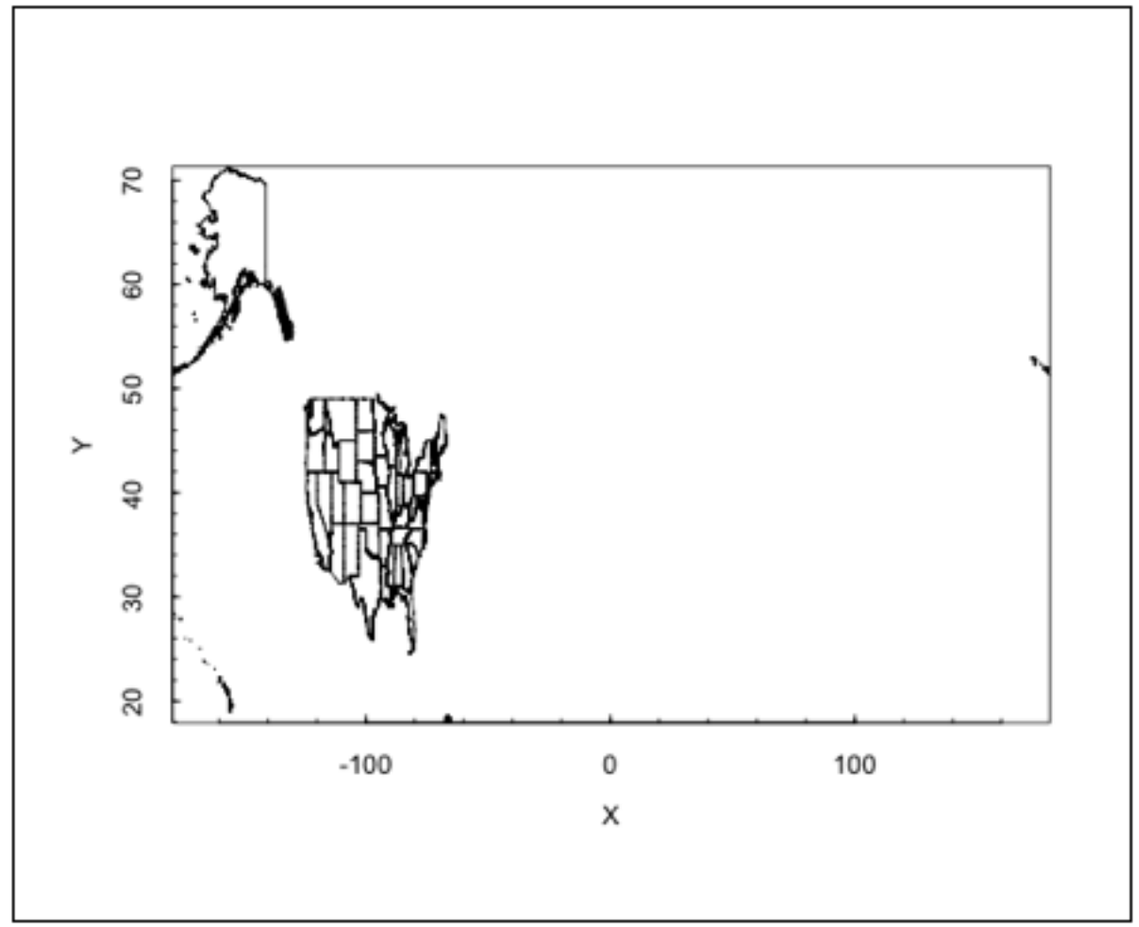
The U.S. Census bureau publishes shapefiles at various levels of detail for every part of the country. Search for the term "shapefile" at "site:census.gov" and you will find several pages with listings of different shapefiles. For this exercise, we are using a relatively low detail map of the whole U.S. We downloaded a "zip" file. Unzip this (usually just by double-clicking on it) and you will find several files inside it. The file ending in "shp" is the main shapefile. Another file that will be useful to us ends in "dbf" - this contains labels and other information.

To get started, we will need two new R packages called PBSmapping and maptools. PBSmapping refers not to public broadcasting, but rather to the Pacific Biology Station, whose researchers and technologists kindly bundled up a wide range of the R processing tools that they use to manage map data. The maptools package was developed by Nicholas J. Lewin-Koh (University of Singapore) and others to provide additional tools and some "wrappers" to make PBSmapping functions easier to use. In this chapter we only scratch the surface of the available tools: there could be a whole book just dedicated to R mapping functions alone.

Before we read in the data we grabbed from the Census Bureau, let's set the working directory in R-Studio so that we don't have to type it out on the command line. Click on the Tools menu and then choose "Set Working Directory." Use the dialog box to designate the folder where you have unzipped the shape data. After that, these commands will load the shape data into R and show us what we have:

```
> usShape <- importShapefile( +
          "gz_2010_us_040_00_500k",readDBF=TRUE)

> summary(usShape)

PolySet

Records          : 90696

  Contours       : 574

    Holes        : 0

  Events         : NA

    On boundary  : NA

Ranges

  X              : [-179.14734, 179.77847]

  Y              : [17.884813, 71.3525606439998]

Attributes

  Projection     : LL

  Zone           : NULL

Extra columns    :

> plotPolys(usShape)
```

This last command gives us a simple plot of the 90,696 shapes that our shapefile contains. Here is the plot:



This is funny looking! The ranges output from the summary() command gives us a hint as to why. The longitude of the elements in our map range from -179 to nearly 180: this covers pretty much the whole of the planet. The reason is that the map contains shapes for Hawaii and Alaska. Both states have far western longitudes, but the Aleutian peninsula in Alaska extends so far that it crosses over the longitude line where -180 and 180 meet in the Pacific Ocean. As

a result, the continental U.S. is super squashed. We can specify a more limited area of the map to consider by using the xlim and ylim parameters. The following command:

```
> plotPolys(usShape,+
                  xlim=c(-130,-60),ylim=c(20,50))
```

...gives a plot that shows the continental U.S. more in its typical proportions.

So now we have some map data stored away and ready to use. The PBSmapping package gives us the capability of adding points to an existing map. For now, let's demonstrate this with a made up point somewhere in Texas:

```
> X <- -100

> Y <- 30

> EID <- 1

> pointData <- data.frame(EID,X,Y)

> eventData <- as.EventData( +
                          pointData,projection=NA)

> addPoints(eventData,col="red",cex=.5)
```

You have to look carefully, but in southern Texas there is now a little red dot. We began by manually creating a single point - specified by X (longitude), Y (latitude), and EID (an identification num-

ber) - and sticking it into a dataframe. Then we converted the data in that dataframe into an EventData object. This is a custom class of object specified by the PBSmapping package. The final command above adds the EventData to the plot.

The idea of EventData is a little confusing, but if you remember that this package was developed by biologists at the Pacific Biology Station to map sightings of fish and other natural phenomena it makes more sense. In their lingo, an event was some observation of interest that occurred at a particular day, time, and location. The "event id" or EID <- 1 that we stuck in the data frame was just saying that this was the first point in our list that we wanted to plot. For us it is not an event so much as a location of something we wanted to see on the map.

Also note that the "projection=NA" parameter in the as.EventData() coercion is just letting the mapping software know that we don't want our point to be transformed according to a mapping projection. If you remember from your Geography class, a projection is a mapping technique to make a curved object like the Earth seem sensible on a flat map. In this example, we've already flattened out the U.S., so there is no need to transform the points.

Next, we need a source of points to add to our map. This could be anything that we're interested in: the locations of restaurants, crime scenes, colleges, etc. In Google a search for filetype:xls or filetype;csv with appropriate additional search terms can provide interesting data sources. You may also have mailing lists of customers or clients. The most important thing is that we will need street address, city, and state in order to geocode the addresses. For this example, we searched for "housing street address list filetype:csv" and this turned up a data set of small businesses that have contracts with the U.S. Department of Health and Human services. Let's read this in using read.csv():

```
> dhhsAddrs <- read.csv("DHHS_Contracts.csv")

> str(dhhsAddrs)

'data.frame':   599 obs. of  10 variables:

 $ Contract.Number           : Factor w/ 285 lev-
els "26301D0054","500000049",..: 125 125 125 279
164 247 19 242 275 70 ...

 $ Contractor.Name           : Factor w/ 245 lev-
els "2020 COMPANY LIMITED LIABILITY COMPANY",..:
1 1 1 2 2 3 4 6 5 7 ...

 $ Contractor.Address        : Factor w/ 244 lev-
els "1 CHASE SQUARE 10TH FLR, ROCHESTER, NY ",..:
116 116 116 117 117 136 230 194 64 164 ...

 $ Description.Of.Requirement: Factor w/ 468 lev-
els "9TH SOW - DEFINTIZE THE LETTER CONTRACT",..:
55 55 55 292 172 354 308 157 221 340 ...

 $ Dollars.Obligated         : Factor w/ 586 lev-
els " $1,000,000.00 ",..: 342 561 335 314 294 2
250 275 421 21 ...

 $ NAICS.Code                : Factor w/ 55 lev-
els "323119","334310",..: 26 26 26 25 10 38 33 29
27 35 ...

 $ Ultimate.Completion.Date  : Factor w/ 206 lev-
els "1-Aug-2011","1-Feb-2013",..: 149 149 149 10
175 161 124 37 150 91 ...
```

```
 $ Contract.Specialist       : Factor w/ 95 lev-
els "ALAN  FREDERICKS",..: 14 14 60 54 16 90 55
25 58 57 ...

 $ Contract.Specialist.Email : Factor w/ 102 lev-
els "410-786-8622",..: 16 16 64 59 40 98 60 29 62
62 ...

 $ X               : logi  NA NA NA NA NA NA ...
```

There's that crazy 60s song again! Anyhow, we read in a comma separated data set with 599 rows and 10 variables. The most important field we have there is Contractor.Address. This contains the street addresses that we need to geocode. We note, however, that the data type for these is Factor rather than character string. So we need to convert that:

```
> dhhsAddrs$strAddr <- +
      as.character(dhhsAddrs$Contractor.Address)

> mode(dhhsAddrs$strAddr)

[1] "character"

> tail(dhhsAddrs$strAddr,4)

[1] "1717 W BROADWAY, MADISON, WI "

[2] "1717 W BROADWAY, MADISON, WI "

[3] "1717 W BROADWAY, MADISON, WI "

[4] "789 HOWARD AVE, NEW HAVEN, CT, "
```

This looks pretty good: Our new column, called dhhsAddrs, is character string data, converted from the factor labels of the original Contractor.Address column. Now we need to geocode these.

We will use the Google geocoding application programming interface (API) which is pretty easy to use, does not require an account or application ID, and allows about 2500 address conversions per day. The API can be accessed over the web, using what is called an HTTP GET request.

These acronyms probably look familiar. HTTP is the Hyper Text Transfer Protocol, and it is the standard method for requesting and receiving web page data. A GET request consists of information that is included in the URL string to specify some details about the information we are hoping to get back from the request. Here is an example GET request to the Google geocoding API:

http://maps.googleapis.com/maps/api/geocode/json?address=1600+Pennsylvania+Avenue,+Washington,+DC&sensor=false

The first part of this should look familiar: The http://maps.googleapis.com part of the URL specifies the domain name just like a regular web page. The next part of the URL, "/maps/api/geocode" tells Google which API we want to use. Then the "json" indicates that we would like to receive our result in "Java Script Object Notation" (JSON) which is a structured, but human readable way of sending back some data. The address appears next, and we are apparently looking for the White House at 1600 Pennsylvania Avenue in Washington, DC. Finally, sensor=false is a required parameter indicating that we are not sending our request from a mobile phone. You can type that whole URL into the address field of any web browser and you should get a sensible result back. The JSON notation is not beautiful, but you will see that it makes sense and provides the names of individual data items along with their values. Here's a small excerpt that shows

the key parts of the data object that we are trying to get our hands on:

```
{
   "results" : [
      {
         "address_components" : [
         "geometry" : {
            "location" : {
               "lat" : 38.89788009999999,
               "lng" : -77.03664780
            },
   "status" : "OK"
}
```

There's tons more data in the JSON object that Google returned, and fortunately there is an R package, called JSONIO, that will extract the data we need from the structure without having to parse it ourselves.

In order to get R to send the HTTP GET requests to google, we will also need to use the RCurl package. This will give us a single command to send the request and receive the result back - essentially doing all of the quirky steps that a web browser takes care of automatically for us. To get started, install.packages() and library() the two packages we will need - RCurl and JSONIO. If you are working on a Windows machine, you may need to jump through a hoop or two to get RCurl, but it is available for Windows even if it is not in the standard CRAN repository. Search for "RCurl Windows" if you run into trouble.

Next, we will create a new helper function to take the address field and turn it into the URL that we need:

```
# Format an URL for the Google Geocode API

MakeGeoURL <- function(address)

{

  root <- "http://maps.google.com/maps/api/geocode/"

  url <- paste(root, "json?address=", +
               address, "&sensor=false", sep = "")

  return(URLencode(url))

}
```

There are three simple steps here. The first line initializes the beginning part of the URL into a string called root. Then we use paste() to glue together the separate parts of the strong (note the sep="" so we don't get spaces between the parts). This creates a string that looks almost like the one in the White House example two pages ago. The final step converts the string to a legal URL using a utility function called URLencode() that RCurl provides. Let's try it:

```
> MakeGeoURL( +
"1600 Pennsylvania Avenue, Washington, DC")

[1]
"http://maps.google.com/maps/api/geocode/json?add
ress=1600%20Pennsylvania%20Avenue,%20Washington,%
20DC&sensor=false"
```

Looks good! Just slightly different than the original example (%20 instead of the plus character) but hopefully that won't make a difference. Remember that you can type this function at the command line or you can create it in the script editing window in the upper left hand pane of R-Studio. The latter is the better way to go and if you click the "Source on Save" checkmark, R-Studio will make sure to update R's stored version of your function every time you save the script file.

Now we are ready to use our new function, MakeGeoURL(), in another function that will actually request the data from the Google API:

```
Addr2latlng <- function(address)

{

  url <- MakeGeoURL(address)

  apiResult <- getURL(url)

  geoStruct <- fromJSON(apiResult, +
                                simplify = FALSE)

  lat <- NA

  lng <- NA

  try(lat <- +
    geoStruct$results[[1]]$geometry$location$lat)

  try(lng <- +
    geoStruct$results[[1]]$geometry$location$lng)

  return(c(lat, lng))

}
```

We have defined this function to receive an address string as its only argument. The first thing it does is to pass the URL string to MakeGeoURL() to develop the formatted URL. Then the function passes the URL to getURL(), which actually does the work of sending the request out onto the Internet. The getURL() function is part of the RCurl package. This step is just like typing a URL into the address box of your browser.

We capture the result in an object called "apiResult". If we were to stop and look inside this, we would find the JSON structure that appeared a couple of pages ago. We can pass this structure to the function fromJSON() - we put the result in an object called geoStruct. This is a regular R data frame such that we can access any individual element using regular $ notation and the array index [[1]]. In other instances, a JSON object may contain a whole list of data structures, but in this case there is just one. If you compare the variable names "geometry", "location", "lat" and "lng" to the JSON example a few pages ago you will find that they match perfectly. The fromJSON() function in the JSONIO package has done all the heavy lifting of breaking the JSON structure into its component pieces.

Note that this is the first time we have encountered the try() function. When programmers expect the possibility of an error, they will frequently use methods that are tolerant of errors or that catch errors before they disrupt the code. If our call to getURL() returns something unusual that we aren't expecting, then the JSON structure may not contain the fields that we want. By surrounding our command to assign the lat and lng variables with a try() function, we can avoid stopping the flow of the code if there is an error. Because we initialized lat and lng to NA above, this function will re-

turn a two item list with both items being NA if an error occurs in accessing the JSON structure. There are more elegant ways to accomplish this same goal. For example, the Google API puts an error code in the JSON structure and we could choose to interpret that instead. We will leave that to the chapter challenge!

In the last step, our new Addr2latlng() function returns a two item list containing the latitude and longitude. We can test it out right now:

```
> testData <- Addr2latlng( +
      "1600 Pennsylvania Avenue, Washington, DC")

> str(testData)

 num [1:2] 38.9 -77
```

Perfect! we called our new function Addr2latlng() with the address of the White House and got back a list with two numeric items containing the latitude and longitude associate with that address. With just a few lines of R code we have harnessed the power of Google's extensive geocoding capability to convert a brief text street address into mapping coordinates.

At this point there isn't too much left to do. We have to create a looping mechanism so that we can process the whole list of addresses in our DHHS data set. We have some small design choices here. It would be possible to attach new fields to the existing dataframe. Instead, the following code keeps everything pretty simple by receiving a list of addresses and returning a new data frame with X, Y, and EID ready to feed into our mapping software:

```
# Process a whole list of addresses

ProcessAddrList <- function(addrList)

{

  resultDF <- data.frame(atext=character(), +
              X=numeric(),Y=numeric(),EID=numeric())

  i <- 1

  for (addr in addrList)

  {

    latlng = Addr2latlng(addr)

    resultDF <- rbind(resultDF,+
                data.frame(atext=addr, +
                X=latlng[[2]],Y=latlng[[1]], EID=i))

    i <- i + 1

  }

  return(resultDF)

}
```

This new function takes one argument, the list of addresses, which should be character strings. In the first step we make an empty dataframe for use in the loop. In the second step we initialize a scalar variable called i to the number one. We will increment this in the loop and use it as our EID.

Then we have the for loop. We are using a neat construct here called "in". The expression "addr in addrList" creates a new variable called addr. Every time that R goes through the loop it assigns to addr the next item in addrList. When addrList runs out of items, the for loop ends. Very handy!

Inside the for loop the first thing we do is to call the function that we developed earlier: Addr2latlng(). This performs one conversion of an address to a geocode (latitude and longitude) as described earlier. We pass addr to it as add contains the text address for this time around the loop. We put the result in a new variable called latlng. Remember that this is a two item list.

The next statement, starting with "resultDF <- rbind" is the heart of this function. Recall that rbind() sticks a new row on the end of a dataframe. So in the arguments to rbind() we supply our earlier version of resultDF (which starts empty and grows from there) plus a new row of data. The new row of data includes the text address (not strictly necessary but handy for diagnostic purposes), the "X" that our mapping software expects (this is the longitude), the "Y" that the mapping software expects, and the event ID, EID, that the mapping software expects.

At the end of the for loop, we increment i so that we will have the next number next time around for EID. Once the for loop is done we simply return the dataframe object, resultDF. Piece of cake!

Now let's try it and plot our points:

```
> dhhsPoints <- ProcessAddrList(dhhsAddrs$strAddr)

> dhhsPoints <- dhhsPoints[!is.na(dhhsPoints$X),]

> eventData <- as.EventData(dhhsPoints,projection=NA)

> addPoints(eventData,col="red",cex=.5)
```

The second command above is the only one of the four that may seem unfamiliar. The as.EventData() coercion is picky and will not process the dataframe if there are any fields that are NA. To get rid of those rows that do not have complete latitude and longitude data, we use is.na() to test whether the X value on a given row is NA. We use the ! (not) operator to reverse the sense of this test. So the only rows that will survive this step are those where X is not NA. The plot below shows the results.

If you like conspiracy theories, there is some support in this graph: The vast majority of the companies that have contracts with DHHS are in the Washington, DC area, with a little trickle of additional companies heading up the eastern seaboard as far as Boston and southern New Hampshire. Elsewhere in the country there are a few companies here and there, particularly near the large cities of the east and south east.

Using some of the parameters on plotPolys() you could adjust the zoom level and look at different parts of the country in more detail. If you remember that the original DHHS data also had the monetary size of the contract in it, it would also be interesting to change the size or color of the dots depending on the amount of money in the Dollars.Obligated field. This would require running addPoints() in a loop and setting the col= or cex= parameters for each new point.

Chapter Challenge(s)

Improve the Addr2latlng() function so that it checks for errors returned from the Google API. This will require going to the Google API site, looking more closely at the JSON structure, and reviewing the error codes that Google mentions in the documentation. You may also learn enough that you can repair some of the addresses that failed.

If you get that far and are still itching for another challenge, try improving the map. One idea for this was mentioned above: you could change the size or color of the dots based on the size of the contract received. An even better (and much harder) idea would be to sum the total dollar amount being given within each state and then color each state according to how much DHHS money it receives. This would require delving into the shapefile data quite substantially so that you could understand how to find the outline of a state and fill it in with a color.

R Functions Used in This Chapter

Addr2latlng() - A custom function built for this chapter

addPoints() - Place more points on an existing plot

as.character() - Coerces data into a character string

as.EventData() - Coerce a regular dataframe into an EventData object for use with PBSmapping routines.

data.frame() - Takes individual variables and ties them together

for() - Runs a loop, iterating a certain number of times depending upon the expression provided in parentheses

fromJSON() - Takes JSON data as input and provides a regular R dataframe as output

function() - Creates a new function

importShapeFile() - Gets shapefile data from a set of ESRI compatible polygon data files

MakeGeoURL() - Custom helper function built for this chapter

paste() - Glues strings together

plotPolys() - Plots a map from shape data

rbind() - Binds new rows on a dataframe

return() - Specifies the object that should be returned from a function

URLencode() - Formats a character string so it can be used in an HTTP request

Sources

http://en.wikipedia.org/wiki/Open_Mashup_Alliance

http://en.wikipedia.org/wiki/Shapefile

http://www.housingmaps.com/

http://www.census.gov/geo/www/cob/cbf_state.html