Tony Fischetti, Eric Mayor
Rui Miguel

R: Predictive Analysis

Learning Path

Master the art of predictive modeling

Packt>

R: Predictive Analysis

Master the art of predictive modeling

A course in three modules

BIRMINGHAM - MUMBAI

R: Predictive Analysis

Published on: March 2017

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

ISBN 978-1-78829-037-1

www.packtpub.com

Credits

Authors

Tony Fischetti

Eric Mayor

Rui Miguel Forte

Reviewers

Dipanjan Sarkar

Ajay Dhamija

Khaled Tannir

Matt Wiley

Prasad Kothari

Dawit Gezahegn Tadesse

Content Development Editor

Mayur Pawanikar

Production Coordinator

Nilesh Mohite

Preface

Frequently the tool of choice for academics, R has spread deep into the private sector and can be found in the production pipelines at some of the most advanced and successful enterprises. The power and domain-specificity of R allows the user to express complex analytics easily, quickly, and succinctly. With over 7,000 user contributed packages, it's easy to find support for the latest and greatest algorithms and techniques.

Packed with engaging problems and exercises, this course begins with a review of R and its syntax. From there, get to grips with the fundamentals of applied statistics and build on this knowledge to perform sophisticated and powerful analytics. Solve the difficulties relating to performing data analysis in practice and find solutions to working with "messy data", large data, communicating results, and facilitating reproducibility.

The primary mission of this course is to bridge the gap between low-level introductory books and tutorials that emphasize intuition and practice over theory, and high-level academic texts that focus on mathematics, detail, and rigor. Another equally important goal is to instill some good practices in you, such as learning how to properly test and evaluate a model. We also emphasize important concepts, such as the bias-variance trade-off and over-fitting, which are pervasive in predictive modeling and come up time and again in various guises and across different models.

This Learning Path combines some of the best that Packt has to offer in one complete, curated package. It includes content from the following Packt products:

- Data Analysis with R
- Learning Predictive Analytics with R
- Mastering Predictive Analytics with R

What this learning path covers

Module 1, Starting with the basics of R and statistical reasoning, Data Analysis with R dives into advanced predictive analytics, showing how to apply those techniques to real-world data though with real-world examples. This course is engineered to be an invaluable resource through many stages of anyone's career as a data analyst.

Module 2, The main purpose of this book is to show you how to analyze data with reasonably simple algorithms. The book is composed of chapters describing the algorithms and their use and of an appendices with exercises and solutions to the exercises and references.

Module 3, The purpose of this course is to show how to use R tools/packages for applied predictive analytics. The course will make full use of R for Predictive models so that by the end of the course, the readers would have gained expertise in building predictive models and performing Predictive Analytics with R.

What you need for this learning path

Module 1:

All code in this book has been written against the latest version of R—3.2.2 at the time of writing. As a matter of good practice, you should keep your R version up to date but most, if not all, code should work with any reasonably recent version of R. Some of the R packages we will be installing will require more recent versions, though. For the other software that this book uses, instructions will be furnished pro re nata. If you want to get a head start, however, install RStudio, JAGS, and a C++ compiler (or Rtools if you use Windows).

Module 2:

All you need for this book is a working installation of R > 3.0 (on any operating system) and an active internet connection.

Following are the links for your reference:

Installing R: https://cran.r-project.org/doc/manuals/r-release/R-admin.html

R Interpreter for Apache Zeppelin: https://zeppelin.apache.org/docs/0.6.0/interpreter/r.html

Module 3:

The only strong requirement for running the code in this book is an installation of R. This is freely available from `http://www.r-project.org/` and runs on all the major operating systems. The code in this book has been tested with R version 3.1.3.

All the chapters introduce at least one new R package that does not come with the base installation of R. We do not explicitly show the installation of R packages in the text, but if a package is not currently installed on your system or if it requires updating, you can install it with the `install.packages()` function. For example, the following command installs the tm package:

```
> install.packages("tm")
```

All the packages we use are available on CRAN. An Internet connection is needed to download and install them as well as to obtain the open source data sets that we use in our real-world examples. Finally, even though not absolutely mandatory, we recommend that you get into the habit of using an Integrated Development Environment (IDE) to work with R. An excellent offering is RStudio (`http://www.rstudio.com/`), which is open source.

Who this learning path is for

If you work with data and want to become an expert in predictive analysis and modeling, then this Learning Path will serve you well. It is intended for budding and seasoned practitioners of predictive modeling alike. You should have basic knowledge of the use of R, although it's not necessary to put this Learning Path to great use.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this course—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the course's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a course, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt course, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this course from your account at `http://www.packtpub.com`. If you purchased this course elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the course in the **Search** box.
5. Select the course for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this course from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the course's webpage at the Packt Publishing website. This page can be accessed by entering the course's name in the **Search** box. Please note that you need to be logged in to your Packt account.

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

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

The code bundle for the course is also hosted on GitHub at `https://github.com/PacktPublishing/R-Predictive-Analysis`. We also have other code bundles from our rich catalog of books, videos and courses available at `https://github.com/PacktPublishing/`. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this course. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your course, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this course, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

Module 1

Module 2

Module 3

Module 1

Data Analysis with R

Load, wrangle, and analyze your data using the world's most powerful statistical programming language

1
RefresheR

Before we dive into the (other) fun stuff (sampling multi-dimensional probability distributions, using convex optimization to fit data models, and so on), it would be helpful if we review those aspects of R that all subsequent chapters will assume knowledge of.

If you fancy yourself as an R guru, you should still, at least, skim through this chapter, because you'll almost certainly find the idioms, packages, and style introduced here to be beneficial in following along with the rest of the material.

If you don't care much about R (yet), and are just in this for the statistics, you can heave a heavy sigh of relief that, for the most part, you can run the code given in this book in the interactive R interpreter with very little modification, and just follow along with the ideas. However, it is my belief (read: delusion) that by the end of this book, you'll cultivate a newfound appreciation of R alongside a robust understanding of methods in data analysis.

Fire up your R interpreter, and let's get started!

Navigating the basics

In the interactive R interpreter, any line starting with a `>` character denotes R asking for input (If you see a `+` prompt, it means that you didn't finish typing a statement at the prompt and R is asking you to provide the rest of the expression.). Striking the *return* key will send your input to R to be evaluated. R's response is then spit back at you in the line immediately following your input, after which R asks for more input. This is called a **REPL** (**Read-Evaluate-Print-Loop**). It is also possible for R to read a batch of commands saved in a file (unsurprisingly called *batch mode*), but we'll be using the interactive mode for most of the book.

As you might imagine, R supports all the familiar mathematical operators as most other languages:

Arithmetic and assignment

Check out the following example:

```
> 2 + 2
[1] 4

> 9 / 3
[1] 3

> 5 %% 2    # modulus operator (remainder of 5 divided by 2)
[1] 1
```

Anything that occurs after the octothorpe or pound sign, `#`, (or *hash-tag* for you young'uns), is ignored by the R interpreter. This is useful for documenting the code in natural language. These are called *comments*.

In a multi-operation arithmetic expression, R will follow the standard order of operations from math. In order to override this natural order, you have to use parentheses flanking the sub-expression that you'd like to be performed first.

```
> 3 + 2 - 10 ^ 2        # ^ is the exponent operator
[1] -95
> 3 + (2 - 10) ^ 2
[1] 67
```

In practice, almost all compound expressions are split up with intermediate values assigned to variables which, when used in future expressions, are just like substituting the variable with the value that was assigned to it. The (primary) assignment operator is `<-`.

```
> # assignments follow the form VARIABLE <- VALUE
> var <- 10
> var
[1] 10
> var ^ 2
[1] 100
> VAR / 2             # variable names are case-sensitive
Error: object 'VAR' not found
```

Notice that the first and second lines in the preceding code snippet didn't have an output to be displayed, so R just immediately asked for more input. This is because assignments don't have a return value. Their only job is to give a value to a variable, or to change the existing value of a variable. Generally, operations and functions on variables in R don't change the value of the variable. Instead, they return the result of the operation. If you want to change a variable to the result of an operation using that variable, you have to reassign that variable as follows:

```
> var               # var is 10
[1] 10
> var ^ 2
[1] 100
> var               # var is still 10
[1] 10
> var <- var ^ 2    # no return value
> var               # var is now 100
[1] 100
```

Be aware that variable names may contain numbers, underscores, and periods; this is something that trips up a lot of people who are familiar with other programming languages that disallow using periods in variable names. The only further restrictions on variable names are that it must start with a letter (or a period and then a letter), and that it must not be one of the reserved words in R such as **TRUE**, **Inf**, and so on.

Although the arithmetic operators that we've seen thus far are functions in their own right, most functions in R take the form: `function_name` (value(s) supplied to the function). The values supplied to the function are called *arguments* of that function.

```
> cos(3.14159)      # cosine function
[1] -1
> cos(pi)           # pi is a constant that R provides
[1] -1
> acos(-1)          # arccosine function
[1] 2.141593
> acos(cos(pi)) + 10
[1] 13.14159
> # functions can be used as arguments to other functions
```

(If you paid attention in math class, you'll know that the cosine of π is -1, and that arccosine is the inverse function of cosine.)

There are hundreds of such useful functions defined in base R, only a handful of which we will see in this book. Two sections from now, we will be building our very own functions.

Before we move on from arithmetic, it will serve us well to visit some of the odd values that may result from certain operations:

```
> 1 / 0
[1] Inf
>  0 / 0
[1] NaN
```

It is common during practical usage of R to accidentally divide by zero. As you can see, this undefined operation yields an infinite value in R. Dividing zero by zero yields the value `NaN`, which stands for *Not a Number*.

Logicals and characters

So far, we've only been dealing with numerics, but there are other atomic data types in R. To wit:

```
> foo <- TRUE        # foo is of the logical data type
> class(foo)         # class() tells us the type
[1] "logical"
> bar <- "hi!"       # bar is of the character data type
> class(bar)
[1] "character"
```

The logical data type (also called Booleans) can hold the values `TRUE` or `FALSE` or, equivalently, T or F. The familiar operators from Boolean algebra are defined for these types:

```
> foo
[1] TRUE
> foo && TRUE                # boolean and
[1] TRUE
> foo && FALSE
[1] FALSE
> foo || FALSE               # boolean or
[1] TRUE
> !foo                       # negation operator
[1] FALSE
```

In a Boolean expression with a logical value and a number, any number that is not 0 is interpreted as `TRUE`.

```
> foo && 1
[1] TRUE
> foo && 2
[1] TRUE
> foo && 0
[1] FALSE
```

Additionally, there are functions and operators that return logical values such as:

```
> 4 < 2           # less than operator
[1] FALSE
> 4 >= 4          # greater than or equal to
[1] TRUE
> 3 == 3          # equality operator
[1] TRUE
> 3 != 2          # inequality operator
[1] TRUE
```

Just as there are functions in R that are only defined for work on the numeric and logical data type, there are other functions that are designed to work only with the character data type, also known as strings:

```
> lang.domain <- "statistics"
> lang.domain <- toupper(lang.domain)
> print(lang.domain)
[1] "STATISTICS"
> # retrieves substring from first character to fourth character
> substr(lang.domain, 1, 4)
[1] "STAT"
> gsub("I", "1", lang.domain)  # substitutes every "I" for "1"
[1] "STAT1ST1CS"
# combines character strings
> paste("R does", lang.domain, "!!!")
[1] "R does STATISTICS !!!"
```

Flow of control

The last topic in this section will be *flow of control* constructs.

The most basic flow of control construct is the `if` statement. The argument to an `if` statement (what goes between the parentheses), is an expression that returns a logical value. The block of code following the `if` statement gets executed only if the expression yields *TRUE*. For example:

```
> if(2 + 2 == 4)
+     print("very good")
[1] "very good"
> if(2 + 2 == 5)
+     print("all hail to the thief")
>
```

It is possible to execute more than one statement if an `if` condition is triggered; you just have to use curly brackets ({}) to contain the statements.

```
> if((4/2==2) && (2*2==4)){
+     print("four divided by two is two...")
+     print("and two times two is four")
+ }
[1] "four divided by two is two..."
[1] "and two times two is four"
>
```

It is also possible to specify a block of code that will get executed if the `if` conditional is *FALSE*.

```
> closing.time <- TRUE
> if(closing.time){
+     print("you don't have to go home")
+     print("but you can't stay here")
+ } else{
+     print("you can stay here!")
+ }
[1] "you don't have to go home"
[1] "but you can't stay here"
> if(!closing.time){
+     print("you don't have to go home")
+     print("but you can't stay here")
+ } else{
+     print("you can stay here!")
+ }
[1] "you can stay here!"
>
```

There are other flow of control constructs (like `while` and `for`), but we won't directly be using them much in this text.

Getting help in R

Before we go further, it would serve us well to have a brief section detailing how to get help in R. Most R tutorials leave this for one of the last sections—if it is even included at all! In my own personal experience, though, getting help is going to be one of the first things you will want to do as you add more bricks to your R knowledge castle. Learning R doesn't have to be difficult; just take it slowly, ask questions, and get help early. Go you!

It is easy to get help with R right at the console. Running the `help.start()` function at the prompt will start a manual browser. From here, you can do anything from going over the basics of R to reading the nitty-gritty details on how R works internally.

You can get help on a particular function in R if you know its name, by supplying that name as an argument to the help function. For example, let's say you want to know more about the `gsub()` function that I sprang on you before. Running the following code:

```
> help("gsub")
> # or simply
> ?gsub
```

will display a manual page documenting what the function is, how to use it, and examples of its usage.

This rapid accessibility to documentation means that I'm never hopelessly lost when I encounter a function which I haven't seen before. The downside to this extraordinarily convenient help mechanism is that I rarely bother to remember the order of arguments, since looking them up is just seconds away.

Occasionally, you won't quite remember the exact name of the function you're looking for, but you'll have an idea about what the name should be. For this, you can use the `help.search()` function.

```
> help.search("chisquare")
> # or simply
> ??chisquare
```

For tougher, more semantic queries, nothing beats a good old fashioned web search engine. If you don't get relevant results the first time, try adding the term *programming* or *statistics* in there for good measure.

Vectors

Vectors are the most basic data structures in R, and they are ubiquitous indeed. In fact, even the single values that we've been working with thus far were actually vectors of length `1`. That's why the interactive R console has been printing `[1]` along with all of our output.

Vectors are essentially *an ordered collection of values of the same atomic data type*. Vectors can be arbitrarily large (with some limitations), or they can be just one single value.

The canonical way of building vectors manually is by using the `c()` function (which stands for *combine*).

```
> our.vect <- c(8, 6, 7, 5, 3, 0, 9)
> our.vect
[1] 8 6 7 5 3 0 9
```

In the preceding example, we created a *numeric* vector of length 7 (namely, Jenny's telephone number).

Note that if we tried to put *character* data types into this vector as follows:

```
> another.vect <- c("8", 6, 7, "-", 3, "0", 9)
> another.vect
[1] "8" "6" "7" "-" "3" "0" "9"
```

R would convert all the items in the vector (called elements) into character data types to satisfy the condition that all elements of a vector must be of the same type. A similar thing happens when you try to use logical values in a vector with numbers; the logical values would be converted into 1 and 0 (for TRUE and FALSE, respectively). These logicals will turn into *TRUE* and *FALSE* (note the quotation marks) when used in a vector that contains characters.

Subsetting

It is very common to want to extract one or more elements from a vector. For this, we use a technique called *indexing* or *subsetting*. After the vector, we put an integer in square brackets (`[]`) called the subscript operator. This instructs R to return the element at that index. The indices (plural for index, in case you were wondering!) for vectors in R start at 1, and stop at the length of the vector.

```
> our.vect[1]                  # to get the first value
[1] 8
```

```
> # the function length() returns the length of a vector
> length(our.vect)
[1] 7
> our.vect[length(our.vect)]   # get the last element of a vector
[1] 9
```

Note that in the preceding code, we used a function in the subscript operator. In cases like these, R evaluates the expression in the subscript operator, and uses the number it returns as the index to extract.

If we get greedy, and try to extract an element at an index that doesn't exist, R will respond with NA, meaning, *not available*. We see this special value cropping up from time to time throughout this text.

```
> our.vect[10]
[1] NA
```

One of the most powerful ideas in R is that you can use vectors to subset other vectors:

```
> # extract the first, third, fifth, and
> # seventh element from our vector
> our.vect[c(1, 3, 5, 7)]
[1] 8 7 3 9
```

The ability to use vectors to index other vectors may not seem like much now, but its usefulness will become clear soon.

Another way to create vectors is by using sequences.

```
> other.vector <- 1:10
> other.vector
 [1]  1  2  3  4  5  6  7  8  9 10
> another.vector <- seq(50, 30, by=-2)
> another.vector
 [1] 50 48 46 44 42 40 38 36 34 32 30
```

Above, the `1:10` statement creates a vector from 1 to 10. `10:1` would have created the same 10 element vector, but in reverse. The `seq()` function is more general in that it allows sequences to be made using steps (among many other things).

Combining our knowledge of sequences and vectors subsetting vectors, we can get the first 5 digits of Jenny's number thusly:

```
> our.vect[1:5]
[1] 8 6 7 5 3
```

Vectorized functions

Part of what makes R so powerful is that many of R's functions take vectors as arguments. These *vectorized* functions are usually extremely fast and efficient. We've already seen one such function, `length()`, but there are many many others.

```
> # takes the mean of a vector
> mean(our.vect)
[1] 5.428571
> sd(our.vect)    # standard deviation
[1] 3.101459
> min(our.vect)
[1] 0
> max(1:10)
[1] 10
> sum(c(1, 2, 3))
[1] 6
```

In practical settings, such as when reading data from files, it is common to have `NA` values in vectors:

```
> messy.vector <- c(8, 6, NA, 7, 5, NA, 3, 0, 9)
> messy.vector
[1]  8  6 NA  7  5 NA  3  0  9
> length(messy.vector)
[1] 9
```

Some vectorized functions will not allow `NA` values by default. In these cases, an extra keyword argument must be supplied along with the first argument to the function.

```
> mean(messy.vector)
[1] NA
> mean(messy.vector, na.rm=TRUE)
[1] 5.428571
> sum(messy.vector, na.rm=FALSE)
[1] NA
> sum(messy.vector, na.rm=TRUE)
[1] 38
```

As mentioned previously, vectors can be constructed from logical values too.

```
> log.vector <- c(TRUE, TRUE, FALSE)
> log.vector
 [1]  TRUE TRUE FALSE
```

Since logical values can be coerced into behaving like numerics, as we saw earlier, if we try to sum a logical vector as follows:.

```
> sum(log.vector)
[1] 2
```

we will, essentially, get a count of the number of `TRUE` values in that vector.

There are many functions in R which operate on vectors and return logical vectors. `is.na()` is one such function. It returns a logical vector—that is, the same length as the vector supplied as an argument—with a `TRUE` in the position of every `NA` value. Remember our messy vector (from just a minute ago)?

```
> messy.vector
[1]  8  6 NA  7  5 NA  3  0  9
> is.na(messy.vector)
[1] FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE
> #  8     6      NA   7     5      NA   3       0    9
```

Putting together these pieces of information, we can get a count of the number of NA values in a vector as follows:

```
> sum(is.na(messy.vector))
[1] 2
```

When you use Boolean operators on vectors, they also return logical vectors of the same length as the vector being operated on.

```
> our.vect > 5
[1]  TRUE  TRUE  TRUE FALSE FALSE FALSE  TRUE
```

If we wanted to—and we do—count the number of digits in Jenny's phone number that are greater than five, we would do so in the following manner:

```
> sum(our.vect > 5)
[1] 4
```

Advanced subsetting

Did I mention that we can use vectors to subset other vectors? When we subset vectors using logical vectors of the same length, only the elements corresponding to the *TRUE* values are extracted. Hopefully, sparks are starting to go off in your head. If we wanted to extract only the legitimate non-NA digits from Jenny's number, we can do it as follows:

```
> messy.vector[!is.na(messy.vector)]
[1] 8 6 7 5 3 0 9
```

This is a very critical trait of R, so let's take our time understanding it; this idiom will come up again and again throughout this book.

The logical vector that yields *TRUE* when an `NA` value occurs in `messy.vector` (from `is.na()`) is then negated (the whole thing) by the negation operator `!`. The resultant vector is *TRUE* whenever the corresponding value in `messy.vector` is not NA. When this logical vector is used to subset the original messy vector, it only extracts the non-NA values from it.

Similarly, we can show all the digits in Jenny's phone number that are greater than five as follows:

```
> our.vect[our.vect > 5]
[1] 8 6 7 9
```

Thus far, we've only been displaying elements that have been extracted from a vector. However, just as we've been assigning and re-assigning variables, we can assign values to various indices of a vector, and change the vector as a result. For example, if Jenny tells us that we have the first digit of her phone number wrong (it's really 9), we can reassign just that element without modifying the others.

```
> our.vect
[1] 8 6 7 5 3 0 9
> our.vect[1] <- 9
> our.vect
[1] 9 6 7 5 3 0 9
```

Sometimes, it may be required to replace all the `NA` values in a vector with the value `0`. To do that with our messy vector, we can execute the following command:

```
> messy.vector[is.na(messy.vector)] <- 0
> messy.vector
[1] 8 6 0 7 5 0 3 0 9
```

Elegant though the preceding solution is, modifying a vector in place is usually discouraged in favor of creating a copy of the original vector and modifying the copy. One such technique for performing this is by using the `ifelse()` function.

Not to be confused with the if/else control construct, `ifelse()` is a function that takes *3* arguments: a test that returns a logical/Boolean value, a value to use if the element passes the test, and one to return if the element fails the test.

The preceding in-place modification solution could be re-implemented with `ifelse` as follows:

```
> ifelse(is.na(messy.vector), 0, messy.vector)
[1] 8 6 0 7 5 0 3 0 9
```

Recycling

The last important property of vectors and vector operations in R is that they can be recycled. To understand what I mean, examine the following expression:

```
> our.vect + 3
[1] 12  9 10  8  6  3 12
```

This expression adds three to each digit in Jenny's phone number. Although it may look so, R is not performing this operation between a vector and a single value. Remember when I said that single values are actually vectors of the length 1? What is really happening here is that R is told to perform element-wise addition on a vector of length 7 and a vector of length 1. Since element-wise addition is not defined for vectors of differing lengths, R recycles the smaller vector until it reaches the same length as that of the bigger vector. Once both the vectors are the same size, then R, element-by-element, performs the addition and returns the result.

```
> our.vect + 3
[1] 12  9 10  8  6  3 12
```

is tantamount to…

```
> our.vect + c(3, 3, 3, 3, 3, 3, 3)
[1] 12  9 10  8  6  3 12
```

If we wanted to extract every other digit from Jenny's phone number, we can do so in the following manner:

```
> our.vect[c(TRUE, FALSE)]
[1] 9 7 3 9
```

This works because the vector `c(TRUE, FALSE)` is repeated until it is of the length `7`, making it equivalent to the following:

```
> our.vect[c(TRUE, FALSE, TRUE, FALSE, TRUE, FALSE, TRUE)]
[1] 9 7 3 9
```

One common snag related to vector recycling that R users (*useRs*, if I may) encounter is that during some arithmetic operations involving vectors of discrepant length, R will warn you if the smaller vector cannot be repeated a whole number of times to reach the length of the bigger vector. This is not a problem when doing vector arithmetic with single values, since `1` can be repeated any number of times to match the length of any vector (which must, of course, be an integer). It would pose a problem, though, if we were looking to add three to every other element in Jenny's phone number.

```
> our.vect + c(3, 0)
[1] 12  6 10  5  6  0 12
Warning message:
In our.vect + c(3, 0) :
  longer object length is not a multiple of shorter object length
```

You will likely learn to love these warnings, as they have stopped many useRs from making grave errors.

Before we move on to the next section, an important thing to note is that in a lot of other programming languages, many of the things that we did would have been implemented using `for` loops and other control structures. Although there is certainly a place for loops and such in R, oftentimes a more sophisticated solution exists in using just vector/matrix operations. In addition to elegance and brevity, the solution that exploits vectorization and recycling is often many, many times more efficient.

Functions

If we need to perform some computation that isn't already a function in R a multiple number of times, we usually do so by defining our own functions. A custom function in R is defined using the following syntax:

```
function.name <- function(argument1, argument2, ...){
  # some functionality
}
```

For example, if we wanted to write a function that determined if a number supplied as an argument was *even*, we can do so in the following manner:

```
> is.even <- function(a.number){
+   remainder <- a.number %% 2
+   if(remainder==0)
+     return(TRUE)
+   return(FALSE)
+ }
>
> # testing it
> is.even(10)
[1] TRUE
> is.even(9)
[1] FALSE
```

As an example of a function that takes more than one argument, let's generalize the preceding function by creating a function that determines whether the first argument is divisible by its second argument.

```
> is.divisible.by <- function(large.number, smaller.number){
+   if(large.number %% smaller.number != 0)
+     return(FALSE)
+   return(TRUE)
+
}
>
> # testing it
> is.divisible.by(10, 2)
[1] TRUE
> is.divisible.by(10, 3)
[1] FALSE
> is.divisible.by(9, 3)
[1] TRUE
```

Our function, `is.even()`, could now be rewritten simply as:

```
> is.even <- function(num){
+   is.divisible.by(num, 2)
+ }
```

It is very common in R to want to apply a particular function to every element of a vector. Instead of using a loop to iterate over the elements of a vector, as we would do in many other languages, we use a function called `sapply()` to perform this. `sapply()` takes a vector and a function as its argument. It then applies the function to every element and returns a vector of results. We can use `sapply()` in this manner to find out which digits in Jenny's phone number are even:

```
> sapply(our.vect, is.even)
[1] FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE
```

This worked great because `sapply` takes each element, and uses it as the argument in `is.even()` which takes only one argument. If you wanted to find the digits that are divisible by three, it would require a little bit more work.

One option is just to define a function `is.divisible.by.three()` that takes only one argument, and use that in `sapply`. The more common solution, however, is to define an unnamed function that does just that in the body of the `sapply` function call:

```
> sapply(our.vect, function(num){is.divisible.by(num, 3)})
[1]  TRUE  TRUE FALSE FALSE  TRUE  TRUE  TRUE
```

Here, we essentially created a function that checks whether its argument is divisible by three, except we don't assign it to a variable, and use it directly in the `sapply` body instead. These one-time-use unnamed functions are called *anonymous functions* or *lambda functions*. (The name comes from Alonzo Church's invention of the lambda calculus, if you were wondering.)

This is somewhat of an advanced usage of R, but it is very useful as it comes up very often in practice.

If we wanted to extract the digits in Jenny's phone number that are divisible by both, *two* and *three*, we can write it as follows:

```
> where.even <- sapply(our.vect, is.even)
> where.div.3 <- sapply(our.vect, function(num){
+   is.divisible.by(num, 3)})
> # "&" is like the "&&" and operator but for vectors
> our.vect[where.even & where.div.3]
[1] 6 0
```

Neat-O!

Note that if we wanted to be sticklers, we would have a clause in the function bodies to preclude a modulus computation, where the first number was smaller than the second. If we had, our function would not have erroneously indicated that 0 was divisible by two and three. I'm not a stickler, though, so the functions will remain as is. Fixing this function is left as an exercise for the (stickler) reader.

Matrices

In addition to the vector data structure, R has the matrix, data frame, list, and array data structures. Though we will be using all these types (except arrays) in this book, we only need to review the first two in this chapter.

A matrix in R, like in math, is a rectangular array of values (of one type) arranged in rows and columns, and can be manipulated as a whole. Operations on matrices are fundamental to data analysis.

One way of creating a matrix is to just supply a vector to the function `matrix()`.

```
> a.matrix <- matrix(c(1, 2, 3, 4, 5, 6))
> a.matrix
     [,1]
[1,]    1
[2,]    2
[3,]    3
[4,]    4
[5,]    5
[6,]    6
```

This produces a matrix with all the supplied values in a single column. We can make a similar matrix with two columns by supplying `matrix()` with an optional argument, `ncol`, that specifies the number of columns.

```
> a.matrix <- matrix(c(1, 2, 3, 4, 5, 6), ncol=2)
> a.matrix
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

We could have produced the same matrix by binding two vectors, `c(1, 2, 3)` and `c(4, 5, 6)` by columns using the `cbind()` function as follows:

```
> a2.matrix <- cbind(c(1, 2, 3), c(4, 5, 6))
```

We could create the transposition of this matrix (where rows and columns are switched) by binding those vectors by row instead:

```
> a3.matrix <- rbind(c(1, 2, 3), c(4, 5, 6))
> a3.matrix
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

or by just using the matrix transposition function in R, `t()`.

```
> t(a2.matrix)
```

Some other functions that operate on whole vectors are `rowSums()/colSums()` and `rowMeans()/colMeans()`.

```
> a2.matrix
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> colSums(a2.matrix)
[1]  6 15
> rowMeans(a2.matrix)
[1] 2.5 3.5 4.5
```

If vectors have `sapply()`, then matrices have `apply()`. The preceding two functions could have been written, more verbosely, as:

```
> apply(a2.matrix, 2, sum)
[1]  6 15
> apply(a2.matrix, 1, mean)
[1] 2.5 3.5 4.5
```

where 1 instructs R to perform the supplied function over its rows, and 2, over its columns.

The matrix multiplication operator in R is `%*%`

```
> a2.matrix %*% a2.matrix
Error in a2.matrix %*% a2.matrix : non-conformable arguments
```

Remember, matrix multiplication is only defined for matrices where the number of columns in the first matrix is equal to the number of rows in the second.

```
> a2.matrix
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> a3.matrix
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
> a2.matrix %*% a3.matrix
     [,1] [,2] [,3]
[1,]   17   22   27
[2,]   22   29   36
[3,]   27   36   45
>
> # dim() tells us how many rows and columns
> # (respectively) there are in the given matrix
> dim(a2.matrix)
[1] 3 2
```

To index the element of a matrix at the second row and first column, you need to supply both of these numbers into the subscripting operator.

```
> a2.matrix[2,1]
[1] 2
```

Many useRs get confused and forget the order in which the indices must appear; remember—it's row first, then columns!

If you leave one of the spaces empty, R will assume you want that whole dimension:

```
> # returns the whole second column
> a2.matrix[,2]
[1] 4 5 6
> # returns the first row
> a2.matrix[1,]
[1] 1 4
```

And, as always, we can use vectors in our subscript operator:

```
> # give me element in column 2 at the first and third row
> a2.matrix[c(1, 3), 2]
[1] 4 6
```

Loading data into R

Thus far, we've only been entering data directly into the interactive R console. For any data set of non-trivial size this is, obviously, an intractable solution. Fortunately for us, R has a robust suite of functions for reading data directly from external files.

Go ahead, and create a file on your hard disk called `favorites.txt` that looks like this:

```
flavor,number
pistachio,6
mint chocolate chip,7
vanilla,5
chocolate,10
strawberry,2
neopolitan,4
```

This data represents the number of students in a class that prefer a particular flavor of soy ice cream. We can read the file into a variable called `favs` as follows:

```
> favs <- read.table("favorites.txt", sep=",", header=TRUE)
```

If you get an error that there is no such file or directory, give R the full path name to your data set or, alternatively, run the following command:

```
> favs <- read.table(file.choose(), sep=",", header=TRUE)
```

The preceding command brings up an open file dialog for letting you navigate to the file you've just created.

The argument `sep=","` tells R that each data element in a row is separated by a comma. Other common data formats have values separated by tabs and pipes (`"|"`). The value of `sep` should then be `"\t"` and `"|"`, respectively.

The argument `header=TRUE` tells R that the first row of the file should be interpreted as the names of the columns. Remember, you can enter `?read.table` at the console to learn more about these options.

Reading from files in this comma-separated-values format (usually with the `.csv` file extension) is so common that R has a more specific function just for it. The preceding data import expression can be best written simply as:

```
> favs <- read.csv("favorites.txt")
```

Now, we have all the data in the file held in a variable of class `data.frame`. A data frame can be thought of as a rectangular array of data that you might see in a spreadsheet application. In this way, a data frame can also be thought of as a matrix; indeed, we can use matrix-style indexing to extract elements from it. A data frame differs from a matrix, though, in that a data frame may have columns of differing types. For example, whereas a matrix would only allow one of these types, the data set we just loaded contains character data in its first column, and numeric data in its second column.

Let's check out what we have by using the `head()` command, which will show us the first few lines of a data frame:

```
> head(favs)
               flavor number
1           pistachio      6
2 mint chocolate chip      7
3             vanilla      5
4           chocolate     10
5          strawberry      2
6          neopolitan      4

> class(favs)
[1] "data.frame"
> class(favs$flavor)
[1] "factor"
> class(favs$number)
[1] "numeric"
```

I lied, ok! So what?! Technically, `flavor` is a *factor* data type, not a character type.

We haven't seen factors yet, but the idea behind them is really simple. Essentially, factors are codings for categorical variables, which are variables that take on one of a finite number of categories—think `{"high", "medium", and "low"}` or `{"control", "experimental"}`.

Though factors are extremely useful in statistical modeling in R, the fact that R, by default, automatically interprets a column from the data read from disk as a type factor if it contains characters, is something that trips up novices and seasoned useRs alike. Because of this, we will primarily prevent this behavior manually by adding the `stringsAsFactors` optional keyword argument to the `read.*` commands:

```
> favs <- read.csv("favorites.txt", stringsAsFactors=FALSE)
> class(favs$flavor)
[1] "character"
```

Much better, for now! If you'd like to make this behavior the new default, read the `?options` manual page. We can always convert to factors later on if we need to!

If you haven't noticed already, I've snuck a new operator on you—`$`, the extract operator. This is the most popular way to extract attributes (or columns) from a data frame. You can also use double square brackets (`[[` and `]]`) to do this.

These are both in addition to the canonical matrix indexing option. The following three statements are thus, in this context, functionally identical:

```
> favs$flavor
[1] "pistachio"           "mint chocolate chip" "vanilla"
[4] "chocolate"           "strawberry"          "neopolitan"
> favs[["flavor"]]
[1] "pistachio"           "mint chocolate chip" "vanilla"
[4] "chocolate"           "strawberry"          "neopolitan"
> favs[,1]
[1] "pistachio"           "mint chocolate chip" "vanilla"
[4] "chocolate"           "strawberry"          "neopolitan"
```

Notice how R has now printed another number in square brackets—besides `[1]`—along with our output. This is to show us that `chocolate` is the fourth element of the vector that was returned from the extraction.

You can use the `names()` function to get a list of the columns available in a data frame. You can even reassign names using the same:

```
> names(favs)
[1] "flavor" "number"
> names(favs)[1] <- "flav"
> names(favs)
[1] "flav"   "number"
```

Lastly, we can get a compact display of the structure of a data frame by using the `str()` function on it:

```
> str(favs)
'data.frame': 6 obs. of  2 variables:
 $ flav  : chr  "pistachio" "mint chocolate chip" "vanilla"
"chocolate" ...
 $ number: num  6 7 5 10 2 4
```

Actually, you can use this function on any R structure—the property of functions that change their behavior based on the type of input is called polymorphism.

Working with packages

Robust, performant, and numerous though base R's functions are, we are by no means limited to them! Additional functionality is available in the form of packages. In fact, what makes R such a formidable statistics platform is the astonishing wealth of packages available (well over 7,000 at the time of writing). R's ecosystem is second to none!

Most of these myriad packages exist on the **Comprehensive R Archive Network** (**CRAN**). CRAN is the primary repository for user-created packages.

One package that we are going to start using right away is the `ggplot2` package. ggplot2 is a plotting system for R. `Base R` has sophisticated and advanced mechanisms to plot data, but many find ggplot2 more consistent and easier to use. Further, the plots are often more aesthetically pleasing by default.

Let's install it!

```
# downloads and installs from CRAN
> install.packages("ggplot2")
```

Now that we have the package downloaded, let's load it into the R session, and test it out by plotting our data from the last section:

```
> library(ggplot2)
> ggplot(favs, aes(x=flav, y=number)) +
+   geom_bar(stat="identity") +
+   ggtitle("Soy ice cream flavor preferences")
```

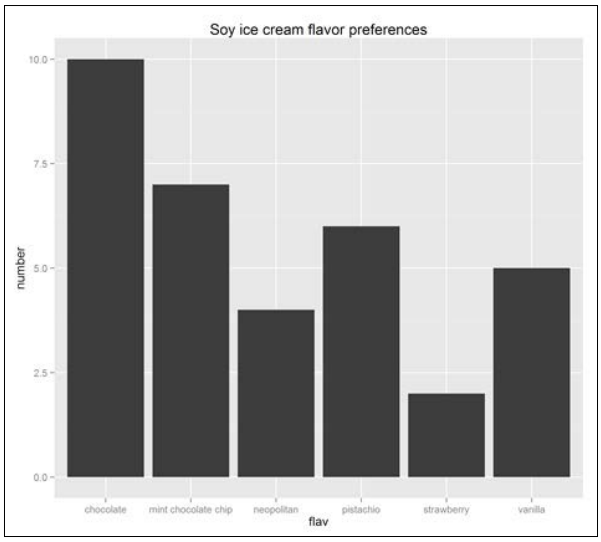

Figure 1.1: Soy ice cream flavor preferences

You're all wrong, Mint Chocolate Chip is way better!

Don't worry about the syntax of the `ggplot` function, yet. We'll get to it in good time.

You will be installing some more packages as you work through this text. In the meantime, if you want to play around with a few more packages, you can install the `gdata` and `foreign` packages that allow you to directly import Excel spreadsheets and SPSS data files respectively directly into R.

Exercises

You can practice the following exercises to help you get a good grasp of the concepts learned in this chapter:

- Write a function called `simon.says` that takes in a character string, and returns that string in all upper case after prepending the string "Simon says: " to the beginning of it.
- Write a function that takes two matrices as arguments, and returns a logical value representing whether the matrices can be matrix multiplied.
- Find a free data set on the web, download it, and load it into R. Explore the structure of the data set.
- Reflect upon how Hester Prynne allowed her scarlet letter to be decorated with flowers by her daughter in the novel The Scarlet Letter: A Romance. To what extent is this indicative of Hester's recasting of the scarlet letter as a positive part of her identity. Back up your thesis with excerpts from the book.

Summary

In this chapter, we learned about the world's greatest analytics platform, R. We started from the beginning and built a foundation, and will now explore R further, based on the knowledge gained in this chapter. By now, you have become well versed in the basics of R (which, paradoxically, is the hardest part).You now know how to:

- Use R as a big calculator to do arithmetic
- Make vectors, operate on them, and subset them expressively
- Load data from disk
- Install packages

You have by no means finished learning about R; indeed, we have gone over mostly just the basics. However, we have enough to continue ahead, and you'll pick up more along the way. Onward to statistics land!

2
The Shape of Data

Welcome back! Since we now have enough knowledge about R under our belt, we can finally move on to applying it. So, join me as we jump out of the R frying pan and into the statistics fire.

Univariate data

In this chapter, we are going to deal with univariate data, which is a fancy way of saying *samples of one variable*—the kind of data that goes into a single R vector. Analysis of univariate data isn't concerned with the why questions—causes, relationships, or anything like that; the purpose of univariate analysis is simply to describe.

In univariate data, one variable—let's call it x—can represent categories like soy ice cream flavors, heads or tails, names of cute classmates, the roll of a die, and so on. In cases like these, we call x a categorical variable.

```
> categorical.data <- c("heads", "tails", "tails", "heads")
```

Categorical data is represented, in the preceding statement, as a vector of character type. In this particular example, we could further specify that this is a binary or dichotomous variable, because it only takes on two values, namely, "heads" and "tails."

Our variable x could also represent a number like air temperature, the prices of financial instruments, and so on. In such cases, we call this a continuous variable.

```
> contin.data <- c(198.41, 178.46, 165.20, 141.71, 138.77)
```

Univariate data of a continuous variable is represented, as seen in the preceding statement, as a vector of numeric type. These data are the stock prices of a hypothetical company that offers a hypothetical commercial statistics platform inferior to R.

You might come to the conclusion that if a vector contains character types, it is a categorical variable, and if it contains numeric types, it is a continuous variable. Not quite! Consider the case of data that contains the results of the roll of a six-sided die. A natural approach to storing this would be by using a numeric vector. However, this isn't a continuous variable, because each result can only take on six distinct values: 1, 2, 3, 4, 5, and 6. This is a *discrete numeric variable*. Other discrete numeric variables can be the number of bacteria in a petri dish, or the number of love letters to cute classmates.

The mark of a continuous variable is that it could take on any value between some theoretical minimum and maximum. The range of values in case of a die roll have a minimum of 1 and a maximum of 6, but it can never be 2.3. Contrast this with, say, the example of the stock prices, which could be zero, zillions, or anything in between.

On occasion, we are unable to neatly classify non-categorical data as either continuous or discrete. In some cases, discrete variables may be treated as if there is an underlying continuum. Additionally, continuous variables can be *discretized*, as we'll see soon.

Frequency distributions

A common way of describing univariate data is with a frequency distribution. We've already seen an example of a frequency distribution when we looked at the preferences for soy ice cream at the end of the last chapter. For each flavor of ice cream (categorical variable), it depicted the count or frequency of the occurrences in the underlying data set.

To demonstrate examples of other frequency distributions, we need to find some data. Fortunately, for the convenience of *useRs* everywhere, R comes preloaded with almost one hundred datasets. You can view a full list if you execute help (`package="datasets"`). There are also hundreds more available from add on packages.

The first data set that we are going to use is `mtcars`—data on the design and performance of 32 automobiles that was extracted from the 1974 Motor Trend US magazine. (To find out more information about this dataset, execute `?mtcars`.)

Take a look at the first few lines of this dataset using the `head` function:

```
> head(mtcars)
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

Check out the `carb` column, which represents the number of carburetors; by now you should recognize this as a discrete numeric variable, though we can (and will!) treat this as a categorical variable for now.

Running the `carb` vector through the unique function yields the distinct values that this vector contains.

```
> unique(mtcars$carb)
[1] 4 1 2 3 6 8
```

We can see that there must be repeats in the `carb` vector, but how many? An easy way for performing a frequency tabulation in R is to use the `table` function:

```
> table(mtcars$carb)
 1  2  3  4  6  8
 7 10  3 10  1  1
```

From the result of the preceding function, we can tell that the are `10` cars with `2` carburetors and `10` with `4`, and there is one car each with `6` and `8` carburetors. The value with the most occurrences in a dataset (in this example, the `carb` column is our whole data set) is called the *mode*. In this case, there are two such values, `2` and `4`, so this dataset is bimodal. (There is a package in R, called `modeest`, to find modes easily.)

Frequency distributions are more often depicted as a chart or plot than as a table of numbers. When the univariate data is categorical, it is commonly represented as a bar chart, as shown in the *Figure 2.1*:

The other data set that we are going to use to demonstrate a frequency distribution of a continuous variable is the `airquality` dataset, which holds the daily air quality measurements from May to September in NY. Take a look at it using the `head` and `str` functions. The univariate data that we will be using is the `Temp` column, which contains the temperature data in degrees Fahrenheit.

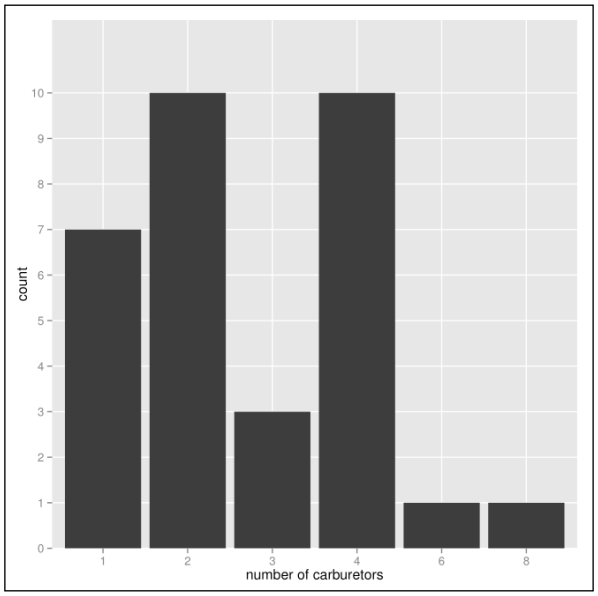

Figure 2.1: Frequency distribution of number of carburetors in mtcars dataset

It would be useless to take the same approach to frequency tabulation as we did in the case of the car carburetors. If we did so, we would have a table containing the frequencies for each of the 40 unique temperatures—and there would be far more if the temperature wasn't rounded to the nearest degree. Additionally, who cares that there was one occurrence of 63 degrees and two occurrences of 64? I sure don't! What we do care about is the approximate temperature.

Our first step towards building a frequency distribution of the temperature data is to *bin* the data—which is to say, we divide the range of values of the vector into a series of smaller intervals. This binning is a method of discretizing a continuous variable. We then count the number of values that fall into that interval.

Choosing the size of bins to use is tricky. If there are too many bins, we run into the same problem as we did with the raw data and have an unwieldy number of columns in our frequency tabulation. If we make too few, however, we lose resolution and may lose important information. Choosing the *right* number of bins is more art than science, but there are certain commonly used heuristics that often produce sensible results.

We can have R construct *n* number of equally-spaced bins for us by using the `cut` function which, in its simplest use case, takes a vector of data and the number of bins to create:

```
> cut(airquality$Temp, 9)
```

We can then feed this result into the `table` function for a far more manageable frequency tabulation:

```
> table(cut(airquality$Temp, 9))

  (56,60.6] (60.6,65.1] (65.1,69.7] (69.7,74.2] (74.2,78.8]
          8          10          14          16          26
(78.8,83.3] (83.3,87.9] (87.9,92.4]   (92.4,97]
         35          22          15           7
```

Rad!

Remember when we used a bar chart to visualize the frequency distributions of categorical data? The common method for visualizing the distribution of discretized continuous data is by using a histogram, as seen in the following image:

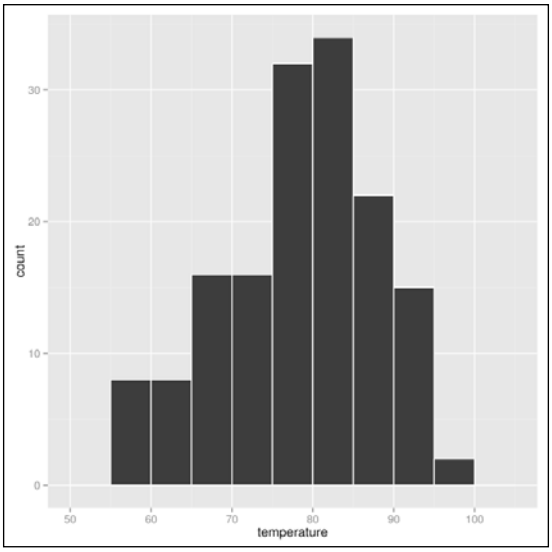

Figure 2.2: Daily temperature measurements from May to September in NYC

Central tendency

One very popular question to ask about univariate data is *What is the typical value?* or *What's the value around which the data are centered?*. To answer these questions, we have to measure the central tendency of a set of data.

We've seen one measure of central tendency already: the mode. The `mtcars$carburetors` data subset was bimodal, with a two and four carburetor setup being the most popular. The mode is the central tendency measure that is applicable to categorical data.

The mode of a discretized continuous distribution is usually considered to be the interval that contains the highest frequency of data points. This makes it dependent on the method and parameters of the binning. Finding the mode of data from a non-discretized continuous distribution is a more complicated procedure, which we'll see later.

Perhaps the most famous and commonly used measure of central tendency is the mean. The mean is the sum of a set of numerics divided by the number of elements in that set. This simple concept can also be expressed as a complex-looking equation:

$$\overline{x} = \frac{\sum x}{n}$$

Where $\overline{x}$ (pronounced *x bar*) is the mean, $\sum x$ is the summation of the elements in the data set, and `n` is the number of elements in the set. (As an aside, if you are intimidated by the equations in this book, don't be! None of them are beyond your grasp—just think of them as sentences of a language you're not proficient in yet.)

The mean is represented as $\overline{x}$ when we are talking about the mean of a sample (or subset) of a larger population, and μ when we are talking about the mean of the population. A population may have too many items to compute the mean directly. When this is the case, we rely on statistics applied to a sample of the population to estimate its parameters.

Another way to express the preceding equation using R constructs is as follows:

```
> sum(nums)/length(nums)    # nums would be a vector of numerics
```

As you might imagine, though, the mean has an eponymous R function that is built-in already:

```
> mean(c(1,2,3,4,5))
[1] 3
```

The mean is not defined for categorical data; remember that mode is the only measure of central tendency that we can use with categorical data.

The mean—occasionally referred to as the arithmetic mean to contrast with the far less often used geometric, harmonic, and trimmed means—while extraordinarily popular is not a very robust statistic. This is because the statistic is unduly affected by outliers (atypically distant data points or observations). A paradigmatic example where the robustness of the mean fails is its application to the different distributions of income.

Imagine the wages of employees in a company called *Marx & Engels, Attorneys at Law*, where the typical worker makes $40,000 a year while the CEO makes $500,000 a year. If we compute the mean of the salaries based on a sample of ten that contains just the exploited class, we will have a fairly accurate representation of the *average* salary of a worker at that company. If however, by the luck of the draw, our sample contains the CEO, the mean of the salaries will skyrocket to a value that is no longer representative or very informative.

More specifically, robust statistics are statistical measures that work well when thrown at a wide variety of different distributions. The mean works well with one particular type of distribution, the normal distribution, and, to varying degrees, fails to accurately represent the central tendency of other distributions.

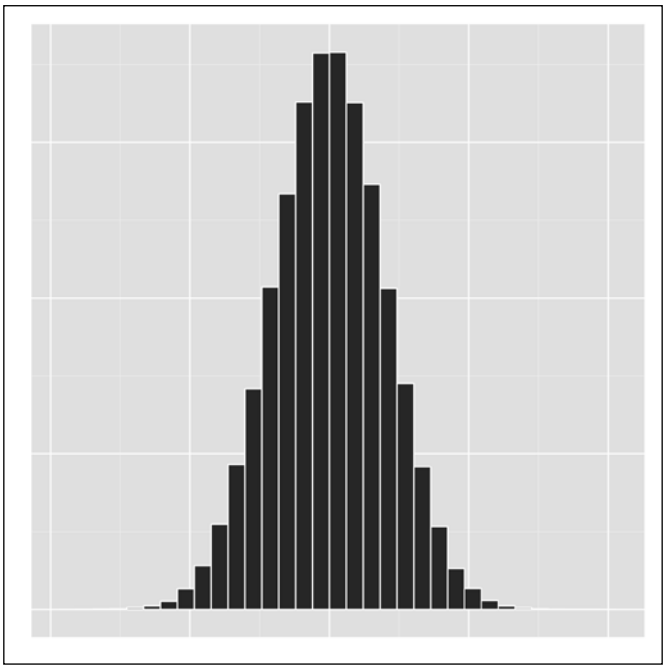

Figure 2.3: A normal distribution

The normal distribution (also called the *Gaussian* distribution if you want to impress people) is frequently referred to as the *bell curve* because of its shape. As seen in the preceding image, the vast majority of the data points lie within a narrow band around the center of the distribution—which is the mean. As you get further and further from the mean, the observations become less and less frequent. It is a symmetric distribution, meaning that the side that is to the right of the mean is a mirror image of the left side of the mean.

Not only is the usage of the normal distribution extremely common in statistics, but it is also ubiquitous in real life, where it can model anything from people's heights to test scores; a few will fare lower than average, and a few fare higher than average, but most are around average.

The utility of the mean as a measure of central tendency becomes strained as the normal distribution becomes more and more skewed, or asymmetrical.

If the majority of the data points fall on the left side of the distribution, with the right side tapering off slower than the left, the distribution is considered *positively skewed* or *right-tailed*. If the longer tail is on the left side and the bulk of the distribution is hanging out to the right, it is called *negatively skewed* or *left-tailed*. This can be seen clearly in the following images:

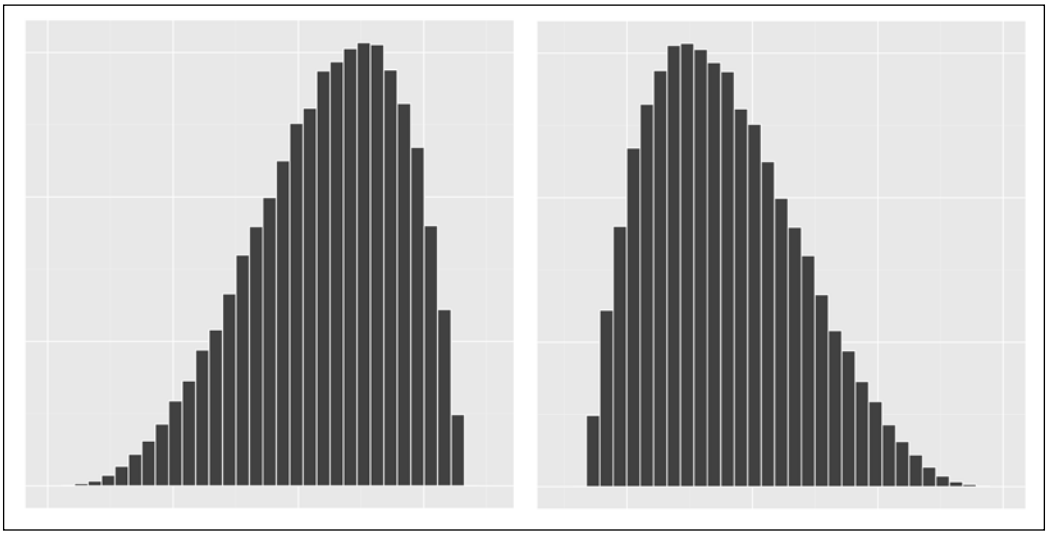

Figure 2.4a: A negatively skewed distribution

Figure 2.4b: A positively skewed distribution

Luckily, for cases of skewed distributions, or other distributions for which the mean is inadequate to describe, we can use the median instead.

The median of a dataset is the middle number in the set after it is sorted. Less concretely, it is the value that cleanly separates the higher-valued half of the data and the lower-valued half.

The median of the set of numbers `{1, 3, 5, 6, 7}` is `5`. In the set of numbers with an even number of elements, the mean of the two middle values is taken to be the median. For example, the median of the set `{3, 3, 6, 7, 7, 10}` is `6.5`. The median is the 50th percentile, meaning that 50 percent of the observations fall below that value.

```
> median(c(3, 7, 6, 10, 3, 7))
[1] 6.5
```

Consider the example of *Marx & Engels, Attorneys at Law* that we referred to earlier. Remember that if the sample of employees' salaries included the CEO, it would give our mean a non-representative value. The median solves our problem beautifully. Let's say our sample of 10 employees' salaries was {41000, 40300, 38000, 500000, 41500, 37000, 39600, 42000, 39900, 39500}. Given this set, the mean salary is $85,880 but the median is $40,100—way more in line with the salary expectations of the proletariat at the law firm.

In symmetric data, the mean and median are often very close to each other in value, if not identical. In asymmetric data, this is not the case. It is telling when the median and the mean are very discrepant. In general, if the median is less than the mean, the data set has a large right tail or outliers/anomalies/erroneous data to the right of the distribution. If the mean is less than the median, it tells the opposite story. The degree of difference between the mean and the median is often an indication of the degree of *skewness*.

This property of the median—resistance to the influence of outliers—makes it a robust statistic. In fact, the median is the most outlier-resistant metric in statistics.

As great as the median is, it's far from being perfect to describe data just by its own. To see what I mean, check out the three distributions in *the following image*. All three have the same mean and median, yet all three are very different distributions.

Clearly, we need to look to other statistical measures to describe these differences.

Before going on to the next chapter, check out the `summary` function in R.

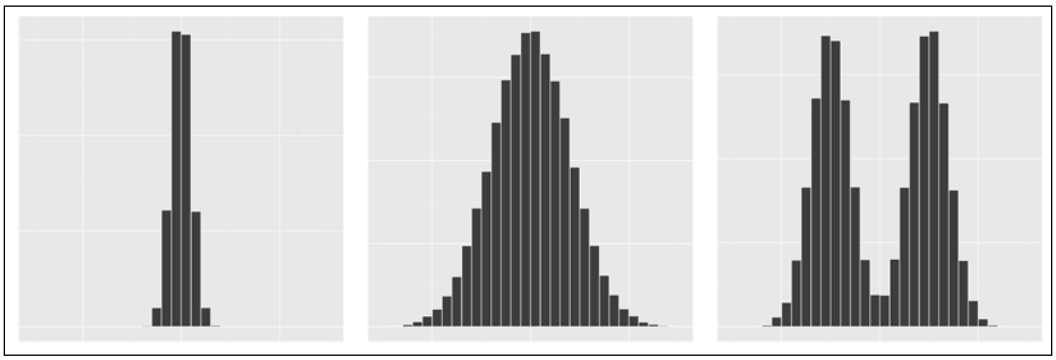

Figure 2.5: three distributions with the same mean and median

Spread

Another very popular question regarding univariate data is, *How variable are the data points?* or *How spread out or dispersed are the observations?*. To answer these questions, we have to measure the spread, or dispersion, of a data sample.

The simplest way to answer that question is to take the smallest value in the dataset and subtract it by the largest value. This will give you the range. However, this suffers from a problem similar to the issue of the mean. The range in salaries at the law firm will vary widely depending on whether the CEO is included in the set. Further, the range is just dependent on two values, the highest and lowest, and therefore, can't speak of the dispersion of the bulk of the dataset.

One tactic that solves the first of these problems is to use the *interquartile range*.

What about measures of spread for categorical data?

The measures of spread that we talk about in this section are only applicable to numeric data. There are, however, measures of spread or diversity of categorical data. In spite of the usefulness of these measures, this topic goes unmentioned or blithely ignored in most data analysis and statistics texts. This is a long and venerable tradition that we will, for the most part, adhere to in this book. If you are interested in learning more about this, search for 'Diversity Indices' on the web.

Remember when we said that the median split a sorted dataset into two equal parts, and that it was the 50th percentile because 50 percent of the observations fell below its value? In a similar way, if you were to divide a sorted data set into four equal parts, or quartiles, the three values that make these divides would be the first, second, and third quartiles respectively. These values can also be called the 25th, 50th, and 75th percentiles. Note that the second quartile, the 50th percentile, and the median are all equivalent.

The interquartile range is the difference between the *third* and *first* quartiles. If you apply the interquartile range to a sample of salaries at the law firm that includes the CEO, the enormous salary will be discarded with the highest 25 percent of the data. However, this still only uses two values, and doesn't speak to the variability of the middle 50 percent.

Well, one way we can use all the data points to inform our spread metric is by subtracting each element of a dataset from the mean of the dataset. This will give us the deviations, or residuals, from the mean. If we add up all these deviations, we will arrive at the sum of the deviations from the mean. Try to find the sum of the deviations from the mean in this set: `{1, 3, 5, 6, 7}`.

If we try to compute this, we notice that the positive deviations are cancelled out by the negative deviations. In order to cope with this, we need to take the absolute value, or the magnitude of the deviation, and sum them.

This is a great start, but note that this metric keeps increasing if we add more data to the set. Because of this, we may want to take the average of these deviations. This is called the average deviation.

For those having trouble following the description in words, the formula for average deviation from the mean is the following:

$$\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)$$

where `μ` is the mean, `N` is the number elements of the sample, and x_i is the ith element of the dataset. It can also be expressed in R as follows:

```
> sum(abs(x - mean(x))) / length(x)
```

Though average deviation is an excellent measure of spread in its own right, its use is commonly—and sometimes unfortunately—supplanted by two other measures.

Instead of taking the absolute value of each residual, we can achieve a similar outcome by squaring each deviation from the mean. This, too, ensures that each residual is positive (so that there is no cancelling out). Additionally, squaring the residuals has the sometimes desirable property of magnifying larger deviations from the mean, while being more forgiving of smaller deviations. The sum of the squared deviations is called (you guessed it!) the sum of squared deviations from the mean or, simply, sum of squares. The average of the sum of squared deviations from the mean is known as the variance and is denoted by σ^2.

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}\left(x_i - \mu\right)^2$$

When we square each deviation, we also square our units. For example, if our dataset held measurements in meters, our variance would be expressed in terms of meters squared. To get back our original units, we have to take the square root of the variance:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(x_i - \mu\right)^2}$$

This new measure, denoted by σ, is the *standard deviation*, and it is one of the most important measures in this book.

Note that we switched from referring to the mean as $\overline{x}$ to referring it as μ. This was not a mistake.

Remember that $\overline{x}$ was the sample mean, and μ represented the population mean. The preceding equations use μ to illustrate that these equations are computing the spread metrics on the population data set, and not on a sample. If we want to describe the variance and standard deviation of a sample, we use the symbols s^2 and s instead of σ^2 and σ respectively, and our equations change slightly:

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}\left(x_i - \overline{x}\right)^2$$

$$s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}\left(x_i - \overline{x}\right)^2}$$

Instead of dividing our sum of squares by the number of elements in the set, we are now dividing it by *n-1*. What gives?

To answer that question, we have to learn a little bit about populations, samples, and estimation.

Populations, samples, and estimation

One of the core ideas of statistics is that we can use a subset of a group, study it, and then make inferences or conclusions about that much larger group.

For example, let's say we wanted to find the average (mean) weight of all the people in Germany. One way do to this is to visit all the 81 million people in Germany, record their weights, and then find the average. However, it is a far more sane endeavor to take down the weights of only a few hundred Germans, and use those to deduce the average weight of all Germans. In this case, the few hundred people we do measure is the sample, and the entirety of people in Germany is called the population.

Now, there are Germans of all shapes and sizes: some heavier, some lighter. If we only pick a few Germans to weigh, we run the risk of, by chance, choosing a group of primarily underweight Germans or overweight ones. We might then come to an inaccurate conclusion about the weight of all Germans. But, as we add more Germans to our sample, those chance variations tend to balance themselves out.

All things being equal, it would be preferable to measure the weights of all Germans so that we can be absolutely sure that we have the right answer, but that just isn't feasible. If we take a large enough sample, though, and are careful that our sample is well-representative of the population, not only can we get extraordinarily close to the actual average weight of the population, but we can quantify our uncertainty. The more Germans we include in our sample, the less uncertain we are about our estimate of the population.

In the preceding case, we are using the sample mean as an estimator of the population mean, and the actual value of the sample mean is called our *estimate*. It turns out that the formula for population mean is a great estimator of the mean of the population when applied to only a sample. This is why we make no distinction between the population and sample means, except to replace the μ with $\overline{x}$. Unfortunately, there exists no perfect estimator for the standard deviation of a population for all population types. There will always be some systematic difference in the expected value of the estimator and the real value of the population. This means that there is some bias in the estimator. Fortunately, we can partially correct it.

Note that the two differences between the population and the sample standard deviation are that (a) the μ is replaced by $\overline{x}$ in the sample standard deviation, and (b) the divisor *n* is replaced by *n-1*.

In the case of the standard deviation of the population, we know the mean μ. In the case of the sample, however, we don't know the population mean, we only have an estimate of the population mean based on the sample mean $\overline{x}$. This must be taken into account and corrected in the new equation. No longer can we divide by the number of elements in the data set—we have to divide by the *degrees of freedom*, which is *n-1*.

What in the world are degrees of freedom? And why is it n-1?

Let's say we were gathering a party of six to play a board game. In this board game, each player controls one of six colored pawns. People start to join in at the board. The first person at the board gets their pick of their favorite colored pawn. The second player has one less pawn to choose from, but she still has a choice in the matter. By the time the last person joins in at the game table, she doesn't have a choice in what pawn she uses; she is forced to use the last remaining pawn. The concept of degrees of freedom is a little like this.

If we have a group of five numbers, but hold the mean of those numbers fixed, all but the last number can vary, because the last number must take on the value that will satisfy the fixed mean. We only have four degrees of freedom in this case.

More generally, the degrees of freedom is the sample size minus the number of parameters estimated from the data. When we are using the mean estimate in the standard deviation formula, we are effectively keeping one of the parameters of the formula fixed, so that only *n-1* observations are free to vary. This is why the divisor of the sample standard deviation formula is *n-1*; it is the degrees of freedom that we are dividing by, not the sample size.

If you thought that the last few paragraphs were heady and theoretical, you're right. If you are confused, particularly by the concept of degrees of freedom, you can take solace in the fact that you are not alone; degrees of freedom, bias, and subtleties of population vs. sample standard deviation are notoriously confusing topics for newcomers to statistics. But you only have to learn it only once!

Probability distributions

Up until this point, when we spoke of distributions, we were referring to frequency distributions. However, when we talk about distributions later in the book—or when other data analysts refer to them—we will be talking about probability distributions, which are much more general.

It's easy to turn a categorical, discrete, or discretized frequency distribution into a probability distribution. As an example, refer to the frequency distribution of carburetors in *the first image in this chapter*. Instead of asking *What number of cars have n number of carburetors?*, we can ask, *What is the probability that, if I choose a car at random, I will get a car with n carburetors?*

We will talk more about probability (and different interpretations of probability) in *Chapter 4*, but for now, probability is a value between 0 and 1 (or 0 percent and 100 percent) that measures how likely an event is to occur. To answer the question *What's the probability that I will pick a car with 4 carburetors?*, the equation is:

$$P\left(I\ will\ pick\ a\ 4\ car\ b\ car\right) = \frac{\#\ of\ 4\ car\ b\ cars}{number\ of\ total\ cars}$$

You can find the probability of picking a car of any one particular number of carburetors as follows:

```
> table(mtcars$carb) / length(mtcars$carb)

      1       2       3       4       6       8
0.21875 0.31250 0.09375 0.31250 0.03125 0.03125
```

Instead of making a bar chart of the frequencies, we can make a bar chart of the probabilities.

This is called a **probability mass function** (**PMF**). It looks the same, but now it maps from carburetors to probabilities, not frequencies. *Figure 2.6a* represents this.

And, just as it is with the bar chart, we can easily tell that `2` and `4` are the number of carburetors most likely to be chosen at random.

We could do the same with discretized numeric variables as well. The following images are a representation of the temperature histogram as a probability mass function.

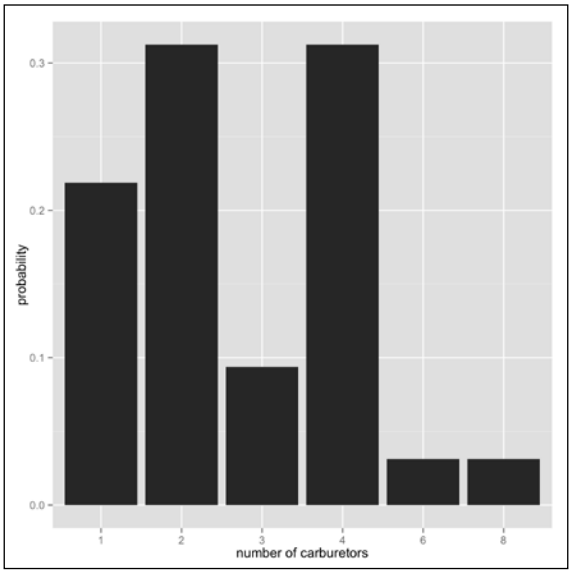

Figure 2.6a: Probability mass function of number of carburetors

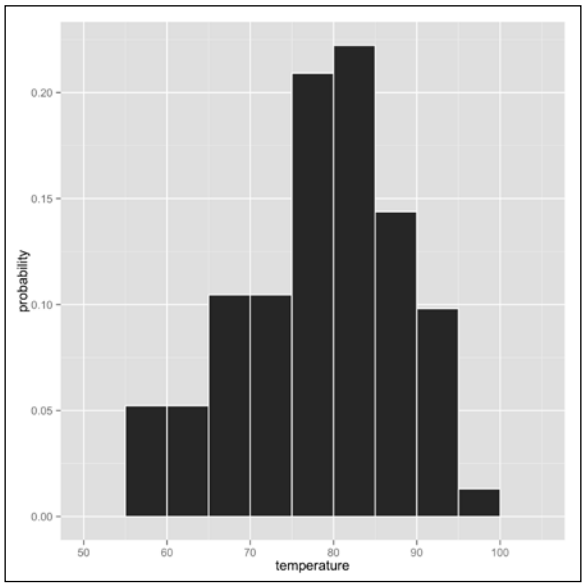

Figure 2.6b: Probability mass function of daily temperature measurements from May to September in NY

Note that this PMF only describes the temperatures of NYC in the data we have.

There's a problem here, though— this PMF is completely dependent on the size of bins (our method of discretizing the temperatures). Imagine that we constructed the bins such that each bin held only one temperature within a degree. In this case, we wouldn't be able to tell very much from the PMF at all, since each specific degree only occurs a few times, if any, in the dataset. The same problem—but worse!—happens when we try to describe continuous variables with probabilities without discretizing them at all. Imagine trying to visualize the probability (or the frequency) of the temperatures if they were measured to the thousandth place (for example, `{90.167, 67.361, ..}`). There would be no visible bars at all!

What we need here is a **probability density function** (**PDF**). A probability density function will tell us the relative likelihood that we will experience a certain temperature. *The next image* shows a PDF that fits the temperature data that we've been playing with; it is analogous to, but better than, the histogram we saw in the beginning of the chapter and the PMF in *the preceding figure*.

The first thing you'll notice about this new plot is that it is smooth, not jagged or boxy like the histogram and PMFs. This should intuitively make more sense, because temperatures are a continuous variable, and there is likely to be no sharp cutoffs in the probability of experiencing temperatures from one degree to the next.

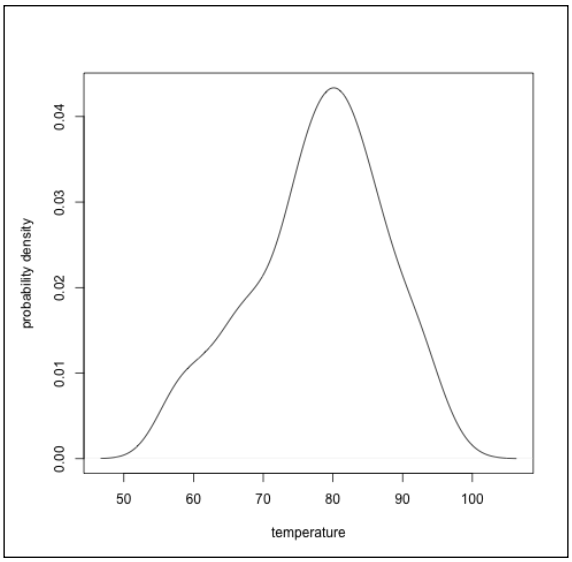

Figure 2.7: Three distributions with the same mean and median

The second thing you should notice is that the units and the values on the *y* axis have changed. The *y* axis no longer represents probabilities—it now represents probability densities. Though it may be tempting, you can't look at this function and answer the question *What is the probability that it will be exactly 80 degrees?*. Technically, the probability of it being 80.0000 exactly is microscopically small, almost zero. But that's okay! Remember, we don't care what the probability of experiencing a temperature of 80.0000 is—we just care the probability of a temperature around there.

We can answer the question *What's the probability that the temperature will be between a particular range?*. The probability of experiencing a temperature, say 80 to 90 degrees, is the area under the curve from 80 to 90. Those of you unfortunate readers who know calculus will recognize this as the integral, or anti-derivative, of the PDF evaluated over the range,

$$\int_{80}^{90} f(x)\,dx$$

where `f(x)` is the probability density function.

The next image shows the area under the curve for this range in pink. You can immediately see that the region covers a lot of area—perhaps one third. According to R, it's about 34 percent.

```
> temp.density <- density(airquality$Temp)
> pdf <- approxfun(temp.density$x, temp.density$y, rule=2)
> integrate(pdf, 80, 90)
     0.3422287 with absolute error < 7.5e-06
```

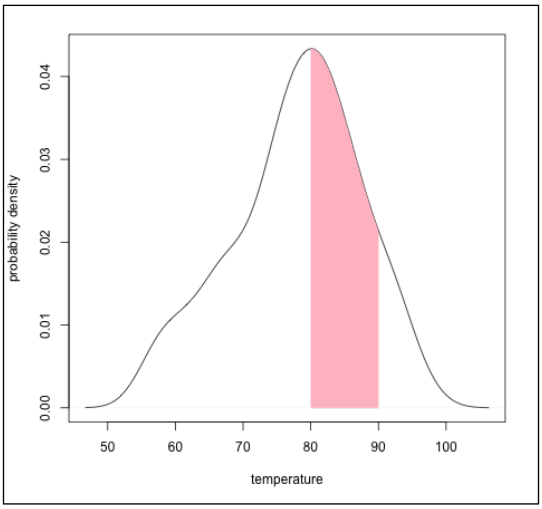

Figure 2.8: PDF with highlighted interval

We don't get a probability density function from the sample for free. The PDF has to be estimated. The PDF isn't so much trying to convey the information about the sample we have as attempting to model the underlying distribution that gave rise to that sample.

To do this, we use a method called *kernel density estimation*. The specifics of kernel density estimation are beyond the scope of this book, but you should know that the density estimation is heavily governed by a parameter that controls the smoothness of the estimation. This is called the *bandwidth*.

How do we choose the bandwidth? Well, it's just like choosing the size to make the bins in a histogram: there's no right answer. It's a balancing act between reducing chance or noise in the model and not losing important information by smoothing over pertinent characteristics of the data. This is a tradeoff we will see time and time again throughout this text.

Anyway, the great thing about PDFs is that you don't have to know calculus to interpret PDFs. Not only are PDFs a useful tool analytically, but they make for a top-notch visualization of the shape of data.

By the way…

Remember when we were talking about modes, and I said that finding the mode of non-discretized continuously distributed data is a more complicated procedure than for discretized or categorical data? The mode for these types of univariate data is the peak of the PDF. So, in the temperature example, the mode is around 80 degrees.

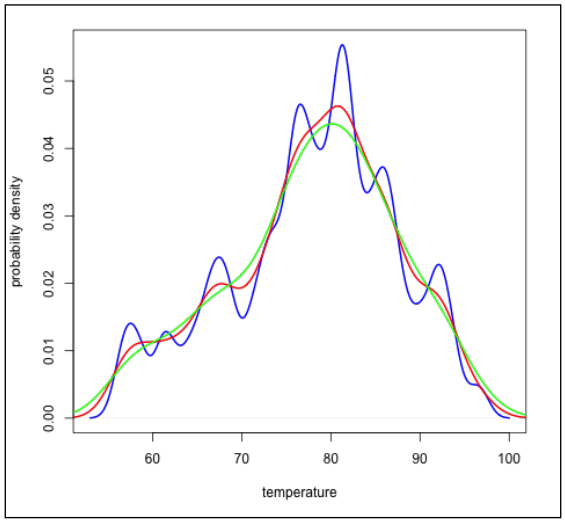

Figure 2.9: Three different bandwidths used on the same data.

Visualization methods

In an earlier image, we saw three very different distributions, all with the same mean and median. I said then that we need to quantify variance to tell them apart. In the following image, there are three very different distributions, all with the same mean, median, and variance.

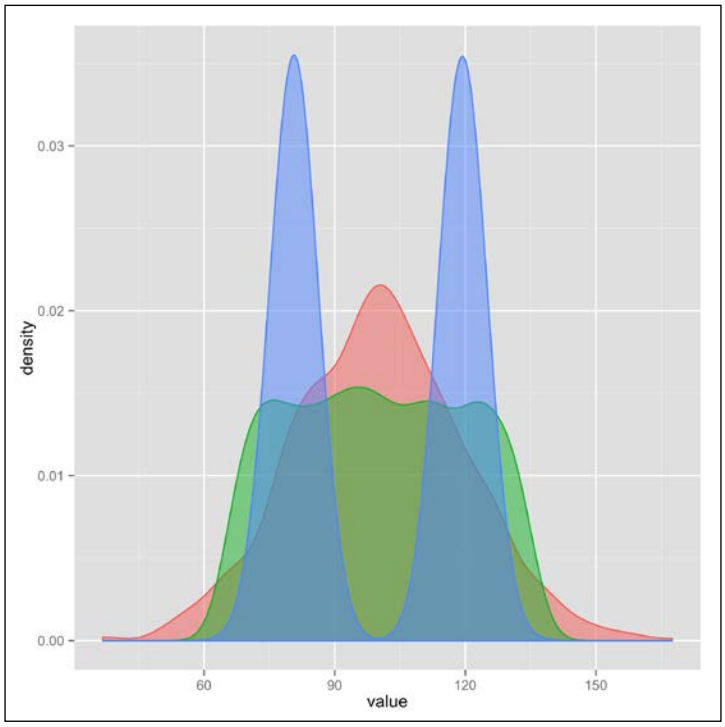

Figure 2.10: Three PDFs with the same mean, median, and standard deviation

If you just rely on basic summary statistics to understand univariate data, you'll never get the full picture. It's only when we visualize it that we can clearly see, at a glance, whether there are any clusters or areas with a high density of data points, the number of clusters there are, whether there are outliers, whether there is a pattern to the outliers, and so on. When dealing with univariate data, the shape is the most important part (that's why this chapter is called *Shape of Data*!).

We will be using ggplot2's `qplot` function to investigate these shapes and visualize these data. `qplot` (for *quick plot*) is the simpler cousin of the more expressive `ggplot` function. `qplot` makes it easy to produce handsome and compelling graphics using consistent grammar. Additionally, much of the skills, lessons, and know-how from `qplot` are transferrable to `ggplot` (for when we have to get more advanced).

What's ggplot2? Why are we using it?

There are a few plotting mechanisms for R, including the default one that comes with R (called *base R*). However, `ggplot2` seems to be a lot of people's favorite. This is not unwarranted, given its wide use, excellent documentation, and consistent grammar.

Since the base R graphics subsystem is what I learned to wield first, I've become adept at using it. There are certain types of plots that I produce faster using base R, so I still use it on a regular basis (*Figure 2.8* to *Figure 2.10* were made using base R!).

Though we will be using `ggplot2` for this book, feel free to go your own way when making your very own plots.

Most of the graphics in this section are going to take the following form:

```
> qplot(column, data=dataframe, geom=...)
```

where `column` is a particular column of the data frame `dataframe`, and the `geom` keyword argument specifies a geometric object—it will control the type of plot that we want. For visualizing univariate data, we don't have many options for `geom`. The three types that we will be using are `bar`, `histogram`, and `density`. Making a bar graph of the frequency distribution of the number of carburetors couldn't be easier:

```
> library(ggplot2)
> qplot(factor(carb), data=mtcars, geom="bar")
```

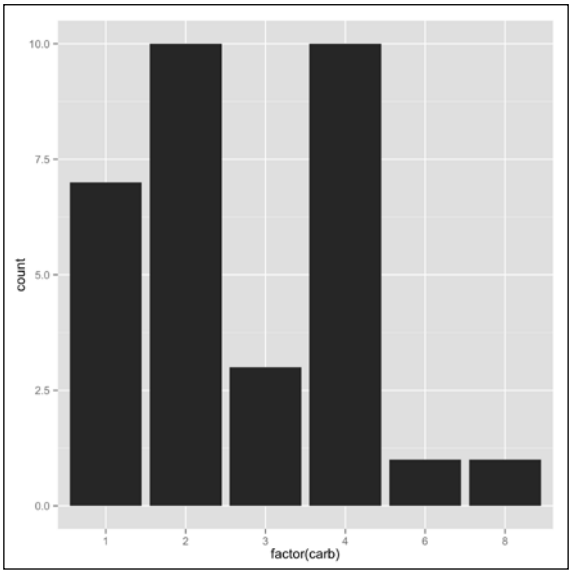

Figure 2.11: Frequency distribution of the number of carburetors

Using the `factor` function on the `carb` column makes the plot look better in this case.

We could, if we wanted to, make an unattractive and distracting plot by coloring all the bars a different color, as follows:

```
> qplot(factor(carb),
+       data=mtcars,
+       geom="bar",
+       fill=factor(carb),
+       xlab="number of carburetors")
```

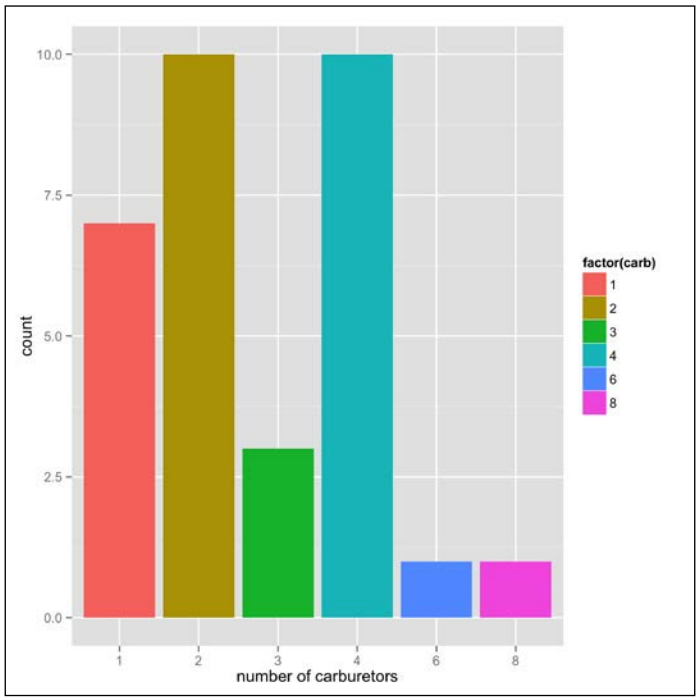

Figure 2.12: With color and label modification

We also relabeled the *x* axis (which is automatically set by `qplot`) to more informative text.

It's just as easy to make a histogram of the temperature data—the main difference is that we switch `geom` from `bar` to `histogram`:

```
> qplot(Temp, data=airquality, geom="histogram")
```

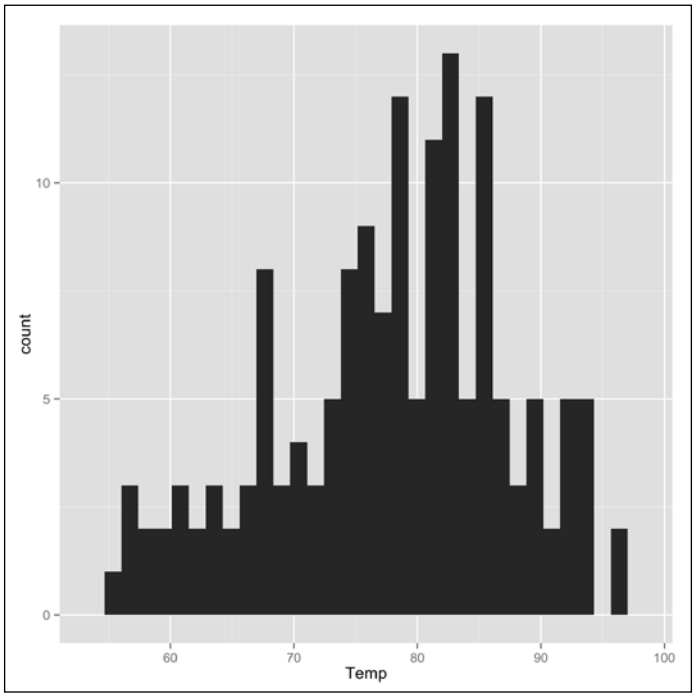

Figure 2.13: Histogram of temperature data

Why doesn't it look like the first histogram in the beginning of the chapter, you ask? Well, that's because of two reasons:

- I adjusted the bin width (size of the bins)
- I added color to the outline of the bars

The code I used for *the first histogram* looked as follows:

```
> qplot(Temp, data=airquality, geom="histogram",
+       binwidth=5, color=I("white"))
```

Making plots of the approximation of the PDF are similarly simple:

```
> qplot(Temp, data=airquality, geom="density")
```

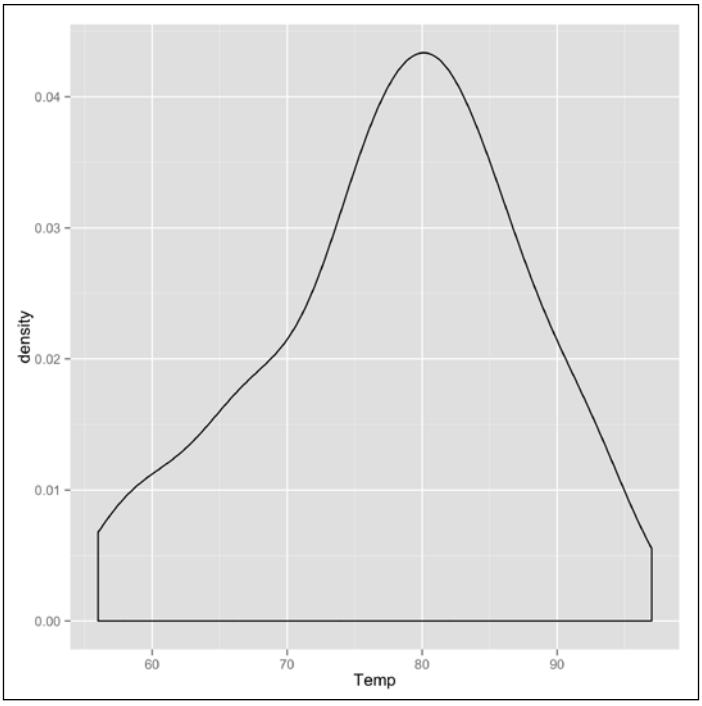

Figure 2.14: PDF of temperature data

By itself, I think the preceding plot is rather unattractive. We can give it a little more flair by:

- Filling the curve pink
- Adding a little transparency to the fill

```
> qplot(Temp, data=airquality, geom="density",
+       adjust=.5,       # changes bandwidth
+       fill=I("pink"),
+       alpha=I(.5),     # adds transparency
+       main="density plot of temperature data")
```

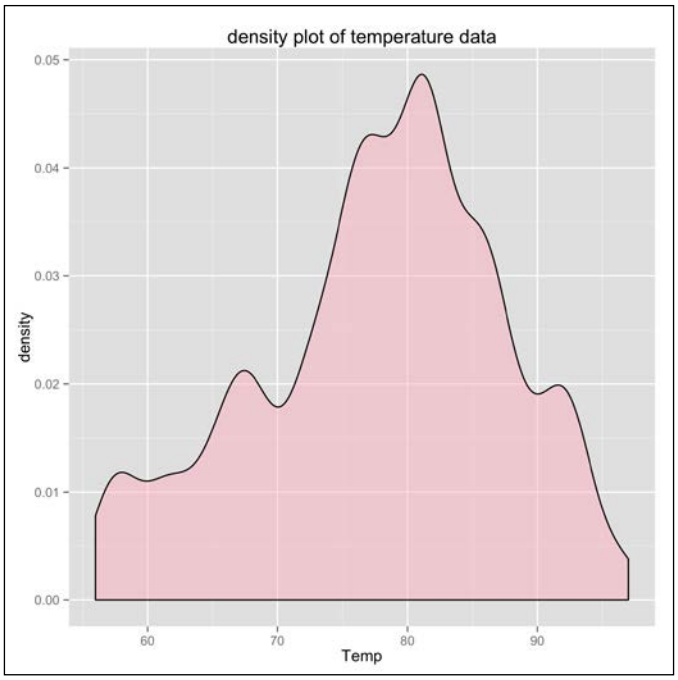

Figure 2.15: Figure 2.14 with modifications

Now that's a handsome plot!

Notice that we also made the bandwidth smaller than the default (1, which made the PDF more squiggly) and added a title to the plot with the main function.

Exercises

Here are a few exercises for you to revise the concepts learned in this chapter:

- Write an R function to compute the interquartile range.
- Learn about windorized, geometric, harmonic, and trimmed means. To what extent do these metrics solve the problem of the non-robustness of the arithmetic mean?
- Craft an assessment of Virginia Woolf's impact on feminine discourse in the 20th century. Be sure to address both prosaic and lyrical forms in your response.

Summary

One of the hardest things about data analysis is statistics, and one of the hardest things about statistics (not unlike computer programming) is that the beginning is the toughest hurdle, because the concepts are so new and unfamiliar. As a result, some might find this to be one of the more challenging chapters in this text.

However, hard work during this phase pays enormous dividends; it provides a sturdy foundation on which to pile on and organize new knowledge.

To recap, in this chapter, we learned about univariate data. We also learned about:

- The types of univariate data
- How to measure the central tendency of these data
- How to measure the spread of these data
- How to visualize the shape of these data

Along the way, we also learned a little bit about probability distributions and population/sample statistics.

I'm glad you made it through! Relax, make yourself a mocktail, and I'll see you at *Chapter 3*, *Describing Relationships* shortly!

3
Describing Relationships

Is there a relationship between smoking and lung cancer? Do people who care for dogs live longer? Is your university's admissions department sexist?

Tackling these exciting questions is only possible when we take a step beyond simply describing univariate data sets—one step beyond!

Multivariate data

In this chapter, we are going to describe relationships, and begin working with multivariate data, which is a fancy way of saying *samples containing more than one variable*.

The troublemaker reader might remark that all the datasets that we've worked with thus far (`mtcars` and `airquality`) have contained more than one variable. This is technically true—but only technically. The fact of the matter is that we've only been working with one of the dataset's variables at any one time. Note that multivariate analytics is not the same as doing univariate analytics on more than one variable–multivariate analyses and describing relationships involve several variables at the same time.

To put this more concretely, in the last chapter we described the shape of, say, the temperature readings in the `airquality` dataset.

```
> head(airquality)
Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
5    NA      NA 14.3   56     5   5
6    28      NA 14.9   66     5   6
```

In this chapter, we will be exploring whether there is a relationship between temperature and the month in which the temperature was taken (spoiler alert: there is!).

The kind of multivariate analysis you perform is heavily influenced by the type of data that you are working with. There are three broad classes of bivariate (or *two variable*) relationships:

- The relationship between one categorical variable and one continuous variable
- The relationship between two categorical variables
- The relationship between two continuous variables

We will get into all of these in the next three sections. In the section after that, we will touch on describing the relationships between more than two variables. Finally, following in the tradition of the previous chapter, we will end with a section on how to create your own plots to capture the relationships that we'll be exploring.

Relationships between a categorical and a continuous variable

Describing the relationship between categorical and continuous variables is perhaps the most familiar of the three broad categories.

When I was in the fifth grade, my class had to participate in an area-wide science fair. We were to devise our own experiment, perform it, and then present it. For some reason, in my experiment I chose to water some lentil sprouts with tap water and some with alcohol to see if they grew differently.

When I measured the heights and compared the measurements of the teetotaller lentils versus the drunken lentils, I was pointing out a relationship between a categorical variable (alcohol/no-alcohol) and a continuous variable (heights of the seedlings).

Note that I wasn't trying to make a broader statement about how alcohol affects plant growth. In the grade-school experiment, I was just summarizing the differences in the heights of those plants—the ones that were in the experiment. In order to make statements or draw conclusions about how alcohol affects plant growth in general, we would be exiting the realm of exploratory data analysis and entering the domain of inferential statistics, which we will discuss in the next unit.

The alcohol could have made the lentils grow faster (it didn't), grow slower (it did), or grow at the same rate as the tap water lentils. All three of these possibilities constitute a relationship: greater than, less than, or equal to.

To demonstrate how to uncover the relationship between these two types of variables in R, we will be using the `iris` dataset that is conveniently built right into R.

```
> head(iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

This is a famous dataset and is used today primarily for teaching purposes. It gives the lengths and widths of the petals and sepals (another part of the flower) of 150 Iris flowers. Of the 150 flowers, it has 50 measurements each from three different species of Iris flowers: *setosa*, *versicolor*, and *virginica*.

By now, we know how to take the mean of all the petal lengths:

```
> mean(iris$Petal.Length)
[1] 3.758
```

But we could also take the mean of the petal lengths of each of the three species to see if there is any difference in the means.

Naively, one might approach this task in R as follows:

```
> mean(iris$Petal.Length[iris$Species=="setosa"])
[1] 1.462
> mean(iris$Petal.Length[iris$Species=="versicolor"])
[1] 4.26
> mean(iris$Petal.Length[iris$Species=="virginica"])
[1] 5.552
```

But, as you might imagine, there is a far easier way to do this:

```
> by(iris$Petal.Length, iris$Species, mean)

iris$Species: setosa
[1] 1.462
```

```
------------------------------------------
iris$Species: versicolor
[1] 4.26
------------------------------------------
iris$Species: virginica
[1] 5.552
```

`by` is a handy function that applies a function to split the subsets of data. In this case, the `Petal.Length` vector is divided into three subsets for each species, and then the `mean` function is called on each of those subsets. It appears as if the setosas in this sample have way shorter petals than the other two species, with the virginica samples' petal length beating out versicolor's by a smaller margin.

Although means are probably the most common statistic to be compared between categories, it is not the only statistic we can use to compare. If we had reason to believe that the virginicas have a more widely varying petal length than the other two species, we could pass the `sd` function to the `by` function as follows

```
> by(iris$Petal.Length, iris$Species, sd)
```

Most often, though, we want to be able to compare many statistics between groups at one time. To this end, it's very common to pass in the `summary` function:

```
> by(iris$Petal.Length, iris$Species, summary)

iris$Species: setosa
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.000   1.400   1.500   1.462   1.575   1.900
------------------------------------------------
iris$Species: versicolor
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   3.00    4.00    4.35    4.26    4.60    5.10
------------------------------------------------
iris$Species: virginica
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  4.500   5.100   5.550   5.552   5.875   6.900
```

As common as this idiom is, it still presents us with a lot of dense information that is difficult to make sense of at a glance. It is more common still to visualize the differences in continuous variables between categories using a box-and-whisker plot:

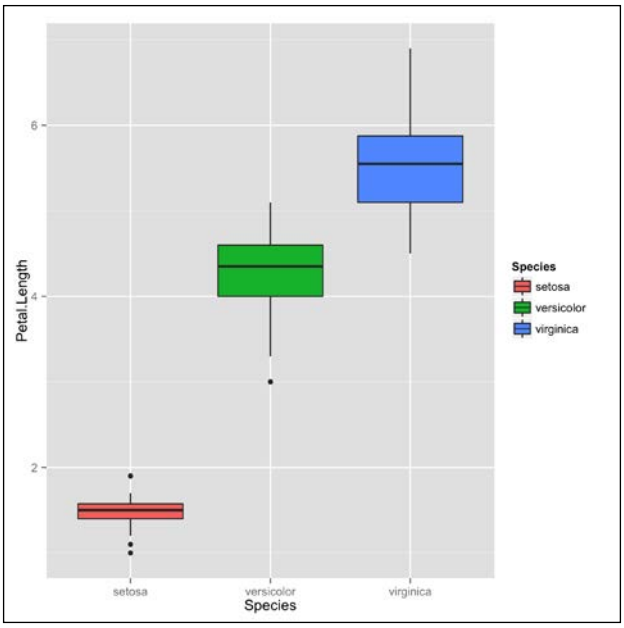

Figure 3.1: A box-and-whisker plot depicting the relationship between the petal lengths of the different iris species in iris dataset

A box-and-whisker plot (or simply, a *box plot* if you have places to go, and you're in a rush) displays a stunningly large amount of information in a single chart. Each categorical variable has its own box and whiskers. The bottom and top ends of the box represent the first and third quartile respectively, and the black band inside the box is the median for that group, as shown in the following figure:

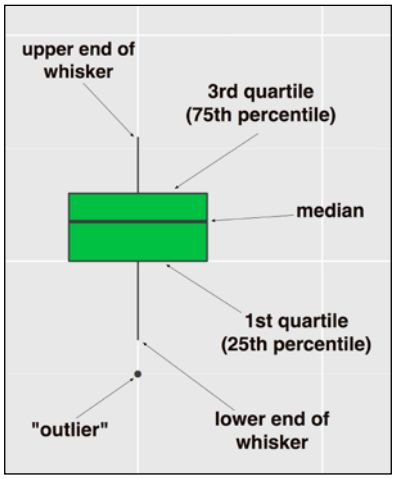

Figure 3.2: The anatomy of a box plot

Depending on whom you talk to and what you use to produce your plots, the edges of the whiskers can mean a few different things. In my favorite variation (called *Tukey's variation*), the bottom of the whiskers extend to the lowest datum within 1.5 times the interquartile range below the bottom of the box. Similarly, the very top of the whisker represents the highest datum 1.5 interquartile ranges above the third quartile (remember: interquartile range is the third quartile minus the first). This is, coincidentally, the variation that `ggplot2` uses.

The great thing about box plots is that not only do we get a great sense of the central tendency and dispersion of the distribution within a category, but we can also immediately spot the important differences between each category.

From the box plot in the previous image, it's easy to tell what we already know about the central tendency of the petal lengths between species: that the setosas in this sample have the shortest petals; that the virginica have the longest on average; and that versicolors are in the middle, but are closer to the virginicas.

In addition, we can see that the setosas have the thinnest dispersion, and that the virginica have the highest—when you disregard the outlier.

But remember, we are not saying anything, or drawing any conclusions yet about Iris flowers in general. In all of these analyses, we are treating all the data we have as the population of interest; in this example, the 150 flowers measured are our population of interest.

Before we move on to the next broad category of relationships, let's look at the `airquality` dataset, treat the month as the categorical variable, the temperature as the continuous variable, and see if there is a relationship between the average temperature across months.

```
> by(airquality$Temp, airquality$Month, mean)
airquality$Month: 5
[1] 65.54839
---------------------------------------------
airquality$Month: 6
[1] 79.1
---------------------------------------------
airquality$Month: 7
[1] 83.90323
---------------------------------------------
airquality$Month: 8
[1] 83.96774
---------------------------------------------
airquality$Month: 9
[1] 76.9
```

This is precisely what we would expect from a city in the Northern hemisphere:

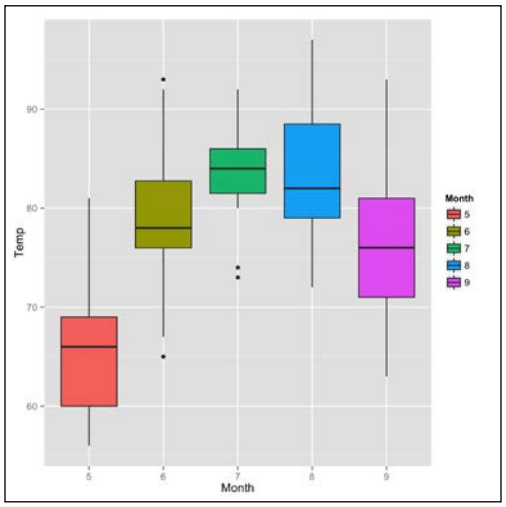

Figure 3.3: A Box plot of NYC temperatures across months (May to September)

Relationships between two categorical variables

Describing the relationships between two categorical variables is done somewhat less often than the other two broad types of bivariate analyses, but it is just as fun (and useful)!

To explore this technique, we will be using the dataset `UCBAdmissions`, which contains the data on graduate school applicants to the University of California Berkeley in 1973.

Before we get started, we have to wrap the dataset in a call to `data.frame` for coercing it into a data frame type variable—I'll explain why, soon.

```
ucba <- data.frame(UCBAdmissions)
> head(ucba)
     Admit Gender Dept Freq
1 Admitted   Male    A  512
2 Rejected   Male    A  313
3 Admitted Female    A   89
4 Rejected Female    A   19
5 Admitted   Male    B  353
6 Rejected   Male    B  207
```

Now, what we want is a count of the frequencies of number of students in each of the following four categories:

- Accepted female
- Rejected female
- Accepted male
- Rejected male

Do you remember the frequency tabulation at the beginning of the last chapter? This is similar—except that now we are dividing the set by one more variable. This is known as *cross-tabulation* or *cross tab*. It is also sometimes referred to as a contingency table. The reason we had to coerce `UCBAdmissions` into a data frame is because it was already in the form of a cross tabulation (except that it further broke the data down into the different departments of the grad school). Check it out by typing `UCBAdmissions` at the prompt.

We can use the `xtabs` function in R to make our own cross-tabulations:

```
# the first argument to xtabs (the formula) should
# be read as: frequency *by* Gender and Admission
> cross <- xtabs(Freq ~ Gender+Admit, data=ucba)
> cross
        Admit
Gender   Admitted Rejected
  Male       1198     1493
  Female      557     1278
```

Here, at a glance, we can see that there were `1198` males that were admitted, `557` females that were admitted, and so on.

Is there a gender bias in UCB's graduate admissions process? Perhaps, but it's hard to tell from just looking at the 2x2 contingency table. Sure, there are fewer females accepted than males, but there are also, unfortunately, far fewer females that applied to UCB in the first place.

To aid us in either implicating UCB of a sexist admissions machine or exonerating them, it would help to look at a proportions table. Using a proportions table, we can easily compare the proportion of the total number of males who were accepted versus the proportion of the total number of females who were accepted. If the proportions are more or less equal, we can conclude that gender does not constitute a factor in UCB's admissions process. If this is the case, gender and admission status is said to be conditionally independent.

```
> prop.table(cross, 1)
        Admit
Gender    Admitted  Rejected
  Male   0.4451877 0.5548123
  Female 0.3035422 0.6964578
```

Why did we supply 1 as an argument to `prop.table`? Look up the documentation at the R prompt. When would we want to use `prop.table(cross, 2)`?

Here, we can see that while 45 percent of the males who applied were accepted, only 30 percent of the females who applied were accepted. This is evidence that the admissions department is sexist, right? Not so fast, my friend!

This is precisely what a lawsuit lodged against UCB purported. When the issue was looked into further, it was discovered that, at the department level, women and men actually had similar admissions rates. In fact, some of the departments appeared to have a small but significant bias in favor of women. Check out department A's proportion table, for example:

```
> cross2 <- xtabs(Freq ~ Gender + Admit, data=ucba[ucba$Dept=="A",])
> prop.table(cross2, 1)
        Admit
Gender    Admitted  Rejected
  Male   0.6206061 0.3793939
  Female 0.8240741 0.1759259
```

If there were any bias in admissions, these data didn't prove it. This phenomenon, where a trend that appears in combined groups of data disappears or reverses when broken down into groups is known as *Simpson's Paradox*. In this case, it was caused by the fact that women tended to apply to departments that were far more selective.

This is probably the most famous case of Simpson's Paradox, and it is also why this dataset is built into R. The lesson here is to be careful when using pooled data, and look out for hidden variables.

The relationship between two continuous variables

Do you think that there is a relationship between women's heights and their weights? If you said *yes*, congratulations, you're right!

We can verify this assertion by using the data in R's built-in dataset, `women`, which holds the height and weight of 15 American women from ages 30 to 39.

```
> head(women)
  height weight
1     58    115
2     59    117
3     60    120
4     61    123
5     62    126
6     63    129
> nrow(women)
[1] 15
```

Specifically, this relationship is referred to as a positive relationship, because as one of the variable increases, we expect an increase in the other variable.

The most typical visual representation of the relationship between two continuous variables is a *scatterplot*.

A scatterplot is displayed as a group of points whose position along the x-axis is established by one variable, and the position along the y-axis is established by the other. When there is a positive relationship, the dots, for the most part, start in the lower-left corner and extend to the upper-right corner, as shown in the following figure. When there is a negative relationship, the dots start in the upper-left corner and extend to the lower-right one. When there is no relationship, it will look as if the dots are all over the place.

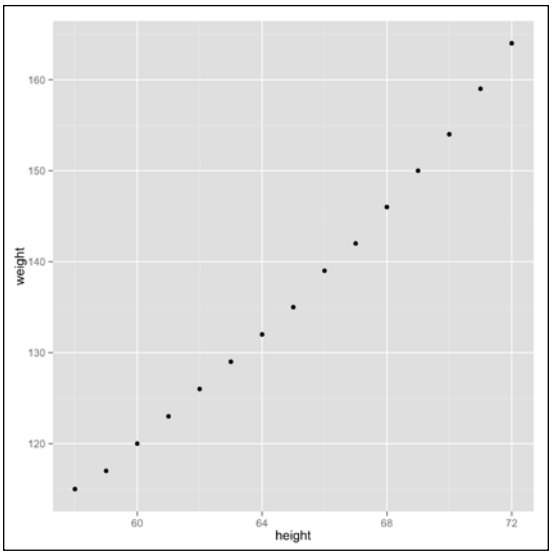

Figure 3.4: Scatterplot of women's heights and weights

The more the dots look like they form a straight line, the stronger is the relationship between two continuous variables is said to be; the more diffuse the points, the weaker is the relationship. The dots in the preceding figure look almost exactly like a straight line—this is pretty much as strong a relationship as they come.

These kinds of relationships are colloquially referred to as correlations.

Covariance

As always, visualizations are great—necessary, even—but on most occasions, we are going to quantify these correlations and summarize them with numbers.

The simplest measure of correlation that is widely use is the *covariance*. For each pair of values from the two variables, the differences from their respective means are taken. Then, those values are multiplied. If they are both positive (that is, both the values are above their respective means), then the product will be positive too. If both the values are below their respective means, the product is still positive, because the product of two negative numbers is positive. Only when one of the values is above its mean will the product be negative.

$$\text{cov}_{xy} = \frac{\sum (x - \bar{x})(y - \bar{y})}{(n-1)}$$

Remember, in sample statistics we divide by the degrees of freedom and not the sample size. Note that this means that the covariance is only defined for two vectors that have the same length.

We can find the covariance between two variables in R using the `cov` function. Let's find the covariance between the heights and weights in the dataset, women:

```
> cov(women$weight, women$height)
[1] 69
# the order we put the two columns in
# the arguments doesn't matter
> cov(women$height, women$weight)
[1] 69
```

The covariance is positive, which denotes a positive relationship between the two variables.

The covariance, by itself, is difficult to interpret. It is especially difficult to interpret in this case, because the measurements use different scales: inches and pounds. It is also heavily dependent on the variability in each variable.

Consider what happens when we take the covariance of the weights in *pounds* and the heights in *centimeters*.

```
# there are 2.54 centimeters in each inch
# changing the units to centimeters increases
# the variability within the height variable
> cov(women$height*2.54, women$weight)
[1] 175.26
```

Semantically speaking, the relationship hasn't changed, so why should the covariance?

Correlation coefficients

A solution to this quirk of covariance is to use Pearson's correlation coefficient instead. Outside its colloquial context, when the word correlation is uttered—especially by analysts, statisticians, or scientists—it usually refers to *Pearson's correlation*.

Pearson's correlation coefficient is different from covariance in that instead of using the sum of the products of the deviations from the mean in the numerator, it uses the sum of the products of the number of standard deviations away from the mean. These number-of-standard-deviations-from-the-mean are called *z-scores*. If a value has a z-score of 1.5, it is 1.5 standard deviations above the mean; if a value has a z-score of -2, then it is 2 standard deviations below the mean.

Pearson's correlation coefficient is usually denoted by r and its equation is given as follows:

$$r = \frac{\sum (x - \bar{x})(y - \bar{y})}{(n-1)s_x s_y}$$

which is the covariance divided by the product of the two variables' standard deviation.

An important consequence of using standardized z-scores instead of the magnitude of distance from the mean is that changing the variability in one variable does not change the correlation coefficient. Now you can meaningfully compare values using two different scales or even two different distributions. The correlation between weight/height-in-inches and weight/height-in-centimeters will now be identical, because multiplication with `2.54` will not change the z-scores of each height.

```
> cor(women$height, women$weight)
[1] 0.9954948
> cor(women$height*2.54, women$weight)
[1] 0.9954948
```

Another important and helpful consequence of this standardization is that the measure of correlation will always range from -1 to 1. A Pearson correlation coefficient of 1 will denote a perfectly positive (linear) relationship, a r of -1 will denote a perfectly negative (linear) relationship, and a r of 0 will denote no (linear) relationship.

Why the *linear* qualification in parentheses, though?

Intuitively, the correlation coefficient shows how well two variables are described by the straight line that fits the data most closely; this is called a *regression* or *trend line*. If there is a strong relationship between two variables, but the relationship is not linear, it cannot be represented accurately by Pearson's r. For example, the correlation between 1 to 100 and 100 to 200 is 1 (because it is perfectly linear), but a cubic relationship is not:

```
> xs <- 1:100
> cor(xs, xs+100)
[1] 1
> cor(xs, xs^3)
[1] 0.917552
```

It is still about 0.92, which is an extremely strong correlation, but not the 1 that you should expect from a perfect correlation.

So Pearson's r assumes a linear relationship between two variables. There are, however, other correlation coefficients that are more tolerant of non-linear relationships. Probably the most common of these is *Spearman's rank coefficient*, also called *Spearman's rho*.

Spearman's rho is calculated by taking the Pearson correlation not of the values, but of their ranks.

What's a rank?

When you assign ranks to a vector of numbers, the lowest number gets 1, the second lowest gets 2, and so on. The highest datum in the vector gets a rank that is equal to the number of elements in that vector.

In rankings, the magnitude of the difference in values of the elements is disregarded. Consider a race to a finish line involving three cars. Let's say that the winner in the first place finished at a speed three times that of the car in the second place, and the car in the second place beat the car in the third place by only a few seconds. The driver of the car that came first has a good reason to be proud of herself, but her rank, *1st place*, does not say anything about how she effectively *cleaned the floor* with the other two candidates.

Try using R's `rank` function on the vector `c(8, 6, 7, 5, 3, 0, 9)`. Now try it on the vector `c(8, 6, 7, 5, 3, -100, 99999)`. The rankings are the same, right?

When we use ranks instead, the pair that has the highest value on both the *x* and the *y* axis will be `c(1,1)`, even if one variable is a non-linear function (cubed, squared, logarithmic, and so on) of the other. The correlations that we just tested will both have Spearman rhos of 1, because cubing a value will not change its rank.

```
> xs <- 1:100
> cor(xs, xs+100, method="spearman")
[1] 1
> cor(xs, xs^3, method="spearman")
[1] 1
```

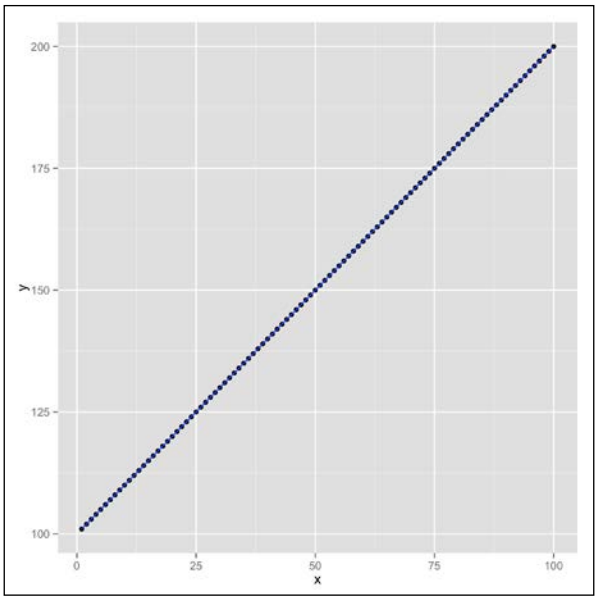

Figure 3.5: Scatterplot of y=x + 100 with regression line. r and rho are both 1

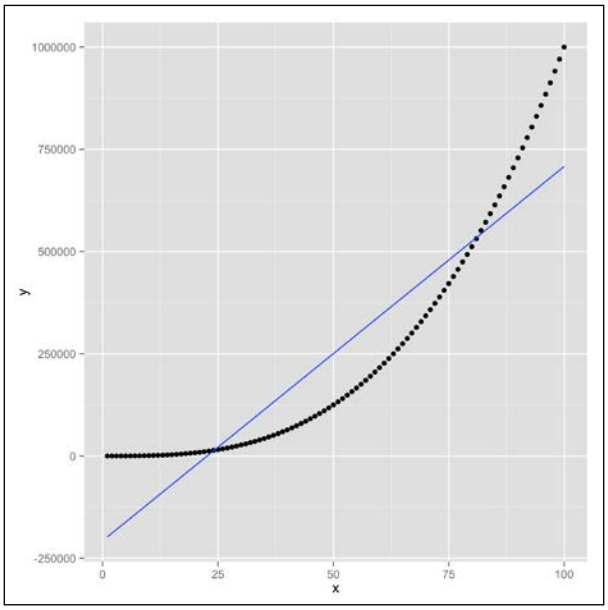

Figure 3.6: Scatterplot of $y = x^3$ with regression line. r is .92, but rho is 1

Let's use what we've learned so far to investigate the correlation between the weight of a car and the number of miles it gets to the gallon. Do you predict a negative relationship (the heavier the car, the lower the miles per gallon)?

```
> cor(mtcars$wt, mtcars$mpg)
[1] -0.8676594
```

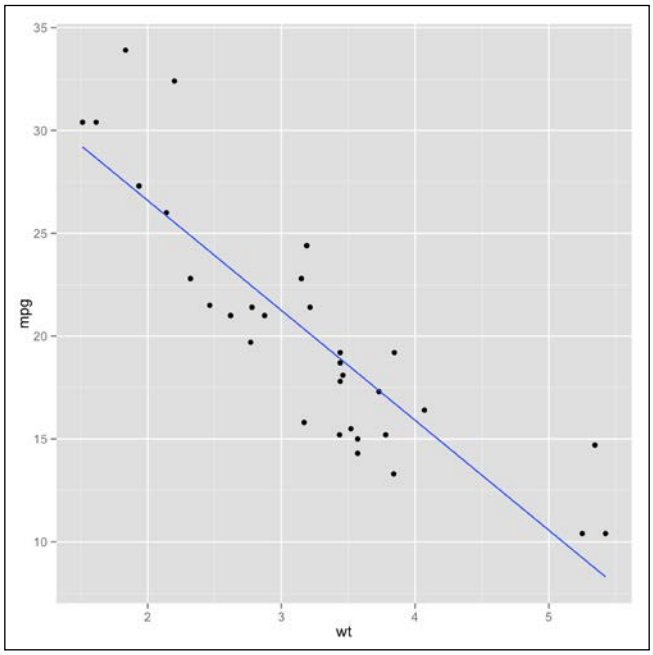

Figure 3.7: Scatterplot of the relationship between the weight of a car and its miles per gallon

That is a strong negative relationship. Although, in the preceding figure, note that the data points are more diffuse and spread around the regression line than in the other plots; this indicates a somewhat weaker relationship than we have seen thus far.

For an even weaker relationship, check out the correlation between wind speed and temperature in the `airquality` dataset as depicted in the following image:

```
> cor(airquality$Temp, airquality$Wind)
[1] -0.4579879
> cor(airquality$Temp, airquality$Wind, method="spearman")
[1] -0.4465408
```

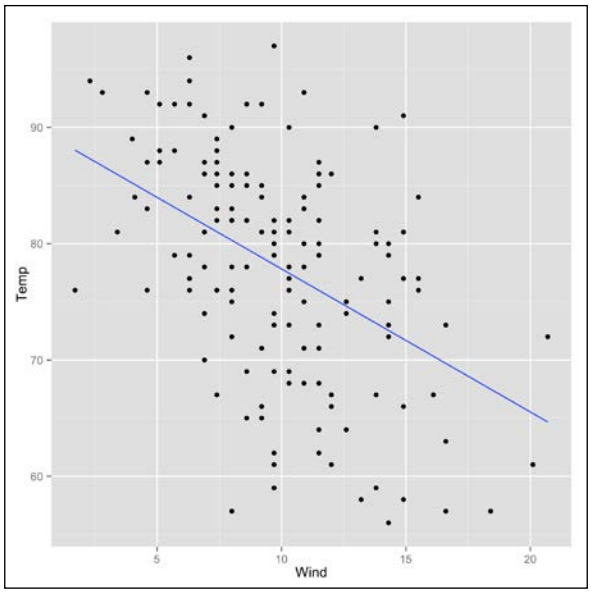

Figure 3.8: Scatterplot of the relationship between wind speed and temperature

Comparing multiple correlations

Armed with our new standardized coefficients, we can now effectively compare the correlations between different pairs of variables directly.

In data analysis, it is common to compare the correlations between all the numeric variables in a single dataset. We can do this with the `iris` dataset using the following R code snippet:

```
> # have to drop 5th column (species is not numeric)
> iris.nospecies <- iris[, -5]
> cor(iris.nospecies)
             Sepal.Length Sepal.Width Petal.Length Petal.Width
Sepal.Length    1.0000000  -0.1175698    0.8717538   0.8179411
Sepal.Width    -0.1175698   1.0000000   -0.4284401  -0.3661259
Petal.Length    0.8717538  -0.4284401    1.0000000   0.9628654
Petal.Width     0.8179411  -0.3661259    0.9628654   1.0000000
```

This produces a correlation matrix (when it is done with the covariance, it is called a *covariance matrix*). It is square (the same number of rows and columns) and symmetric, which means that the matrix is identical to its transposition (the matrix with the axes flipped). It is symmetrical, because there are two elements for each pair of variables on either side of the diagonal line of 1s. The diagonal line is all 1's, because every variable is perfectly correlated with itself. Which are the most highly (positively) correlated pairs of variables? What about the most negatively correlated?

Visualization methods

We are now going to see how we can create these kinds of visualizations on our own.

Categorical and continuous variables

We have seen that box plots are a great way of comparing the distribution of a continuous variable across different categories. As you might expect, box plots are very easy to produce using `ggplot2`. The following snippet produces the box-and-whisker plot that we saw earlier, depicting the relationship between the petal lengths of the different iris species in the `iris` dataset:

```
> library(ggplot)
> qplot(Species, Petal.Length, data=iris, geom="boxplot",
+       fill=Species)
```

First, we specify the variable on the x-axis (the iris species) and then the continuous variable on the y-axis (the petal length). Finally, we specify that we are using the iris dataset, that we want a box plot, and that we want to fill the boxes with different colors for each iris species.

Another fun way of comparing distributions between the different categories is by using an overlapping density plot:

```
> qplot(Petal.Length, data=iris, geom="density", alpha=I(.7),
+       fill=Species)
```

Here we need only specify the continuous variable, since the `fill` parameter will break down the density plot by species. The `alpha` parameter adds transparency to show more clearly the extent to which the distributions overlap.

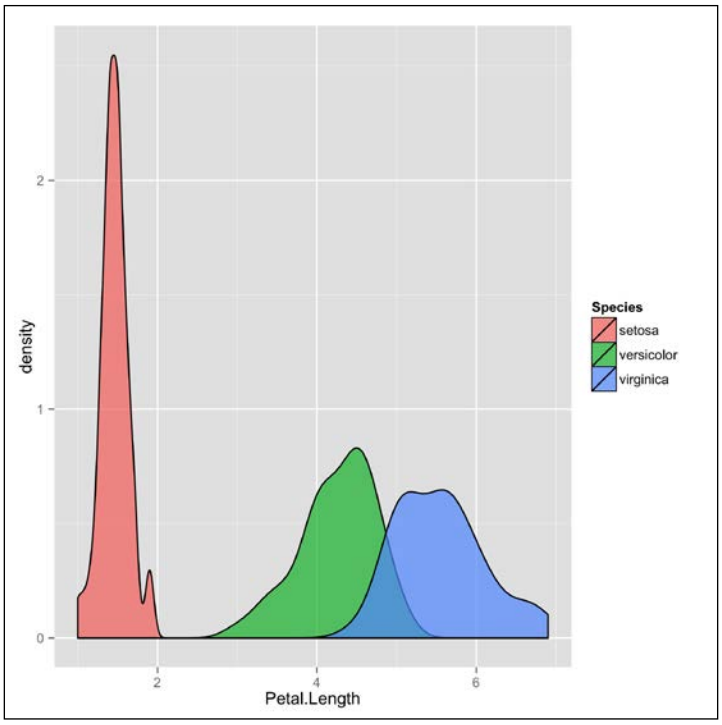

Figure 3.9: Overlapping density plot of petal length of iris flowers across species

If it is not the distribution you are trying to compare but some kind of single-value statistic (like standard deviation or sample counts), you can use the `by` function to get that value across all categories, and then build a bar plot where each category is a bar, and the heights of the bars represent that category's statistic. For the code to construct a bar plot, refer back to the last section in *Chapter 1, RefresheR*.

Two categorical variables

The visualization of categorical data is a grossly understudied domain and, in spite of some fairly powerful and compelling visualization methods, these techniques remain relatively unpopular.

My favorite method for graphically illustrating contingency tables is to use a *mosaic plot*. To make mosaic plots, we will need to install and load the **VCD** (**Visualizing Categorical Data**) package:

```
> # install.packages("vcd")
> library(vcd)
>
> ucba <- data.frame(UCBAdmissions)
> mosaic(Freq ~ Gender + Admit, data=ucba,
+        shade=TRUE, legend=FALSE)
```

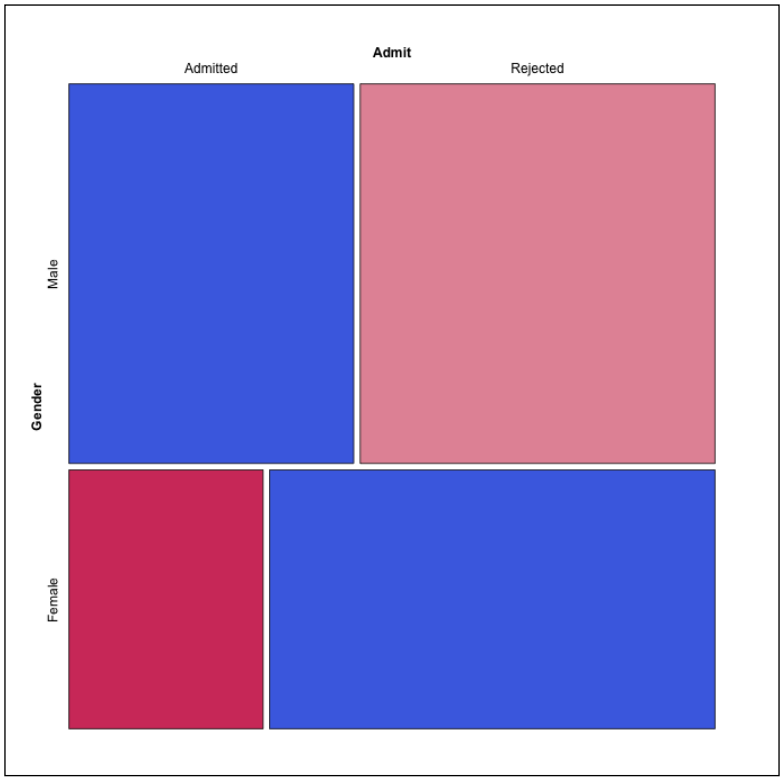

Figure 3.10: A mosaic plot of the UCBAdmissions dataset (across all departments)

The first argument to the mosaic function is a formula. This formula is meant to be read as: *display frequency broken down by gender and whether the applicant was admitted*. `shade=TRUE` adds a little life to the plot by adding colors to the boxes. The colors are actually very meaningful, as is the legend we opted not to show with the final parameter—but its meaning is beyond the scope of this section.

The mosaic plot represents each cell of a 2x2 contingency table as a tile; the area of the box is proportional to the number of observations in that cell. From this plot, we can easily tell that (a) *more men applied to UCB than women*, (b) *more applicants were rejected than accepted*, and (c) *women were rejected at a higher proportion than male applicants*.

You remember how this was misleading, right? Let's look at the mosaic plot for only department A:

```
> mosaic(Freq ~ Gender+Admit, data=ucba[ucba$Dept=="A",],
+         shade=TRUE, legend=FALSE)
```

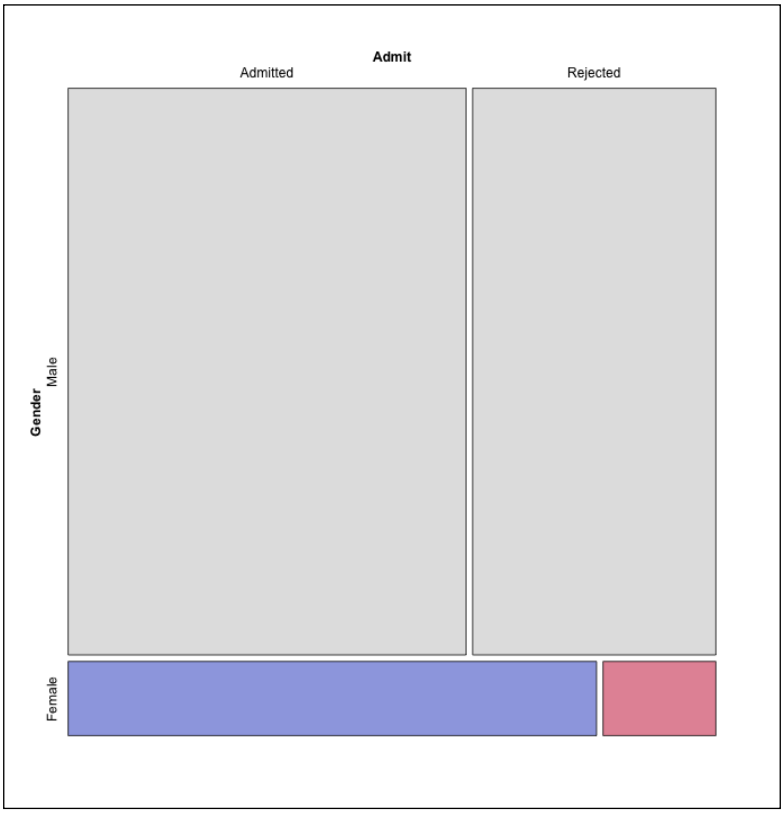

Figure 3.11: A mosaic plot of the UCBAdmissions dataset for department A

Hopefully, this plot makes the treachery of Simpson's paradox more apparent. Notice how there were far fewer female applicants than males, but the admission rates for the female applicants were much higher. Try visualizing the mosaic plots for the other departments by yourself!

Two continuous variables

The canonical way of displaying relationships between two continuous variables is via scatterplots. The scatterplot for the women's heights and weights that we saw earlier in this chapter was produced with the following R code snippet:

```
> qplot(height, weight, data=women, geom="point")
```

Whether you put `height` and `weight` first depends on which variable you want tied to the x-axis.

What about that fancy regression line?!, you ask frantically. `ggplot2` gracefully provides this feature with just a few extra characters. The scatterplot of the relationship between the weight of a car and its miles per gallon was produced as follows:

```
> qplot(wt, mpg, data=mtcars, geom=c("point", "smooth"),
+       method="lm", se=FALSE)'
```

Here, we are specifying that we want two kinds of geometric objects, `point` and `smooth`. The latter is responsible for the regression line. `method="lm"` tells `qplot` that we want to use a linear model to create the trend line.

If we leave out the method, `ggplot2` will choose a method automatically; in this case, it would default to a method of drawing a non-linear trend line called *LOESS*:

```
> qplot(wt, mpg, data=mtcars, geom=c("point", "smooth"), se=FALSE)
```

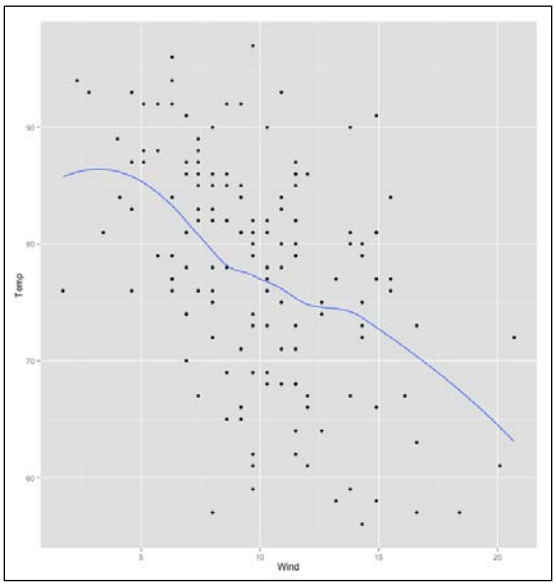

Figure 3.12: A scatterplot of the relationship between the weight of a car and its miles per gallon, and a trend-line smoothed with LOESS

The `se=FALSE` directive instructs `ggplot2` not to plot the estimates of the error. We will get to what this means in a later chapter.

More than two continuous variables

Finally, there is an excellent way to visualize correlation matrices like the one we saw with the `iris` dataset in the section *Comparing multiple correlations*. To do this, we have to install and load the `corrgram` package as follows:

```
> # install.packages("corrgram")
> library(corrgram)
>
> corrgram(iris, lower.panel=panel.conf, upper.panel=panel.pts)
```

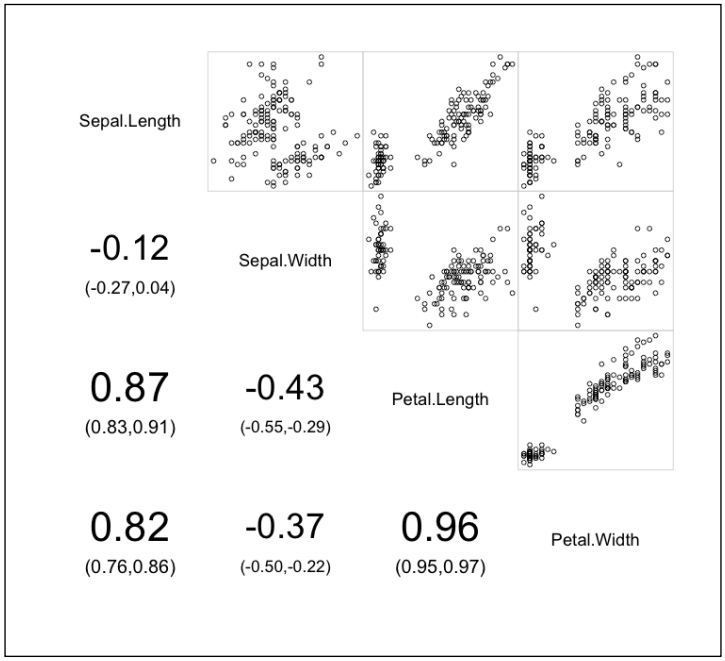

Figure 3.13: A corrgram of the iris data set's continuous variables

With corrgrams, we can exploit the fact the correlation matrices are symmetrical by packing in more information. On the lower left panel, we have the Pearson correlation coefficients (never mind the small ranges beneath each coefficient for now). Instead of repeating these coefficients for the upper right panel, we can show a small scatterplot there instead.

We aren't limited to showing the coefficients and scatterplots in our `corrgram`, though; there are many other options and configurations available:

```
> corrgram(iris, lower.panel=panel.pie, upper.panel=panel.pts,
+          diag.panel=panel.density,
+          main=paste0("corrgram of petal and sepal ",
+                      "measurements in iris data set"))
```

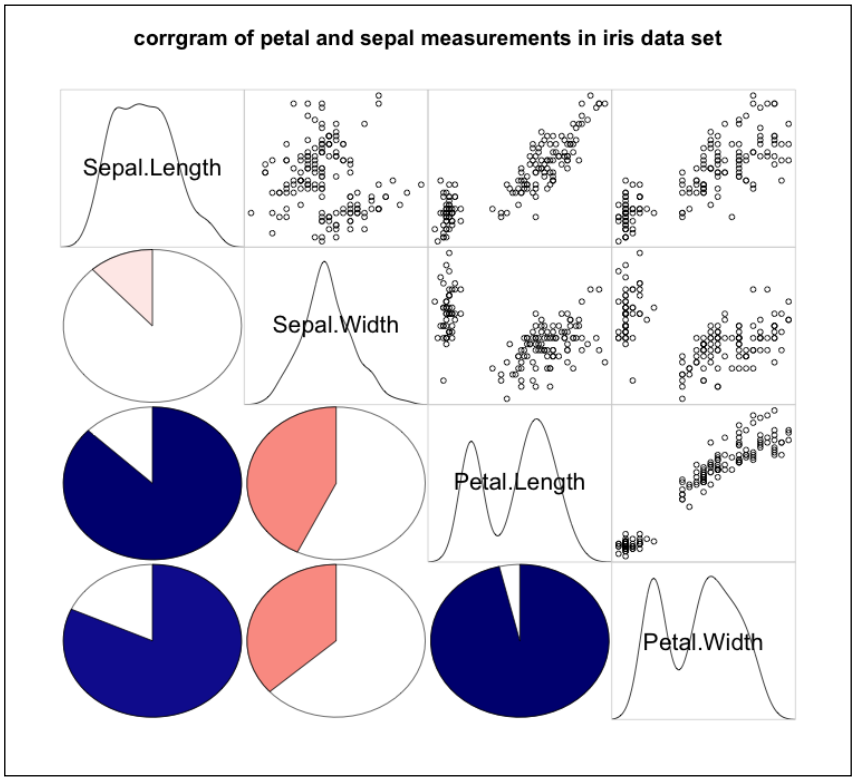

Figure 3.14: Another corrgram of the iris dataset's continuous variables

Notice that this time, we can overlay a density plot wherever there is a variable name (on the diagonal) —just to get a sense of the variables' shapes. More saliently, instead of text coefficients, we have pie charts in the lower-left panel. These pie charts are meant to graphically depict the strength of the correlations.

If the color of the pie is blue (or any shade thereof), the correlation is positive; the bigger the shaded area of the pie, the stronger the magnitude of the correlation. If, however, the color of the pie is red or a shade of red, the correlation is negative, and the amount of shading on the pie is proportional to the magnitude of the correlation.

To top it all off, we added the main parameter to set the title of the plot. Note the use of `paste0` so that I could split the title up into two lines of code.

To get a better sense of what `corrgram` is capable of, you can view a live demonstration of examples if you execute the following at the prompt:

```
> example(corrgram)
```

Exercises

Try out the following exercises to revise the concepts learned so far:

- Look at the documentation on cor with `help("cor")`. You can see, in addition to `"pearson"` and `"spearman"`, there is an option for `"kendall"`. Learn about *Kendall's tau*. Why, and under what conditions, is it considered better than Spearman's rho?
- For each species of iris, find the correlation coefficient between the sepal length and width. Are there any differences? How did we just combine two different types of the broad categories of bivariate analyses to perform a complex multivariate analysis?
- Download a dataset from the web, or find another built-into-R dataset that suits your fancy (using `library(help = "datasets")`). Explore relationships between the variables that you think might have some connection.
- Gustave Flaubert is well understood to be a classist misogynist and this, of course, influenced how he developed the character of Emma Bovary. However, it is not uncommon for the readers to identify and empathize with her, and they are often devastated by the book's conclusion. In fact, translator Geoffrey Wall asserts that Emma *dies in a pain that is exactly adjusted to the intensity of our preceding identification*.

  How can the fact that some sympathize with Emma be reconciled with Flaubert's apparent intention? In your response, assume a post-structuralist approach to authorial intent.

Summary

There were many new ideas introduced in this chapter, so kudos to you for making it through! You're well on the way to being able to tackle some extraordinarily interesting problems on your own!

To summarize, in this chapter, we learned that the relationships between two variables can be broken down into three broad categories.

For categorical/continuous variables, we learned how to use the `by` function to retrieve the statistics on the continuous variable for each category. We also saw how we can use box-and-whisker plots to visually inspect the distributions of the continuous variable across categories.

For categorical/categorical configurations, we used contingency and proportions tables to compare frequencies. We also saw how mosaic plots can help spot interesting aspects of the data that might be difficult to detect when just looking at the raw numbers.

For continuous/continuous data we discovered the concepts of covariance and correlations and explored different correlation coefficients with different assumptions about the nature of the bivariate relationship. We also learned how these concepts could be expanded to describe the relationship between more than two continuous variables. Finally, we learned how to use scatterplots and corrgrams to visually depict these relationships.

With this chapter, we've concluded the unit on exploratory data analysis, and we'll be moving on to *confirmatory data analysis* and *inferential statistics*.

4
Probability

It's time for us to put descriptive statistics down for the time being. It was fun for a while, but we're no longer content just determining the properties of observed data; now we want to start making deductions about data we haven't observed. This leads us to the realm of inferential statistics.

In data analysis, probability is used to quantify uncertainty of our deductions about unobserved data. In the land of inferential statistics, probability reigns queen. Many regard her as a harsh mistress, but that's just a rumor.

Basic probability

Probability measures the likeliness that a particular event will occur. When mathematicians (us, for now!) speak of an event, we are referring to a set of potential outcomes of an experiment, or trial, to which we can assign a probability of occurrence.

Probabilities are expressed as a number between 0 and 1 (or as a percentage out of 100). An event with a probability of 0 denotes an impossible outcome, and a probability of 1 describes an event that is certain to occur.

The canonical example of probability at work is a coin flip. In the coin flip event, there are two outcomes: the coin lands on heads, or the coin lands on tails. Pretending that coins never land on their edge (they almost never do), those two outcomes are the only ones possible. The sample space (the set of all possible outcomes), therefore, is {heads, tails}. Since the entire sample space is covered by these two outcomes, they are said to be collectively exhaustive.

The sum of the probabilities of collectively exhaustive events is always 1. In this example, the probability that the coin flip will yield heads or yield tails is 1; it is certain that the coin will land on one of those. In a fair and correctly balanced coin, each of those two outcomes is equally likely. Therefore, we split the probability equally among the outcomes: in the event of a coin flip, the probability of obtaining heads is 0.5, and the probability of tails is 0.5 as well. This is usually denoted as follows:

$$P(heads) = 0.5$$

The probability of a coin flip yielding either heads or tails looks like this:

$$P(heads \cup tails) = 1$$

And the probability of a coin flip yielding both heads and tails is denoted as follows:

$$P(heads \cap tails) = 0$$

The two outcomes, in addition to being collectively exhaustive, are also mutually exclusive. This means that they can never co-occur. This is why the probability of heads and tails is 0; it just can't happen.

The next obligatory application of beginner probability theory is in the case of rolling a standard six-sided die. In the event of a die roll, the sample space is `{1, 2, 3, 4, 5, 6}`. With every roll of the die, we are sampling from this space. In this event, too, each outcome is equally likely, except now we have to divide the probability across six outcomes. In the following equation, we denote the probability of rolling a 1 as P(1):

$$P(1) = 1/6$$

Rolling a 1 or rolling a 2 is not collectively exhaustive (we can still roll a 3, 4, 5, or 6), but they are mutually exclusive; we can't roll a 1 and 2. If we want to calculate the probability of either one of two mutually exclusive events occurring, we add the probabilities:

$$P(1 \cup 2) = P(1) + P(2) = 1/3$$

While rolling a 1 or rolling a 2 aren't mutually exhaustive, rolling 1 and not rolling a 1 are. This is usually denoted in this manner:

$$P\left(1 \cup \neg 1\right) = 1$$

These two events—and all events that are both collectively exhaustive and mutually exclusive—are called complementary events.

Our last pedagogical example in the basic probability theory is using a deck of cards. Our deck has 52 cards—4 for each number from 2 to 10 and 4 each of Jack, Queen, King, and Ace (no Jokers!). Each of these 4 cards belong to one suit, either a Heart, Club, Spade or Diamond. There are, therefore, 13 cards in each suit. Further, every Heart and Diamond card is colored red, and every Spade and Club are black. From this, we can deduce the following probabilities for the outcome of randomly choosing a card:

$$P\left(Ace\right) = \frac{4}{52}$$

$$P\left(Queen \cup King\right) = \frac{8}{52}$$

$$P\left(Black\right) = \frac{26}{52}$$

$$P\left(Club\right) = \frac{13}{52}$$

$$P\left(Club \cup Heart \cup Spade\right) = \frac{39}{52}$$

$$P\left(Club \cup Heart \cup Spade \cup Diamond\right) = 1\left(collectively\ exhaustive\right)$$

What, then, is the probability of getting a black card and an Ace? Well, these events are *conditionally independent*, meaning that the probability of either outcome does not affect the probability of the other. In cases like these, the probability of event A and event B is the product of the probability of A and the probability of B. Therefore:

$$P\left(Black \cap Ace\right) = 26/52 * 4/52 = 2/52$$

Intuitively, this makes sense, because there are two black Aces out of a possible 52.

What about the probability that we choose a red card and a Heart? These two outcomes are not conditionally independent, because knowing that the card is red has a bearing on the likelihood that the card is also a Heart. In cases like these, the probability of event A and B is denoted as follows:

$$P(A \cap B) = P(A)P(B \mid A) \; or \; P(B)P(A \mid B)$$

Where `P(A|B)` means *the probability of A given B*. For example, if we represent A as drawing a Heart and B as drawing a red card, `P(A | B)` means *what's the probability of drawing a heart if we know that the card we drew was red?*. Since a red card is equally likely to be a Heart or a Diamond, `P(A|B)` is 0.5. Therefore:

$$P(Heart \cap Red) = P(Red)P(Heart \mid Red) = \frac{26}{52} * \frac{1}{2} = \frac{1}{4}$$

In the preceding equation, we used the form `P(B) P(A|B)`. Had we used the form `P(A) P(B|A)`, we would have got the same answer:

$$P(Heart \cap Red) = P(Heart)P(Red \mid Heart) = \frac{13}{52} * 1 = \frac{13}{52} = \frac{1}{4}$$

So, these two forms are equivalent:

$$P(B)P(A \mid B) = P(A)P(B \mid A)$$

For kicks, let's divide both sides of the equation by P(B). That yields the following equivalence:

$$P(A \mid B) = \frac{P(A)P(B \mid A)}{P(B)}$$

This equation is known as *Bayes' Theorem*. This equation is very easy to derive, but its meaning and influence is profound. In fact, it is one of the most famous equations in all of mathematics.

Bayes' Theorem has been applied to and proven useful in an enormous amount of different disciplines and contexts. It was used to help crack the German Enigma code during World War II, saving the lives of millions. It was also used recently, and famously, by Nate Silver to help correctly predict the voting patterns of 49 states in the 2008 US presidential election.

At its core, Bayes' Theorem tells us how to update the probability of a hypothesis in light of new evidence. Due to this, the following formulation of Bayes' Theorem is often more intuitive:

$$P(H \mid E) = \frac{P(E \mid H)P(H)}{P(E)}$$

where `H` is the hypothesis and `E` is the evidence.

Let's see an example of Bayes' Theorem in action!

There's a hot new recreational drug on the scene called *Allighate* (or *Ally* for short). It's named as such because it makes its users go wild and act like an alligator. Since the effect of the drug is so deleterious, very few people actually take the drug. In fact, only about 1 in every thousand people (0.1%) take it.

Frightened by fear-mongering late-night news, Daisy Girl, Inc., a technology consulting firm, ordered an Allighate testing kit for all of its 200 employees so that it could offer treatment to any employee who has been using it. Not sparing any expense, they bought the best kit on the market; it had 99% sensitivity and 99% specificity. This means that it correctly identified drug users 99 out of 100 times, and only falsely identified a non-user as a user once in every 100 times.

When the results finally came back, two employees tested positive. Though the two denied using the drug, their supervisor, Ronald, was ready to send them off to get help. Just as Ronald was about to send them off, Shanice, a clever employee from the statistics department, came to their defense.

Ronald incorrectly assumed that each of the employees who tested positive were using the drug with 99% certainty and, therefore, the chances that both were using it was 98%. Shanice explained that it was actually far more likely that neither employee was using Allighate.

How so? Let's find out by applying Bayes' theorem!

Let's focus on just one employee right now; let H be the hypothesis that one of the employees is using Ally, and E represent the evidence that the employee tested positive.

$$P(\textit{Ally User} \mid \textit{Positive Test}) = \frac{P(\textit{Positive Test} \mid \textit{Ally User})\,P(\textit{Ally User})}{P(\textit{Testing positive}, \textit{in general})}$$

We want to solve the left side of the equation, so let's plug in values. The first part of the right side of the equation, `P(Positive Test | Ally User)`, is called the likelihood. The probability of testing positive if you use the drug is 99%; this is what tripped up Ronald—and most other people when they first heard of the problem. The second part, `P(Ally User)`, is called the prior. This is our belief that any one person has used the drug before we receive any evidence. Since we know that only .1% of people use Ally, this would be a reasonable choice for a prior. Finally, the denominator of the equation is a normalizing constant, which ensures that the final probability in the equation will add up to one of all possible hypotheses. Finally, the value we are trying to solve, `P(Ally user | Positive Test)`, is the posterior. It is the probability of our hypothesis updated to reflect new evidence.

$$P(\textit{Ally User} \mid \textit{Positive Test}) = \frac{.99 * .001}{P(\textit{Testing positive}, \textit{in general})}$$

In many practical settings, computing the normalizing factor is very difficult. In this case, because there are only two possible hypotheses, being a user or not, the probability of finding the evidence of a positive test is given as follows:

$$P(\textit{Testing positive} \mid \textit{Ally User})\,P(\textit{Ally User}) + P(\textit{Testing positive} \mid \textit{Not an Ally User})\,P(\textit{Not an Ally User})$$

Which is: (.99 * .001) + (.01 * .999) = 0.01098

Plugging that into the denominator, our final answer is calculated as follows:

$$P(\textit{Ally User} \mid \textit{Positive Test}) = \frac{.99 * .001}{0.01098} = 0.090164$$