An R companion to Statistical Thinking for the 21st Century

Draft: 2024-03-31

Contents

Preface

This book is a companion to Statistical Thinking for the 21st Century, an open source statistical textbook. It focuses on the use of the R statistical programming language for statistics and data analysis.

0.1 Why R?

In my statistics course, students learn to analyze data hands-on using the R language. The question "Why R?" could be interpreted to mean "Why R instead of a graphical software package like (insert name here)?". After all, most of the students who enroll in my class have never programmed before, so teaching them to program is going to take away from instruction in statistical concepts. My answer is that I think that the best way to learn statistical tools is to work directly with data, and that working with graphical packages insulates one from the data and methods in way that impedes true understanding. In addition, for many of the students in my class this may be the only course in which they are exposed to programming; given that programming is an essential ability in a growing number of academic fields, I think that providing these students with basic programming literacy is critical to their future success, and will hopefully inspire at least a few of them to learn more.

The question could also be interpreted to mean "Why R instead of (insert language here)?". On this question I am much more conflicted, because I deeply dislike R as a programming language (I greatly prefer to use Python for my own work). Why then do I use R? The first reason is that R has become the "lingua franca" for statistical analysis. There are a number of tools that I use in this book (such as the linear modeling tools in the `lme4`

package and the Bayes factor tools in the `BayesFactor` package) that are simply not available in other languages.

The second reason is that the free Rstudio software makes using R relatively easy for new users. In particular, I like the RMarkdown Notebook feature that allows the mixing of narrative and executable code with integrated output. It's similar in spirit to the Jupyter notebooks that many of us use for Python programming, but I find it easier to deal with because you edit it as a plain text file, rather than through an HTML interface. In my class, I give students a skeleton RMarkdown file for problem sets, and they submit the file with their solution added, which I then score using a set of automated grading scripts.

That said, there are good reasons to prefer a real programming language over R; my preferred language is Python, and a parallel Python companion to this one is currently under development.

0.2 The golden age of data

Throughout this book I have tried when possible to use examples from real data. This is now very easy because we are swimming in open datasets, as governments, scientists, and companies are increasingly making data freely available. I think that using real datasets is important because it prepares students to work with real data rather than toy datasets, which I think should be one of the major goals of statistical training. It also helps us realize (as we will see at various points throughout the book) that data don't always come to us ready to analyze, and often need *wrangling* to help get them into shape. Using real data also shows that the idealized statistical distributions often assumed in statistical methods don't always hold in the real world – for example, as we will see in Chapter 2, distributions of some real-world quantities (like the number of friends on Facebook) can have very long tails that can break many standard assumptions.

I apologize up front that the datasets are heavily US-centric. This is primarily because the best dataset for many of the demonstrations is the National Health and Nutrition Examination Surveys (NHANES) dataset that is available as an R package, and because many of the other complex datasets included in R (such as those in the `fivethirtyeight` package) are also based in the US. If

you have suggestions for datasets from other regions, please pass them along to me!

0.3 An open source book

This book is meant to be a living document, which is why its source is available online at https://github.com/statsthinking21/statsthinking21-R. If you find any errors in the book or want to make a suggestion for how to improve it, please open an issue on the Github site. Even better, submit a pull request with your suggested change.

0.4 Acknowledgements

Thanks to everyone who has contributed to this project.

Chapter 1

Introduction to R

R is a computer programming language that was purpose-built for statistical data analysis. The name "R" is a play on the names of the two authors of the software package (Ross Ihaka and Robert Gentleman) as well as an homage to an older statistical software package called "S". R has become one of the most popular programming languages for statistical analysis and "data science", which refers to the broader enterprise of working with and analyzing data. Unlike general-purpose programming languages such as Python or Java, R is purpose-built for statistics. That doesn't mean that you can't do more general things with it, but the place where it really shines is in data analysis and statistics.

1.1 Why programming is hard to learn

Programming a computer is a skill, just like playing a musical instrument or speaking a second language. And just like those skills, it takes a lot of work to get good at it — the only way to acquire a skill is through practice. There is nothing special or magical about people who are experts, other than the quality and quantity of their experience! However, not all practice is equally effective. A large amount of psychological research has shown that practice needs to be *deliberate*, meaning that it focuses on developing the specific skills that one needs to perform the activity, at a level that is always pushing one's ability.

If you have never programmed before, then it's going to seem hard, just as it would seem hard for a native English speaker to start speaking Mandarin. However, just as a beginning guitarist needs to learn to play their scales, we will teach you how to perform the basics of programming, which you can then use to do more powerful things.

One of the most important aspects of computer programming is that you can try things to your heart's content; the worst thing that can happen is that the program will crash. Trying new things and making mistakes is one of the keys to learning.

The hardest part of programming is figuring out why something didn't work, which we call *debugging*. In programming, things are going to go wrong in ways that are often confusing and opaque. Every programmer has a story about spending hours trying to figure out why something didn't work, only to realize that the problem was completely obvious. The more practice you get, the better you will get at figuring out how to fix these errors. But there are a few strategies that can be helpful.

1.1.1 Use the web

In particular, you should take advantage of the fact that there are millions of people programming in R around the world, so nearly any error message you see has already been seen by someone else. Whenever I experience an error that I don't understand, the first thing that I do is to copy and paste the error message into a search engine. Often this will provide several links discussing the problem and the ways that people have solved it.

1.1.2 Rubber duck debugging

The idea behind *rubber duck debugging* is to pretend that you are trying to explain what your code is doing to an inanimate object, like a rubber duck. Often, the process of explaning it aloud is enough to help you find the problem.

1.2 Using RStudio

When I am using R in my own work, I generally use a free software package called RStudio, which provides a number of nice tools for working with R. In particular, RStudio provides the ability to create "notebooks" that mix together R code and text (formatted using the Markdown text formatting system). In fact, this book is written using exactly that system! You can see the R code used to generate this book here.

In some cases it m ay not be possible to install RStudio on one's computer (for example, if that computer is a Chromebook). Another alteranative is to use an online platform for coding. Two popular platforms that support R are Google's Colab and Kaggle Kernels.

1.3 Getting started with R

When we work with R, we often do this using a *command line* in which we type commands and it responds to those commands. In the simplest case, if we just type in a number, it will simply respond with that number. Go into the R console and type the number 3. You should see something like this:

```
> 3
[1] 3
```

The `>` symbol is the *command prompt*, which is prompting you to type something in. The next line (`[1] 3`) is R's answer. Let's try something a bit more complicated:

```
> 3 + 4
[1] 7
```

R spits out the answer to whatever you type in, as long as it can figure it out. Now let's try typing in a word:

```
> hello
Error: object 'hello' not found
```

What? Why did this happen? When R encounters a letter or word, it assumes that it is referring to the name of a *variable* — think of `X` from high school algebra. We will return to variables in a little while, but if we want R to print

out the word *hello* then we need to contain it in quotation marks, telling R that it is a *character string*.

```
> "hello"
[1] "hello"
```

There are many types of variables in R. You have already seen two examples: integers (like the number 3) and character strings (like the word "hello"). Another important one is *real numbers*, which are the most common kind of numbers that we will deal with in statistics, which span the entire number line including the spaces in between the integers. For example:

```
> 1/3
[1] 0.33
```

In reality the result should be 0.33 followed by an infinite number of threes, but R only shows us two decimal points in this example.

Another kind of variable is known as a *logical* variable, because it is based on the idea from logic that a statement can be either true or false. In R, these are capitalized (`TRUE` and `FALSE`).

To determine whether a statement is true or not, we use *logical operators*. You are already familiar with some of these, like the greater-than (`>`) and less-than (`<`) operators.

```
> 1 < 3
[1] TRUE
> 2 > 4
[1] FALSE
```

Often we want to know whether two numbers are equal or not equal to one another. There are special operators in R to do this: `==` for equals, and `!=` for not-equals:

```
> 3 == 3
[1] TRUE
> 4 != 4
[1] FALSE
```

1.4 Variables

A *variable* is a symbol that stands for another value (just like "X" in algebra). We can create a variable by assigning a value to it using the `<-` operator. If we then type the name of the variable R will print out its value.

```
> x <- 4
> x
[1] 4
```

The variable now stands for the value that it contains, so we can perform operations on it and get the same answer as if we used the value itself.

```
> x + 3
[1] 7
> x == 5
[1] FALSE
```

We can change the value of a variable by simply assigning a new value to it.

```
> x <- x + 1
> x
[1] 5
```

A note: You can also use the equals sign `=` instead of the `<-`

1.5 Functions

A *function* is an operator that takes some input and gives an output based on the input. For example, let's say that we have a number, and we want to determine its absolute value. R has a function called `abs()` that takes in a number and outputs its absolute value:

```
> x <- -3
> abs(x)
[1] 3
```

Most functions take an input like the `abs()` function (which we call an *argument*), but some also have special keywords that can be used to change how the function works. For example, the `rnorm()` function generates random numbers from a normal distribution (which we will learn more about later).

Have a look at the help page for this function by typing `help(rnorm)` in the console, which will cause a help page to appear below. The section of the help page for the `rnorm()` function shows the following:

```
rnorm(n, mean = 0, sd = 1)

Arguments

n     number of observations.

mean    vector of means.

sd   vector of standard deviations.
```

You can also obtain some examples of how the function is used by typing `example(rnorm)` in the console.

We can see that the rnorm function has two arguments, *mean* and *sd*, that are shown to be equal to specific values. This means that those values are the *default* settings, so that if you don't do anything, then the function will return random numbers with a mean of 0 and a standard deviation of 1. The other argument, *n*, does not have a default value. Try typing in the function `rnorm()` with no arguments and see what happens — it will return an error telling you that the argument "n" is missing and does not have a default value.

If we wanted to create random numbers with a different mean and standard deviation (say mean == 100 and standard deviation == 15), then we could simply set those values in the function call. Let's say that we would like 5 random numbers from this distribution:

```
> my_random_numbers <- rnorm(5, mean=100, sd=15)
> my_random_numbers
[1] 104 115 101  97 115
```

You will see that I set the variable to the name `my_random_numbers`. In general, it's always good to be as descriptive as possible when creating variables; rather than calling them x or y, use names that describe the actual contents. This will make it much easier to understand what's going on once things get more complicated.

1.6 Vectors

You may have noticed that the `my_random_numbers` created above wasn't like the variables that we had seen before — it contained a number of values in it. We refer to this kind of variable as a *vector*.

If you want to create your own new vector, you can do that using the `c()` function:

```
> my_vector <- c(4, 5, 6)
> my_vector
[1] 4 5 6
```

You can access the individual elements within a vector by using square brackets along with a number that refers to the location within the vector. These *index* values start at 1, which is different from many other programming languages that start at zero. Let's say we want to see the value in the second place of the vector:

```
> my_vector[2]
[1] 5
```

You can also look at a range of positions, by putting the start and end locations with a colon in between:

```
> my_vector[2:3]
[1] 5 6
```

You can also change the values of specific locations using the same indexing:

```
> my_vector[3] <- 7
> my_vector
[1] 4 5 7
```

1.7 Math with vectors

You can apply mathematical operations to the elements of a vector just as you would with a single number:

```
> my_vector <- c(4, 5, 6)
> my_vector_times_ten <- my_vector * 10
> my_vector_times_ten
```

```
[1] 40 50 60
```

You can also apply mathematical operations on pairs of vectors. In this case, each matching element is used for the operation.

```
> my_first_vector <- c(1, 2, 3)
> my_second_vector <- c(10, 20, 20)
> my_first_vector + my_second_vector
[1] 11 22 23
```

We can also apply logical operations across vectors; again, this will return a vector with the operation applied to the pairs of values at each position.

```
> vector_a <- c(1, 2, 3)
> vector_b <- c(1, 2, 4)
> vector_a == vector_b
[1]  TRUE  TRUE FALSE
```

Most functions will work with vectors just as they would with a single number. For example, let's say we wanted to obtain the trigonometric sine for each of a set of values. We could create a vector and pass it to the `sin()` function, which will return as many sine values as there are input values:

```
> my_angle_values <- c(0, 1, 2)
> my_sin_values <- sin(my_angle_values)
> my_sin_values
[1] 0.00 0.84 0.91
k
```

1.8 Data Frames

Often in a dataset we will have a number of different variables that we want to work with. Instead of having a different named variable that stores each one, it is often useful to combine all of the separate variables into a single package, which is referred to as a *data frame*.

If you are familiar with a spreadsheet (say from Microsoft Excel) then you already have a basic understanding of a data frame.
Let's say that we have values of price and mileage for three different types of

cars. We could start by creating a variable for each one, making sure that the three cars are in the same order for each of the variables:

```
car_model <- c("Ford Fusion", "Hyundai Accent", "Toyota Corolla")
car_price <- c(25000, 16000, 18000)
car_mileage <- c(27, 36, 32)
```

We can then combine these into a single data frame, using the `data.frame()` function. I like to use "_df" in the names of data frames just to make clear that it's a data frame, so we will call this one "cars_df":

```
cars_df <- data.frame(model=car_model, price=car_price, mileage=car_mileage)
```

We can view the data frame by using the `View()` function:

```
View(cars_df)
```

Which will present a view of the data frame much like a spreadsheet, as shown in Figure 1.1:

	model	price	mileage
1	Ford Fusion	25000	27
2	Hyundai Accent	16000	36
3	Toyota Corolla	18000	32

Figure 1.1: A view of the cars data frame generated by the View() function.

Each of the columns in the data frame contains one of the variables, with the name that we gave it when we created the data frame. We can access each of those columns using the `$` operator. For example, if we wanted to access the mileage variable, we would combine the name of the data frame with the name of the variable as follows:

```
> cars_df$mileage
[1] 27 36 32
```

This is just like any other vector, in that we can refer to its individual values using square brackets as we did with regular vectors:

```
> cars_df$mileage[3]
[1] 32
```

In some of the examples in the book, you will see something called a *tibble*; this is basically a souped-up version of a data frame, and can be treated mostly in the same way.

1.9 Using R libraries

Many of the useful features in R are not contained in the primary R package, but instead come from *libraries* that have been developed by various members of the R community. For example, the `ggplot2` package provides a number of features for visualizing data, as we will see in a later chapter. Before we can use a package, we need to install it on our system, using the `install.packages()` function:

```
> install.packages("ggplot2")
trying URL 'https://cran.rstudio.com/...
Content type 'application/x-gzip' length 3961383 bytes (3.8 MB)
==================================================
downloaded 3.8 MB


The downloaded binary packages are in
    /var/folders/.../downloaded_packages
```

This will automatically download the package from the Comprehensive R Archive Network (CRAN) and install it on your system. Once it's installed, you can then load the library using the `library()` function:

```
> library(ggplot2)
```

After loading the function, you can now access all of its features. If you want to learn more about its features, you can find them using the help function:

```
> help(ggplot2)
```

1.10 Working with data files

When we are doing statistics, we often need to load in the data that we will analyze. Those data will live in a file on one's computer or on the internet. For this example, let's use a file that is hosted on the internet, which contains the gross domestic product (GDP) values for a number of countries around the world. This file is stored as *comma-delimited text*, meaning that the values for each of the variables in the dataset are separate by commas. There are three variables: the relative rank of the countries, the name of the country, and its GDP value. Here is what the first few lines of the file look like:

```
Rank,Country,GDP
1,Liechtenstein,141100
2,Qatar,104300
3,Luxembourg,81100
```

We can load a comma-delimited text file into R using the `read.csv()` function, which will accept either the location of a file on one's computer, or a URL for files that are located on the web:

```
url < 'https://raw.githubusercontent.com/psych10/psych10/master/notebooks/Session03-I
gdp_df <- read.csv(url)
```

Once you have done this, take a look at the data frame using the `View()` function, and make sure that it looks right — it should have a column for each of the three variables.

Let's say that we wanted to create a new file, which contained GDP values in Euros rather than US Dollars. We use today's exchange rate, which is 1 USD == 0.90 Euros. To convert from Dollars to Euros, we simply multiple the GDP values by the exchange rate, and assign those values to a new variable within the data frame:

```
> exchange_rate <- 0.9
> gdp_df$GDP_euros <- gdp_df$GDP * exchange_rate
```

You should now see a new variable within the data frame, called "GDP_euros" which contains the new values. Now let's save this to a comma-delimited text file on our computer called "gdp_euro.csv". We do this using the `write.table()` command.

```
> write.table(gdp_df, file='gdp_euro.csv')
```

This file will be created with the working directory that RStudio is using. You can find this directory using the `getwd()` function:

```
> getwd()
[1] "/Users/me/MyClasses/Psych10/LearningR"
```

1.11 Learning objectives

Having finished this chapter, you should be able to:

- Interact with an RMarkdown notebook in RStudio
- Describe the difference between a variable and a function
- Describe the different types of variables
- Create a vector or data frame and access its elements
- Install and load an R library
- Load data from a file and view the data frame

1.12 Suggested readings and videos

There are many online resources for learning R. Here are a few:

- Datacamp: Offers free online courses for many aspects of R programming
- A Student's Guide to R
- R for cats: A humorous introduction to R programming
- aRrgh: a newcomer's (angry) guide to R
- Quick-R
- RStudio Cheat Sheets: Quick references for many different aspects of R programming
- tidverse Style Guide: Make your code beautiful and reader-friendly!
- R for Data Science: This free online book focuses on working with data in R.
- Advanced R: This free online book by Hadley Wickham will help you get to the next level once your R skills start to develop.
- R intro for Python users: Used Python before? Check this out for a guide on how to transition to R.

Chapter 2

Summarizing data using R (with Lucy King)

This chapter will introduce you to how to summarize data using R, as well as providing an introduction to a popular set of R tools known as the "Tidyverse."

Before doing anything else we need to load the libraries that we will use in this notebook.

```
library(tidyverse)
library(cowplot)
library(knitr)
set.seed(123456)
opts_chunk$set(tidy.opts=list(width.cutoff=80))
options(tibble.width = 60)
```

We will use the NHANES dataset for several of our examples, so let's load the library that contains the data.

```
# load the NHANES data library
# first unload it if it's already loaded, to make sure
# we have a clean version
rm('NHANES')
library(NHANES)
dim(NHANES)
```

```
## [1] 10000    76
```

2.1 Introduction to the Tidyverse

In this chapter we will introduce a way of working with data in R that is often referred to as the "Tidyverse." This comprises a set of packages that provide various tools for working with data, as well as a few special ways of using those functions

2.1.1 Making a data frame using tibble()

The tidyverse provides its own version of a data frame, which is known as a *tibble*. A tibble is a data frame but with some smart tweaks that make it easier to work with, expecially when using functions from the tidyverse. See here for more information on the function `tibble()`: https://r4ds.had.co.nz/tibbles.html

```
# first create the individual variables
n <- c("russ", "lucy", "jaclyn", "tyler")
x <- c(1, 2, 3, 4)
y <- c(4, 5, 6, 7)
z <- c(7, 8, 9, 10)

# create the data frame
my_data_frame <-
  tibble(
    n, #list each of your columns in the order you want them
    x,
    y,
    z
  )

my_data_frame
```

```
## # A tibble: 4 x 4
##   n          x     y     z
##   <chr>  <dbl> <dbl> <dbl>
## 1 russ       1     4     7
```

```
## 2 lucy       2     5     8
## 3 jaclyn     3     6     9
## 4 tyler      4     7    10
```

Take a quick look at the properties of the data frame using `glimpse()`:

```
glimpse(my_data_frame)
```

```
## Rows: 4
## Columns: 4
## $ n <chr> "russ", "lucy", "jaclyn", "tyler"
## $ x <dbl> 1, 2, 3, 4
## $ y <dbl> 4, 5, 6, 7
## $ z <dbl> 7, 8, 9, 10
```

2.1.2 Selecting an element

There are various ways to access the contents within a data frame.

2.1.2.1 Selecting a row or column by name

```
my_data_frame$x
```

```
## [1] 1 2 3 4
```

The first index refers to the row, the second to the column.

```
my_data_frame[1, 2]
```

```
## # A tibble: 1 x 1
##       x
##   <dbl>
## 1     1
```

```
my_data_frame[2, 3]
```

```
## # A tibble: 1 x 1
##       y
##   <dbl>
## 1     5
```

2.1.2.2 Selecting a row or column by index

```
my_data_frame[1, ]
```

```
## # A tibble: 1 x 4
##   n         x     y     z
##   <chr> <dbl> <dbl> <dbl>
## 1 russ      1     4     7
```

```
my_data_frame[, 1]
```

```
## # A tibble: 4 x 1
##   n
##   <chr>
## 1 russ
## 2 lucy
## 3 jaclyn
## 4 tyler
```

2.1.2.3 Select a set of rows

```
my_data_frame %>%
  slice(1:2)
```

```
## # A tibble: 2 x 4
##   n         x     y     z
##   <chr> <dbl> <dbl> <dbl>
## 1 russ      1     4     7
## 2 lucy      2     5     8
```

`slice()` is a function that selects out rows based on their row number.

You will also notice something we haven't discussed before: `%>%`. This is called a "pipe", which is commonly used within the tidyverse; you can read more here. A pipe takes the output from one command and feeds it as input to the next command. In this case, simply writing the name of the data frame (my_data_frame) causes it to be input to the `slice()` command following the pipe. The benefit of pipes will become especially apparent when we want to start stringing together multiple data processing operations into a single command.

In the previous example, no new variable was created - the output was simply printed to the screen, just like it would be if you typed the name of the variable. If you wanted to save it to a new variable, you would use the <- assignment operator, like this:

```
my_data_frame_slice <- my_data_frame %>%
  slice(1:2)

my_data_frame_slice
```

```
## # A tibble: 2 x 4
##   n         x     y     z
##   <chr> <dbl> <dbl> <dbl>
## 1 russ      1     4     7
## 2 lucy      2     5     8
```

2.1.2.4 Select a set of rows based on specific value(s)

```
my_data_frame %>%
  filter(n == "russ")
```

```
## # A tibble: 1 x 4
##   n         x     y     z
##   <chr> <dbl> <dbl> <dbl>
## 1 russ      1     4     7
```

`filter()` is a function that retains only those rows that meet your stated criteria. We can also filter for multiple criteria at once — in this example, the | symbol indicates "or":

```
my_data_frame %>%
  filter(n == "russ" | n == "lucy")
```

```
## # A tibble: 2 x 4
##   n         x     y     z
##   <chr> <dbl> <dbl> <dbl>
## 1 russ      1     4     7
## 2 lucy      2     5     8
```

2.1.2.5 Select a set of columns

```
my_data_frame %>%
  select(x:y)
```

```
## # A tibble: 4 x 2
##       x     y
##   <dbl> <dbl>
## 1     1     4
## 2     2     5
## 3     3     6
## 4     4     7
```

`select()` is a function that selects out only those columns you specify using their names

You can also specify a vector of columns to select.

```
my_data_frame %>%
  select(c(x,z))
```

```
## # A tibble: 4 x 2
##       x     z
##   <dbl> <dbl>
## 1     1     7
## 2     2     8
## 3     3     9
## 4     4    10
```

2.1.3 Adding a row or column

add a named row

```
tiffany_data_frame <-
  tibble(
    n = "tiffany",
    x = 13,
    y = 14,
    z = 15
  )
```

```
my_data_frame %>%
  bind_rows(tiffany_data_frame)
```

```
## # A tibble: 5 x 4
##   n           x     y     z
##   <chr>   <dbl> <dbl> <dbl>
## 1 russ        1     4     7
## 2 lucy        2     5     8
## 3 jaclyn      3     6     9
## 4 tyler       4     7    10
## 5 tiffany    13    14    15
```

`bind_rows()` is a function that combines the rows from another dataframe to the current dataframe

2.2 Creating or modifying variables using `mutate()`

Often we will want to either create a new variable based on an existing variable, or modify the value of an existing variable. Within the tidyverse, we do this using a function called `mutate()`. Let's start with a toy example by creating a data frame containing a single variable.

```
toy_df <- data.frame(x = c(1, 2, 3, 4))
glimpse(toy_df)
```

```
## Rows: 4
## Columns: 1
## $ x <dbl> 1, 2, 3, 4
```

Let's say that we wanted to create a new variable called `y` that would contain the value of x multiplied by 10. We could do this using `mutate()` and then assign the result back to the same data frame:

```
toy_df <- toy_df %>%
  # create a new variable called y that contains x*10
  mutate(y = x * 10)
glimpse(toy_df)
```

```
## Rows: 4
## Columns: 2
## $ x <dbl> 1, 2, 3, 4
## $ y <dbl> 10, 20, 30, 40
```

We could also overwrite a variable with a new value:

```
toy_df2 <- toy_df %>%
  # create a new variable called y that contains x*10
  mutate(y = y + 1)
glimpse(toy_df2)
```

```
## Rows: 4
## Columns: 2
## $ x <dbl> 1, 2, 3, 4
## $ y <dbl> 11, 21, 31, 41
```

We will use `mutate()` often so it's an important function to understand.

Here we can use it with our example data frame to create a new variable that is the sum of several other variables.

```
my_data_frame <-
  my_data_frame %>%
  mutate(total = x + y + z)

kable(my_data_frame)
```

n	x	y	z	total
russ	1	4	7	12
lucy	2	5	8	15
jaclyn	3	6	9	18
tyler	4	7	10	21

mutate() is a function that creates a new variable in a data frame using the existing variables. In this case, it creates a variable called total that is the sum of the existing variables x, y, and z.

2.2.1 Remove a column using the select() function

Adding a minus sign to the name of a variable within the `select()` command will remove that variable, leaving all of the others.

```
my_data_frame <-
  my_data_frame %>%
  dplyr::select(-total)

kable(my_data_frame)
```

n	x	y	z
russ	1	4	7
lucy	2	5	8
jaclyn	3	6	9
tyler	4	7	10

2.3 Tidyverse in action

To see the tidyverse in action, let's clean up the NHANES dataset. Each individual in the NHANES dataset has a unique identifier stored in the variable `ID`. First let's look at the number of rows in the dataset:

```
nrow(NHANES)
```

```
## [1] 10000
```

Now let's see how many unique IDs there are. The `unique()` function returns a vector containing all of the unique values for a particular variable, and the `length()` function returns the length of the resulting vector.

```
length(unique(NHANES$ID))
```

```
## [1] 6779
```

This shows us that while there are 10,000 observations in the data frame, there are only 6779 unique IDs. This means that if we were to use the entire dataset, we would be reusing data from some individuals, which could give us incorrect results. For this reason, we wold like to discard any observations that are duplicated.

Let's create a new variable called `NHANES_unique` that will contain only the

distinct observations, with no individuals appearing more than once. The `dplyr` library provides a function called `distinct()` that will do this for us. You may notice that we didn't explicitly load the `dplyr` library above; however, if you look at the messages that appeared when we loaded the `tidyverse` library, you will see that it loaded `dplyr` for us. To create the new data frame with unique observations, we will pipe the NHANES data frame into the `distinct()` function and then save the output to our new variable.

```
NHANES_unique <-
  NHANES %>%
  distinct(ID, .keep_all = TRUE)
```

If we number of rows in the new data frame, it should be the same as the number of unique IDs (6779):

```
nrow(NHANES_unique)
```

```
## [1] 6779
```

In the next example you will see the power of pipes come to life, when we start tying together multiple functions into a single operation (or "pipeline").

2.4 Looking at individual variables using pull() and head()

The NHANES data frame contains a large number of variables, but usually we are only interested in a particular variable. We can extract a particular variable from a data frame using the `pull()` function. Let's say that we want to extract the variable `PhysActive`. We could do this by piping the data frame into the pull command, which will result in a list of many thousands of values. Instead of printing out this entire list, we will pipe the result into the `head()` function, which just shows us the first few values contained in a variable. In this case we are not assigning the value back to a variable, so it will simply be printed to the screen.

```
NHANES %>%
  # extract the PhysActive variable
  pull(PhysActive) %>%
  # extract the first 10 values
```

```
  head(10) %>%
  kable()
```

x
No
No
No
NA
No
NA
NA
Yes
Yes
Yes

There are two important things to notice here. The first is that there are three different values apparent in the answers: "Yes", "No", and , which means that the value is missing for this person (perhaps they didn't want to answer that question on the survey). When we are working with data we generally need to remove missing values, as we will see below.

The second thing to notice is that R prints out a list of "Levels" of the variable. This is because this variable is defined as a particular kind of variable in R known as a *factor*. You can think of a factor variable as a categorial variable with a specific set of levels. The missing data are not treated as a level, so it can be useful to make the missing values explicit, which can be done using a function called `fct_explicit_na()` in the `forcats` package. Let's add a line to do that:

```
NHANES %>%
  mutate(PhysActive = fct_explicit_na(PhysActive)) %>%
  # extract the PhysActive variable
  pull(PhysActive) %>%
  # extract the first 10 values
  head(10) %>%
  kable()
```

x
No
No
No
(Missing)
No
(Missing)
(Missing)
Yes
Yes
Yes

This new line overwrote the old value of `PhysActive` with a version that has been processed by the `fct_explicit_na()` function to convert values to explicitly missing values. Now you can see that Missing values are treated as an explicit level, which will be useful later.

Now we are ready to start summarizing data!

2.5 Computing a frequency distribution (Section 2.5)

We would like to compute a frequency distribution showing how many people report being either active or inactive. The following statement is fairly complex so we will step through it one bit at a time.

```
PhysActive_table <- NHANES_unique %>%
  # convert the implicit missing values to explicit
  mutate(PhysActive = fct_explicit_na(PhysActive)) %>%
  # select the variable of interest
  dplyr::select(PhysActive) %>%
  # group by values of the variable
  group_by(PhysActive) %>%
  # count the values
  summarize(AbsoluteFrequency = n())

# kable() prints out the table in a prettier way.
kable(PhysActive_table)
```

PhysActive	AbsoluteFrequency
No	2473
Yes	2972
(Missing)	1334

This data frame still contains all of the original variables. Since we are only interested in the `PhysActive` variable, let's extract that one and get rid of the rest. We can do this using the `select()` command from the `dplyr` package. Because there is also another select command available in R, we need to explicitly refer to the one from the `dplyr` package, which we do by including the package name followed by two colons: `dplyr::select()`.

```
NHANES_unique %>%
  # convert the implicit missing values to explicit
  mutate(PhysActive = fct_explicit_na(PhysActive)) %>%
  # select the variable of interest
  dplyr::select(PhysActive) %>%
  head(10) %>%
  kable()
```

PhysActive
No
(Missing)
No
(Missing)
(Missing)
Yes
Yes
Yes
Yes
(Missing)

The next function, `group_by()` tells R that we are going to want to analyze the data separate according to the different levels of the `PhysActive` variable:

```
NHANES_unique %>%
  # convert the implicit missing values to explicit
  mutate(PhysActive = fct_explicit_na(PhysActive)) %>%
  # select the variable of interest
  dplyr::select(PhysActive) %>%
```

```
  group_by(PhysActive) %>%
  head(10) %>%
  kable()
```

PhysActive
No
(Missing)
No
(Missing)
(Missing)
Yes
Yes
Yes
Yes
(Missing)

The final command tells R to create a new data frame by summarizing the data that we are passing in (which in this case is the PhysActive variable, grouped by its different levels). We tell the `summarize()` function to create a new variable (called `AbsoluteFrequency`) will contain a count of the number of observations for each group, which is generated by the `n()` function.

```
NHANES_unique %>%
  # convert the implicit missing values to explicit
  mutate(PhysActive = fct_explicit_na(PhysActive)) %>%
  # select the variable of interest
  dplyr::select(PhysActive) %>%
  group_by(PhysActive) %>%
  summarize(AbsoluteFrequency = n())  %>%
  kable()
```

PhysActive	AbsoluteFrequency
No	2473
Yes	2972
(Missing)	1334

Now let's say we want to add another column with percentage of observations in each group. We compute the percentage by dividing the absolute frequency for each group by the total number. We can use the table that we already

generated, and add a new variable, again using `mutate()`:

```
PhysActive_table <- PhysActive_table %>%
  mutate(
    Percentage = AbsoluteFrequency /
      sum(AbsoluteFrequency) * 100
  )

kable(PhysActive_table, digits=2)
```

PhysActive	AbsoluteFrequency	Percentage
No	2473	36.48
Yes	2972	43.84
(Missing)	1334	19.68

2.6 Computing a cumulative distribution (Section 2.6)

Let's compute a cumulative distribution for the `SleepHrsNight` variable in NHANES. This looks very similar to what we saw in the previous section.

```
# create summary table for relative frequency of different
# values of SleepHrsNight

SleepHrsNight_cumulative <-
  NHANES_unique %>%
  # drop NA values for SleepHrsNight variable
  drop_na(SleepHrsNight) %>%
  # remove other variables
  dplyr::select(SleepHrsNight) %>%
  # group by values
  group_by(SleepHrsNight) %>%
  # create summary table
  summarize(AbsoluteFrequency = n()) %>%
  # create relative and cumulative frequencies
  mutate(
    RelativeFrequency = AbsoluteFrequency /
      sum(AbsoluteFrequency),
```

```
    CumulativeDensity = cumsum(RelativeFrequency)
  )

kable(SleepHrsNight_cumulative)
```

SleepHrsNight	AbsoluteFrequency	RelativeFrequency	CumulativeDensity
2	9	0.0017875	0.0017875
3	49	0.0097319	0.0115194
4	200	0.0397219	0.0512413
5	406	0.0806356	0.1318769
6	1172	0.2327706	0.3646475
7	1394	0.2768620	0.6415094
8	1405	0.2790467	0.9205561
9	271	0.0538232	0.9743793
10	97	0.0192651	0.9936445
11	15	0.0029791	0.9966236
12	17	0.0033764	1.0000000

2.7 Data cleaning and tidying with R

Now that you know a bit about the tidyverse, let's look at the various tools that it provides for working with data. We will use as an example an analysis of whether attitudes about statistics are different between the different student year groups in the class.

2.7.1 Statistics attitude data from course survey

These data were collected using the Attitudes Towards Statistics (ATS) scale (from https://www.stat.auckland.ac.nz/~iase/cblumberg/wise2.pdf).

The 29-item ATS has two subscales. The Attitudes Toward Field subscale consists of the following 20 items, with reverse-keyed items indicated by an "(R)": 1, 3, 5, 6(R), 9, 10(R), 11, 13, 14(R), 16(R), 17, 19, 20(R), 21, 22, 23, 24, 26, 28(R), 29

The Attitudes Toward Course subscale consists of the following 9 items: 2(R), 4(R), 7(R), 8, 12(R), 15(R), 18(R), 25(R), 27(R)

For our purposes, we will just combine all 29 items together, rather than separating them into these subscales.

Note: I have removed the data from the graduate students and 5+ year students, since those would be too easily identifiable given how few there are.

Let's first create a variable containing the file path to the data.

```
attitudeData_file <- 'data/statsAttitude.txt'
```

Next, let's load the data from the file using the tidyverse function `read_tsv()`. There are several functions available for reading in different file formats as part of the the `readr` tidyverse package.

Right now the variable names are unwieldy, since they include the entire name of the item; this is how Google Forms stores the data. Let's change the variable names to be somewhat more readable. We will change the names to "ats" where is replaced with the question number and ats indicates Attitudes Toward Statistics scale. We can create these names using the `rename()` and `paste0()` functions. `rename()` is pretty self-explanatory: a new name is assigned to an old name or a column position. The `paste0()` function takes a string along with a set of numbers, and creates a vector that combines the string with the number.

```
nQuestions <- 29 # other than the first two columns,
# the rest of the columns are for the 29 questions in the statistics
# attitude survey; we'll use this below to rename these columns
# based on their question number

# use rename to change the first two column names
# rename can refer to columns either by their number or their name
attitudeData <-
  attitudeData %>%
  rename(     # rename using columm numbers
    # The first column is the year
    Year = 1,
    # The second column indicates
    # whether the person took stats before
    StatsBefore = 2
  ) %>%
```

```
  rename_at(
    # rename all the columns except Year and StatsBefore
    vars(-Year, -StatsBefore),
    #rename by pasting the word "stat" and the number
    list(name = ~paste0('ats', 1:nQuestions))
  )

# print out the column names
names(attitudeData)
```

```
##  [1] "Year"        "StatsBefore" "ats1"        "ats2"        "ats3"
##  [6] "ats4"        "ats5"        "ats6"        "ats7"        "ats8"
## [11] "ats9"        "ats10"       "ats11"       "ats12"       "ats13"
## [16] "ats14"       "ats15"       "ats16"       "ats17"       "ats18"
## [21] "ats19"       "ats20"       "ats21"       "ats22"       "ats23"
## [26] "ats24"       "ats25"       "ats26"       "ats27"       "ats28"
## [31] "ats29"
```

```
#check out the data again
glimpse(attitudeData)
```

```
## Rows: 148
## Columns: 31
## $ Year        <dbl> 3, 4, 2, 1, 2, 3, 4, 2, 2, 2, 4, 2, 3,~
## $ StatsBefore <chr> "Yes", "No", "No", "Yes", "No", "No", ~
## $ ats1        <dbl> 6, 4, 6, 3, 7, 4, 6, 5, 7, 5, 5, 4, 2,~
## $ ats2        <dbl> 1, 5, 5, 2, 7, 5, 5, 4, 2, 2, 3, 3, 7,~
## $ ats3        <dbl> 7, 6, 5, 7, 2, 4, 7, 7, 7, 5, 6, 5, 7,~
## $ ats4        <dbl> 2, 5, 5, 2, 7, 3, 3, 4, 5, 3, 3, 2, 3,~
## $ ats5        <dbl> 7, 5, 6, 7, 5, 4, 6, 6, 7, 5, 3, 5, 4,~
## $ ats6        <dbl> 1, 4, 5, 2, 2, 4, 2, 3, 1, 2, 2, 3, 1,~
## $ ats7        <dbl> 1, 4, 3, 2, 4, 4, 2, 2, 3, 2, 4, 2, 4,~
## $ ats8        <dbl> 2, 1, 4, 3, 1, 4, 4, 4, 7, 3, 2, 4, 1,~
## $ ats9        <dbl> 5, 4, 5, 5, 7, 4, 5, 5, 7, 6, 3, 5, 5,~
## $ ats10       <dbl> 2, 3, 2, 2, 1, 4, 2, 2, 1, 3, 3, 1, 1,~
## $ ats11       <dbl> 6, 4, 6, 2, 7, 4, 6, 5, 7, 3, 3, 4, 2,~
## $ ats12       <dbl> 2, 4, 1, 2, 5, 7, 2, 1, 2, 4, 4, 2, 4,~
## $ ats13       <dbl> 6, 4, 5, 5, 7, 3, 6, 6, 7, 5, 2, 5, 1,~
```

```
## $ ats14       <dbl> 2, 4, 3, 3, 3, 4, 2, 1, 1, 3, 3, 2, 1,~
## $ ats15       <dbl> 2, 4, 3, 3, 5, 6, 3, 4, 2, 3, 2, 4, 3,~
## $ ats16       <dbl> 1, 3, 2, 5, 1, 5, 2, 1, 2, 3, 2, 2, 1,~
## $ ats17       <dbl> 7, 7, 5, 7, 7, 4, 6, 6, 7, 6, 6, 7, 4,~
## $ ats18       <dbl> 2, 5, 4, 5, 7, 4, 2, 4, 2, 5, 2, 4, 6,~
## $ ats19       <dbl> 3, 3, 4, 3, 2, 3, 6, 5, 7, 3, 3, 5, 2,~
## $ ats20       <dbl> 1, 4, 1, 2, 1, 4, 2, 2, 1, 2, 3, 2, 3,~
## $ ats21       <dbl> 6, 3, 5, 5, 7, 5, 6, 5, 7, 3, 4, 6, 6,~
## $ ats22       <dbl> 7, 4, 5, 6, 7, 5, 6, 5, 7, 5, 5, 5, 5,~
## $ ats23       <dbl> 6, 4, 6, 6, 7, 5, 6, 7, 7, 5, 3, 5, 3,~
## $ ats24       <dbl> 7, 4, 4, 6, 7, 5, 6, 5, 7, 5, 5, 5, 3,~
## $ ats25       <dbl> 3, 5, 3, 3, 5, 4, 3, 4, 2, 3, 3, 2, 5,~
## $ ats26       <dbl> 7, 4, 5, 6, 2, 4, 6, 5, 7, 3, 4, 4, 2,~
## $ ats27       <dbl> 2, 4, 2, 2, 4, 4, 2, 1, 2, 3, 3, 2, 1,~
## $ ats28       <dbl> 2, 4, 3, 5, 2, 3, 3, 1, 1, 4, 3, 2, 2,~
## $ ats29       <dbl> 4, 4, 3, 6, 2, 1, 5, 3, 3, 3, 2, 3, 2,~
```

The next thing we need to do is to create an ID for each individual. To do this, we will use the `rownames_to_column()` function from the tidyverse. This creates a new variable (which we name "ID") that contains the row names from the data frame; thsee are simply the numbers 1 to N.

```
# let's add a participant ID so that we will be able to
# identify them later
attitudeData <-
  attitudeData %>%
  rownames_to_column(var = 'ID')

head(attitudeData)
```

```
## # A tibble: 6 x 32
##   ID     Year StatsBefore  ats1  ats2  ats3  ats4  ats5
##   <chr> <dbl> <chr>       <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 1         3 Yes             6     1     7     2     7
## 2 2         4 No              4     5     6     5     5
## 3 3         2 No              6     5     5     5     6
## 4 4         1 Yes             3     2     7     2     7
## 5 5         2 No              7     7     2     7     5
## 6 6         3 No              4     5     4     3     4
```

```
## # i 24 more variables: ats6 <dbl>, ats7 <dbl>, ats8 <dbl>,
## #   ats9 <dbl>, ats10 <dbl>, ats11 <dbl>, ats12 <dbl>,
## #   ats13 <dbl>, ats14 <dbl>, ats15 <dbl>, ats16 <dbl>,
## #   ats17 <dbl>, ats18 <dbl>, ats19 <dbl>, ats20 <dbl>,
## #   ats21 <dbl>, ats22 <dbl>, ats23 <dbl>, ats24 <dbl>,
## #   ats25 <dbl>, ats26 <dbl>, ats27 <dbl>, ats28 <dbl>,
## #   ats29 <dbl>
```

If you look closely at the data, you can see that some of the participants have some missing responses. We can count them up for each individual and create a new variable to store this to a new variable called `numNA` using `mutate()`.

We can also create a table showing how many participants have a particular number of NA values. Here we use two additional commands that you haven't seen yet. The `group_by()` function tells other functions to do their analyses while breaking the data into groups based on one of the variables. Here we are going to want to summarize the number of people with each possible number of NAs, so we will group responses by the numNA variable that we are creating in the first command here.

The summarize() function creates a summary of the data, with the new variables based on the data being fed in. In this case, we just want to count up the number of subjects in each group, which we can do using the special n() function from dpylr.

```
# compute the number of NAs for each participant
attitudeData <-
  attitudeData %>%
  mutate(
    # we use the . symbol to tell the is.na function
    # to look at the entire data frame
    numNA = rowSums(is.na(.))
  )

# present a table with counts of the number of missing responses
attitudeData %>%
  count(numNA)
```

```
## # A tibble: 3 x 2
##   numNA     n
```

```
##   <dbl> <int>
## 1     0   141
## 2     1     6
## 3     2     1
```

We can see from the table that there are only a few participants with missing data; six people are missing one answer, and one is missing two answers. Let's find those individuals, using the filter() command from dplyr. filter() returns the subset of rows from a data frame that match a particular test - in this case, whether numNA is > 0.

```
attitudeData %>%
  filter(numNA > 0)
```

```
## # A tibble: 7 x 33
##   ID     Year StatsBefore  ats1  ats2  ats3  ats4  ats5
##   <chr> <dbl> <chr>       <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 42        2 No             NA     2     7     5     6
## 2 55        1 No              5     3     7     4     5
## 3 90        1 No              7     2     7     5     7
## 4 113       2 No              5     7     7     5     6
## 5 117       2 Yes             6     6     7     4     6
## 6 137       3 No              7     5     6     5     6
## 7 139       1 No              7     5     7     5     6
## # i 25 more variables: ats6 <dbl>, ats7 <dbl>, ats8 <dbl>,
## #   ats9 <dbl>, ats10 <dbl>, ats11 <dbl>, ats12 <dbl>,
## #   ats13 <dbl>, ats14 <dbl>, ats15 <dbl>, ats16 <dbl>,
## #   ats17 <dbl>, ats18 <dbl>, ats19 <dbl>, ats20 <dbl>,
## #   ats21 <dbl>, ats22 <dbl>, ats23 <dbl>, ats24 <dbl>,
## #   ats25 <dbl>, ats26 <dbl>, ats27 <dbl>, ats28 <dbl>,
## #   ats29 <dbl>, numNA <dbl>
```

There are fancy techniques for trying to guess the value of missing data (known as "imputation") but since the number of participants with missing values is small, let's just drop those participants from the list. We can do this using the `drop_na()` function from the `tidyr` package, another tidyverse package that provides tools for cleaning data. We will also remove the numNA variable, since we won't need it anymore after removing the subjects with missing answeres. We do this using the `select()` function from the `dplyr`

tidyverse package, which selects or removes columns from a data frame. By putting a minus sign in front of numNA, we are telling it to remove that column.

`select()` and `filter()` are similar - `select()` works on columns (i.e. variables) and `filter()` works on rows (i.e. observations).

```
# this is equivalent to drop_na(attitudeData)
attitudeDataNoNA <-
  attitudeData %>%
  drop_na() %>%
  select(-numNA)
```

Try the following on your own: Using the attitudeData data frame, drop the NA values, create a new variable called mystery that contains a value of 1 for anyone who answered 7 to question ats4 ("Statistics seems very mysterious to me"). Create a summary that includes the number of people reporting 7 on this question, and the proportion of people who reported 7.

2.7.1.1 Tidy data

These data are in a format that meets the principles of "tidy data", which state the following:

- Each variable must have its own column.
- Each observation must have its own row.
- Each value must have its own cell.

In our case, each column represents a variable: `ID` identifies which student responded, `Year` contains their year at Stanford, `StatsBefore` contains whether or not they have taken statistics before, and ats1 through ats29 contain their responses to each item on the ATS scale. Each observation (row) is a response from one individual student. Each value has its own cell (e.g., the values for `Year` and `StatsBefoe` are stored in separate cells in separate columns).

For an example of data that are NOT tidy, take a look at these data Belief in Hell - click on the "Table" tab to see the data.

- What are the variables
- Why aren't these data tidy?

2.7.1.2 Recoding data

We now have tidy data; however, some of the ATS items require recoding. Specifically, some of the items need to be "reverse coded"; these items include: ats2, ats4, ats6, ats7, ats10, ats12, ats14, ats15, ats16, ats18, ats20, ats25, ats27 and ats28. The raw responses for each item are on the 1-7 scale; therefore, for the reverse coded items, we need to reverse them by subtracting the raw score from 8 (such that 7 becomes 1 and 1 becomes 7). To recode these items, we will use the tidyverse `mutate()` function. It's a good idea when recoding to preserve the raw original variables and create new recoded variables with different names.

There are two ways we can use `mutate()` function to recode these variables. The first way is easier to understand as a new code, but less efficient and more prone to error. Specifically, we repeat the same code for every variable we want to reverse code as follows:

```
attitudeDataNoNA %>%
  mutate(
    ats2_re = 8 - ats2,
    ats4_re = 8 - ats4,
    ats6_re = 8 - ats6,
    ats7_re = 8 - ats7,
    ats10_re = 8 - ats10,
    ats12_re = 8 - ats12,
    ats14_re = 8 - ats14,
    ats15_re = 8 - ats15,
    ats16_re = 8 - ats16,
    ats18_re = 8 - ats18,
    ats20_re = 8 - ats20,
    ats25_re = 8 - ats25,
    ats27_re = 8 - ats27,
    ats28_re = 8 - ats28
  )
```

```
## # A tibble: 141 x 46
##    ID     Year StatsBefore  ats1  ats2  ats3  ats4  ats5
##    <chr> <dbl> <chr>       <dbl> <dbl> <dbl> <dbl> <dbl>
##  1 1         3 Yes             6     1     7     2     7
```

```
##  2 2         4 No              4     5     6     5     5
##  3 3         2 No              6     5     5     5     6
##  4 4         1 Yes             3     2     7     2     7
##  5 5         2 No              7     7     2     7     5
##  6 6         3 No              4     5     4     3     4
##  7 7         4 Yes             6     5     7     3     6
##  8 8         2 Yes             5     4     7     4     6
##  9 9         2 Yes             7     2     7     5     7
## 10 10        2 Yes             5     2     5     3     5
## # i 131 more rows
## # i 38 more variables: ats6 <dbl>, ats7 <dbl>, ats8 <dbl>,
## #   ats9 <dbl>, ats10 <dbl>, ats11 <dbl>, ats12 <dbl>,
## #   ats13 <dbl>, ats14 <dbl>, ats15 <dbl>, ats16 <dbl>,
## #   ats17 <dbl>, ats18 <dbl>, ats19 <dbl>, ats20 <dbl>,
## #   ats21 <dbl>, ats22 <dbl>, ats23 <dbl>, ats24 <dbl>,
## #   ats25 <dbl>, ats26 <dbl>, ats27 <dbl>, ats28 <dbl>, ...
```

The second way is more efficient and takes advatange of the use of "scoped verbs" (https://dplyr.tidyverse.org/reference/scoped.html), which allow you to apply the same code to several variables at once. Because you don't have to keep repeating the same code, you're less likely to make an error:

```
#create a vector of the names of the variables to recode
ats_recode <-
  c(
    "ats2",
    "ats4",
    "ats6",
    "ats7",
    "ats10",
    "ats12",
    "ats14",
    "ats15",
    "ats16",
    "ats18",
    "ats20",
    "ats25",
    "ats27",
    "ats28"
```

```
  )


attitudeDataNoNA <-
  attitudeDataNoNA %>%
  mutate_at(
    vars(ats_recode), # the variables you want to recode
    funs(re = 8 - .) # the function to apply to each variable
  )
```

Whenever we do an operation like this, it's good to check that it actually worked correctly. It's easy to make mistakes in coding, which is why it's important to check your work as well as you can.

We can quickly select a couple of the raw and recoded columns from our data and make sure things appear to have gone according to plan:

```
attitudeDataNoNA %>%
  select(
    ats2,
    ats2_re,
    ats4,
    ats4_re
  )
```

```
## # A tibble: 141 x 4
##     ats2 ats2_re  ats4 ats4_re
##    <dbl>   <dbl> <dbl>   <dbl>
##  1     1       7     2       6
##  2     5       3     5       3
##  3     5       3     5       3
##  4     2       6     2       6
##  5     7       1     7       1
##  6     5       3     3       5
##  7     5       3     3       5
##  8     4       4     4       4
##  9     2       6     5       3
## 10     2       6     3       5
## # i 131 more rows
```

Let's also make sure that there are no responses outside of the 1-7 scale that we expect, and make sure that no one specified a year outside of the 1-4 range.

```
attitudeDataNoNA %>%
  summarise_at(
    vars(ats1:ats28_re),
    funs(min, max)
  )
```

```
## # A tibble: 1 x 86
##   ats1_min ats2_min ats3_min ats4_min ats5_min ats6_min
##      <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>
## 1        1        1        2        1        2        1
## # i 80 more variables: ats7_min <dbl>, ats8_min <dbl>,
## #   ats9_min <dbl>, ats10_min <dbl>, ats11_min <dbl>,
## #   ats12_min <dbl>, ats13_min <dbl>, ats14_min <dbl>,
## #   ats15_min <dbl>, ats16_min <dbl>, ats17_min <dbl>,
## #   ats18_min <dbl>, ats19_min <dbl>, ats20_min <dbl>,
## #   ats21_min <dbl>, ats22_min <dbl>, ats23_min <dbl>,
## #   ats24_min <dbl>, ats25_min <dbl>, ats26_min <dbl>, ...
```

```
attitudeDataNoNA %>%
  summarise_at(
    vars(Year),
    funs(min, max)
  )
```

```
## # A tibble: 1 x 2
##     min   max
##   <dbl> <dbl>
## 1     1     4
```

2.7.1.3 Different data formats

Sometimes we need to reformat our data in order to analyze it or visualize it in a specific way. Two tidyverse functions, `gather()` and `spread()`, help us to do this.

For example, say we want to examine the distribution of the raw responses

to each of the ATS items (i.e., a histogram). In this case, we would need our x-axis to be a single column of the responses across all the ATS items. However, currently the responses for each item are stored in 29 different columns.

This means that we need to create a new version of this dataset. It will have four columns: - ID - Year - Question (for each of the ATS items) - ResponseRaw (for the raw response to each of the ATS items)

Thus, we want change the format of the dataset from being "wide" to being "long".

We change the format to "wide" using the `gather()` function.

`gather()` takes a number of variables and reformates them into two variables: one that contains the variable values, and another called the "key" that tells us which variable the value came from. In this case, we want it to reformat the data so that each response to an ATS question is in a separate row and the key column tells us which ATS question it corresponds to. It is much better to see this in practice than to explain in words!

```
attitudeData_long <-
  attitudeDataNoNA %>%
  #remove the raw variables that you recoded
  select(-ats_recode) %>%
  gather(
    # key refers to the new variable containing the question number
    key = question,
    # value refers to the new response variable
    value = response,
    #the only variables we DON'T want to gather
    -ID, -Year, -StatsBefore
  )

attitudeData_long %>%
  slice(1:20)
```

```
## # A tibble: 20 x 5
##    ID     Year StatsBefore question response
##    <chr> <dbl> <chr>       <chr>       <dbl>
```

```
##  1 1          3 Yes          ats1             6
##  2 2          4 No           ats1             4
##  3 3          2 No           ats1             6
##  4 4          1 Yes          ats1             3
##  5 5          2 No           ats1             7
##  6 6          3 No           ats1             4
##  7 7          4 Yes          ats1             6
##  8 8          2 Yes          ats1             5
##  9 9          2 Yes          ats1             7
## 10 10         2 Yes          ats1             5
## 11 11         4 No           ats1             5
## 12 12         2 No           ats1             4
## 13 13         3 Yes          ats1             2
## 14 14         1 Yes          ats1             6
## 15 15         2 No           ats1             7
## 16 16         4 No           ats1             7
## 17 17         2 No           ats1             7
## 18 18         2 No           ats1             6
## 19 19         1 No           ats1             6
## 20 20         1 No           ats1             3
```

```
glimpse(attitudeData_long)
```

```
## Rows: 4,089
## Columns: 5
## $ ID           <chr> "1", "2", "3", "4", "5", "6", "7", "8"~
## $ Year         <dbl> 3, 4, 2, 1, 2, 3, 4, 2, 2, 2, 4, 2, 3,~
## $ StatsBefore <chr> "Yes", "No", "No", "Yes", "No", "No", ~
## $ question     <chr> "ats1", "ats1", "ats1", "ats1", "ats1"~
## $ response     <dbl> 6, 4, 6, 3, 7, 4, 6, 5, 7, 5, 5, 4, 2,~
```

Say we now wanted to undo the `gather()` and return our dataset to wide format. For this, we would use the function `spread()`.

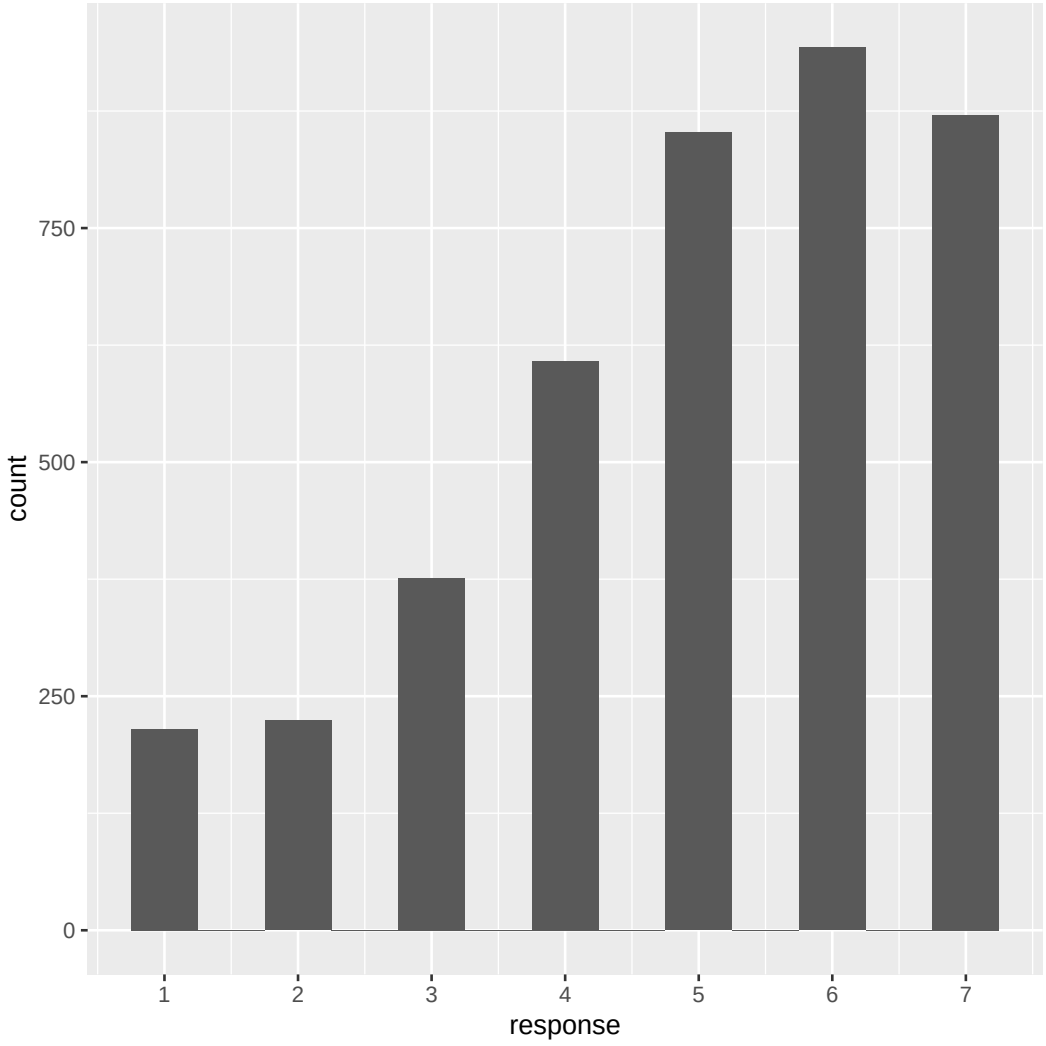
```
attitudeData_wide <-
  attitudeData_long %>%
  spread(
    #key refers to the variable indicating which question
    # each response belongs to
```

```
    key = question,
    value = response
  )

attitudeData_wide %>%
  slice(1:20)
```

```
## # A tibble: 20 x 32
##    ID     Year StatsBefore  ats1 ats10_re ats11 ats12_re
##    <chr> <dbl> <chr>       <dbl>    <dbl> <dbl>    <dbl>
##  1 1         3 Yes             6        6     6        6
##  2 10        2 Yes             5        5     3        4
##  3 100       4 Yes             5        6     4        2
##  4 101       2 No              4        7     2        4
##  5 102       3 Yes             5        6     5        6
##  6 103       2 No              6        7     5        7
##  7 104       2 Yes             6        5     5        3
##  8 105       3 No              6        6     5        6
##  9 106       1 No              4        4     4        4
## 10 107       2 No              1        2     1        1
## 11 108       2 No              7        7     7        7
## 12 109       2 No              4        4     4        6
## 13 11        4 No              5        5     3        4
## 14 110       3 No              5        7     4        4
## 15 111       2 No              6        6     6        3
## 16 112       3 No              6        7     5        7
## 17 114       2 No              5        4     4        3
## 18 115       3 No              5        7     5        1
## 19 116       3 No              5        6     5        5
## 20 118       2 No              6        6     6        1
## # i 25 more variables: ats13 <dbl>, ats14_re <dbl>,
## #   ats15_re <dbl>, ats16_re <dbl>, ats17 <dbl>,
## #   ats18_re <dbl>, ats19 <dbl>, ats2_re <dbl>,
## #   ats20_re <dbl>, ats21 <dbl>, ats22 <dbl>, ats23 <dbl>,
## #   ats24 <dbl>, ats25_re <dbl>, ats26 <dbl>,
## #   ats27_re <dbl>, ats28_re <dbl>, ats29 <dbl>,
## #   ats3 <dbl>, ats4_re <dbl>, ats5 <dbl>, ...
```

Now that we have created a "long" version of our data, they are in the right format to create the plot. We will use the tidyverse function `ggplot()` to create our histogram with `geom_histogram`.

```
attitudeData_long %>%
  ggplot(aes(x = response)) +
  geom_histogram(binwidth = 0.5) +
  scale_x_continuous(breaks = seq.int(1, 7, 1))
```



It looks like responses were fairly positively overall.

We can also aggregate each participant's responses to each question during each year of their study at Stanford to examine the distribution of mean ATS responses across people by year.

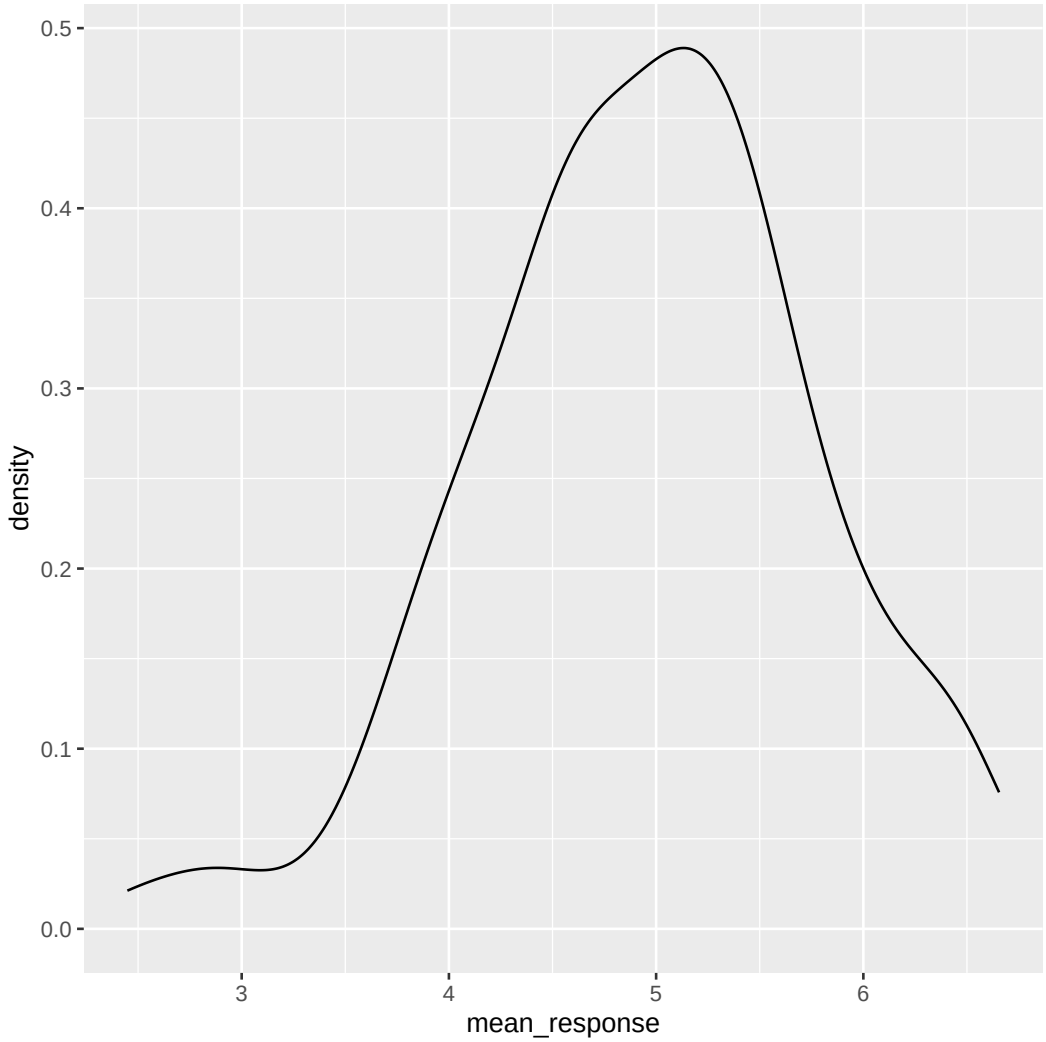
We will use the `group_by()` and `summarize()` functions to aggregate the responses.

```
attitudeData_agg <-
  attitudeData_long %>%
  group_by(ID, Year) %>%
  summarize(
    mean_response = mean(response)
  )
attitudeData_agg
```

```
## # A tibble: 141 x 3
## # Groups:   ID [141]
##    ID     Year mean_response
##    <chr> <dbl>         <dbl>
##  1 1         3          6
##  2 10        2          4.66
##  3 100       4          5.03
##  4 101       2          5.10
##  5 102       3          4.66
##  6 103       2          5.55
##  7 104       2          4.31
##  8 105       3          5.10
##  9 106       1          4.21
## 10 107       2          2.45
## # i 131 more rows
```

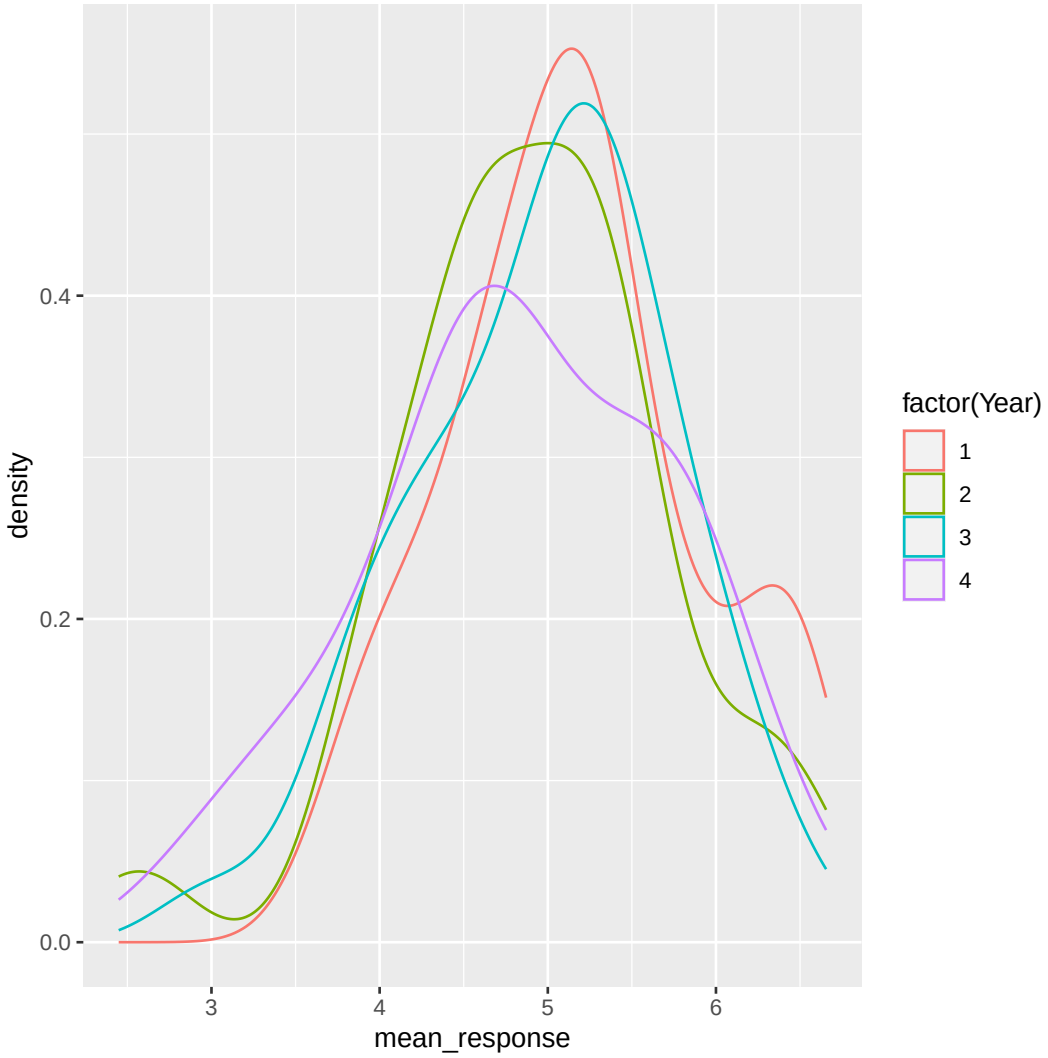
First let's use the geom_density argument in `ggplot()` to look at mean responses across people, ignoring year of response. The density argrument is like a histogram but smooths things over a bit.

```
attitudeData_agg %>%
  ggplot(aes(mean_response)) +
  geom_density()
```



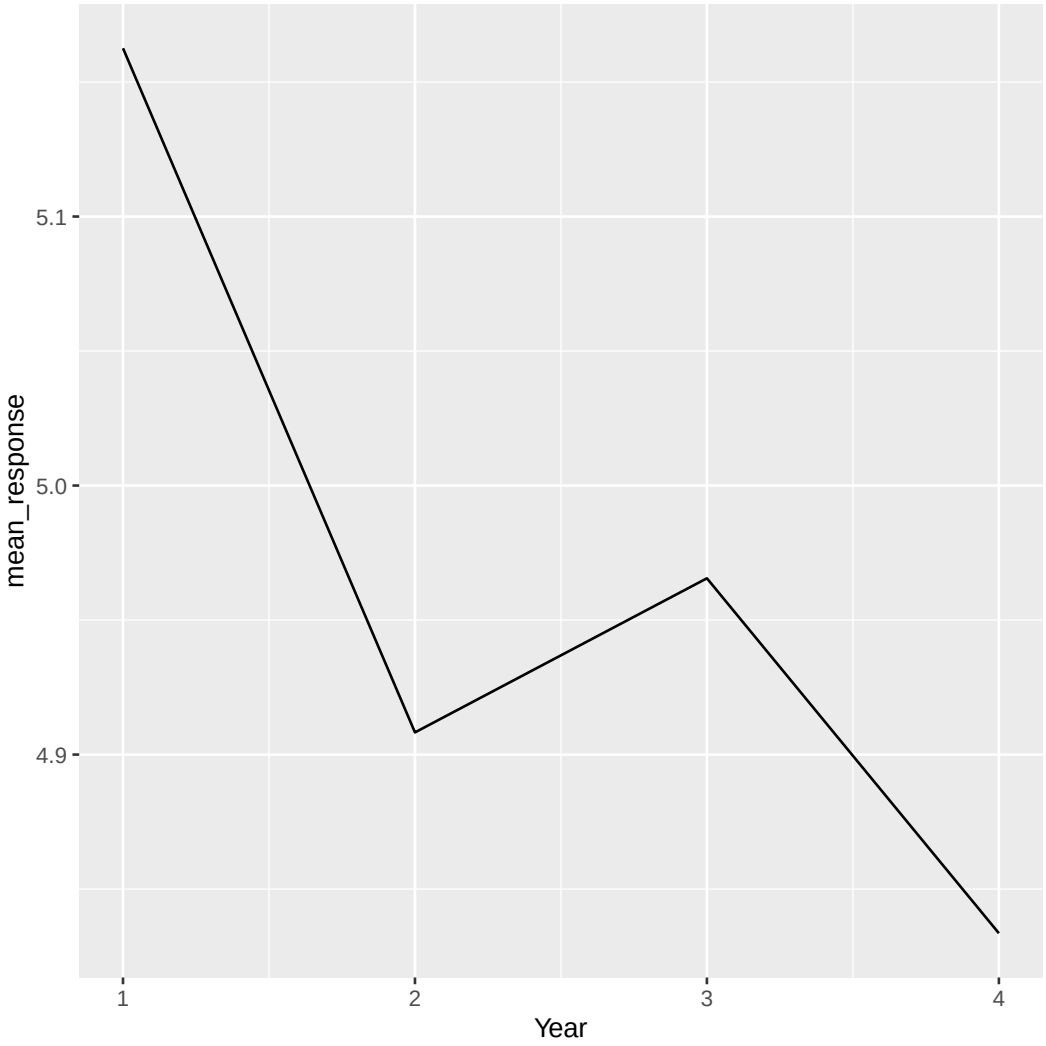
Now we can also look at the distribution for each year.

```
attitudeData_agg %>%
  ggplot(aes(mean_response, color = factor(Year))) +
  geom_density()
```



Or look at trends in responses across years.

```
attitudeData_agg %>%
  group_by(Year) %>%
  summarise(
    mean_response = mean(mean_response)
  ) %>%
  ggplot(aes(Year, mean_response)) +
  geom_line()
```



This looks like a precipitous drop - but how might that be misleading?

Chapter 3

Data visualization using R (with Anna Khazenzon)

There are many different tools for plotting data in R, but we will focus on the `ggplot()` function provided by a package called `ggplot2`. ggplot is very powerful, but using it requires getting one's head around how it works.

3.1 The grammar of graphics

or, the "gg" in ggplot

Each language has a grammar consisting of types of words and the rules with which to string them together into sentences. If a sentence is grammatically correct, we're able to parse it, even though that doesn't ensure that it's interesting, beautiful, or even meaningful.

Similarly, plots can be divided up into their core components, which come together via a set of rules.

Some of the major components are :

- data
- aesthetics
- geometries
- themes

The data are the actual variables we're plotting, which we pass to ggplot through the data argument. As you've learned, ggplot takes a **dataframe** in which each column is a variable.

Now we need to tell ggplot *how* to plot those variables, by mapping each variable to an *axis* of the plot. You've seen that when we plot histograms, our variable goes on the x axis. Hence, we set `x=<variable>` in a call to `aes()` within `ggplot()`. This sets **aesthetics**, which are mappings of data to certain scales, like axes or things like color or shape. The plot still had two axes – x and y – but we didn't need to *specify* what went on the y axis because ggplot knew by *default* that it should make a count variable.

How was ggplot able to figure that out? Because of **geometries**, which are *shapes* we use to represent our data. You've seen `geom_histogram`, which basically gives our graph a bar plot shape, except that it also sets the default y axis variable to be `count`. Other shapes include points and lines, among many others.

We'll go over other aspects of the grammar of graphics (such as facets, statistics, and coordinates) as they come up. Let's start visualizing some data by first choosing a **theme**, which describes all of the non-data ink in our plot, like grid lines and text.

3.2 Getting started

Load ggplot and choose a theme you like (see here for examples).

```
library(tidyverse)

theme_set(theme_bw()) # I like this fairly minimal one
```

3.3 Let's think through a visualization

Principles we want to keep in mind:

- Show the data without distortion
- Use color, shape, and location to encourage comparisons
- Minimize visual clutter (maximize your information to ink ratio)

The two questions you want to ask yourself before getting started are:

- What type of variable(s) am I plotting?
- What comparison do I want to make salient for the viewer (possibly myself)?

Figuring out *how* to highlight a comparison and include relevant variables usually benefits from sketching the plot out first.

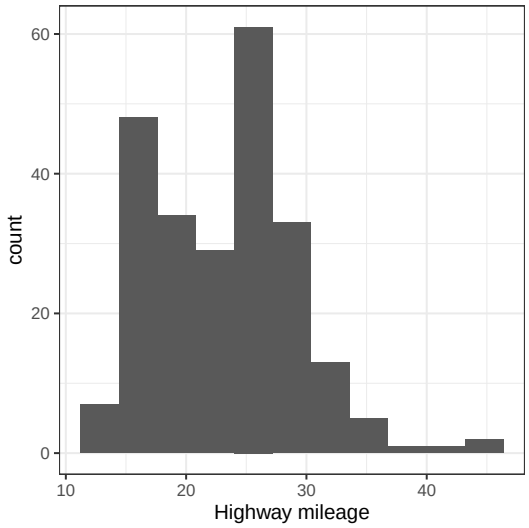
3.4 Plotting the distribution of a single variable

How do you choose which **geometry** to use? ggplot allows you to choose from a number of geometries. This choice will determine what sort of plot you create. We will use the built-in mpg dataset, which contains fuel efficiency data for a number of different cars.

3.4.1 Histogram

The histogram shows the overall distribution of the data. Here we use the nclass.FD function to compute the optimal bin size.

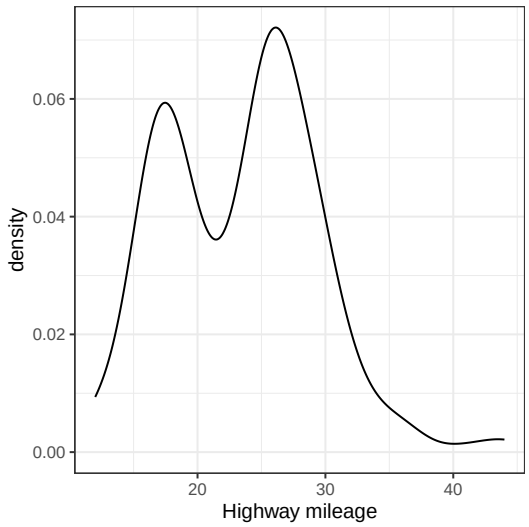
```
ggplot(mpg, aes(hwy)) +
  geom_histogram(bins = nclass.FD(mpg$hwy)) +
  xlab('Highway mileage')
```



Instead of creating discrete bins, we can look at relative density continuously.

3.4.2 Density plot

```
ggplot(mpg, aes(hwy)) +
  geom_density() +
  xlab('Highway mileage')
```



A note on defaults: The default statistic (or "stat") underlying `geom_density`

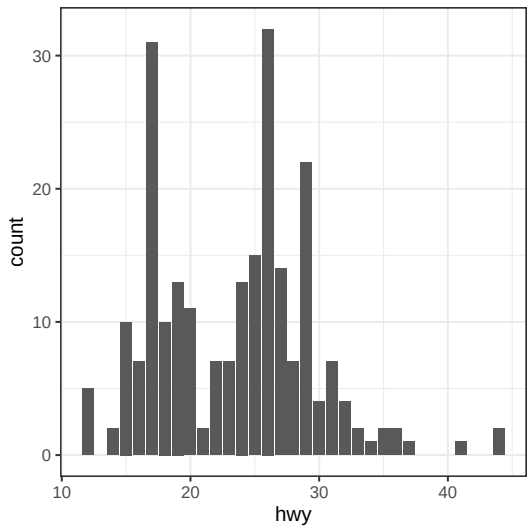
is called "density" – not surprising. The default stat for `geom_histogram` is "count". What do you think would happen if you overrode the default and set `stat="count"`?

```
ggplot(mpg, aes(hwy)) +
  geom_density(stat = "count")
```

What we discover is that the *geometric* difference between `geom_histogram` and `geom_density` can actually be generalized. `geom_histogram` is a shortcut for working with `geom_bar`, and `geom_density` is a shortcut for working with `geom_line`.

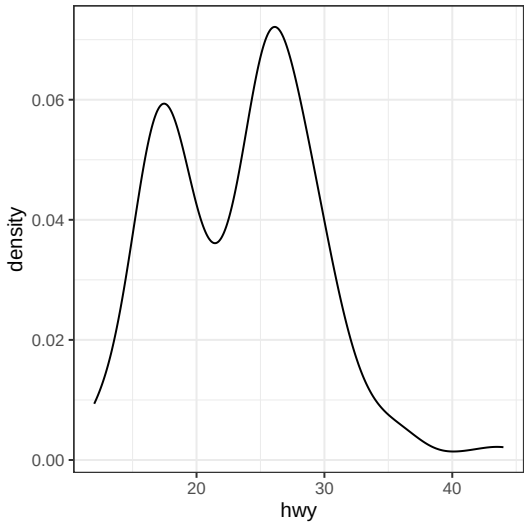
3.4.3 Bar vs. line plots

```
ggplot(mpg, aes(hwy)) +
  geom_bar(stat = "count")
```



Note that the geometry tells ggplot what kind of plot to use, and the statistic (*stat*) tells it what kind of summary to present.

```
ggplot(mpg, aes(hwy)) +
  geom_line(stat = "density")
```

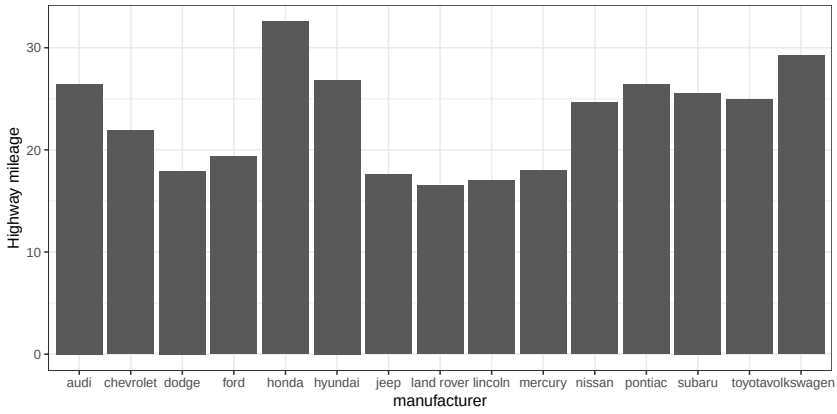


3.5 Plots with two variables

Let's check out mileage by car manufacturer. We'll plot one *continuous* variable by one *nominal* one.

First, let's make a bar plot by choosing the stat "summary" and picking the "mean" function to summarize the data.

```
ggplot(mpg, aes(manufacturer, hwy)) +
  geom_bar(stat = "summary", fun.y = "mean")  +
  ylab('Highway mileage')
```

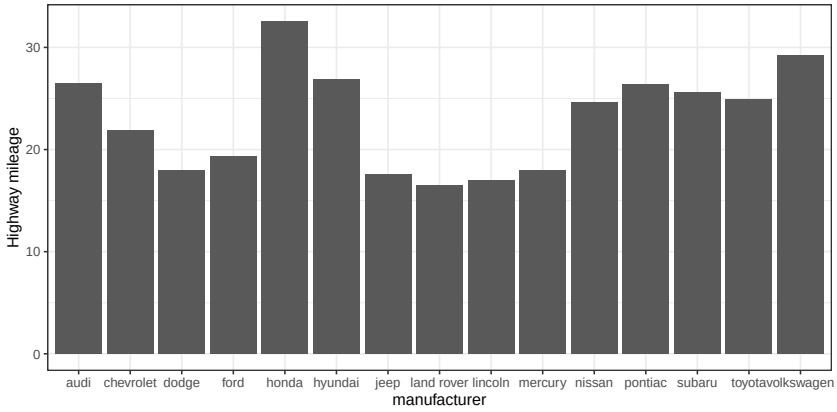


One problem with this plot is that it's hard to read some of the labels because

they overlap. How could we fix that? Hint: search the web for "ggplot rotate x axis labels" and add the appropriate command.

TBD: fix

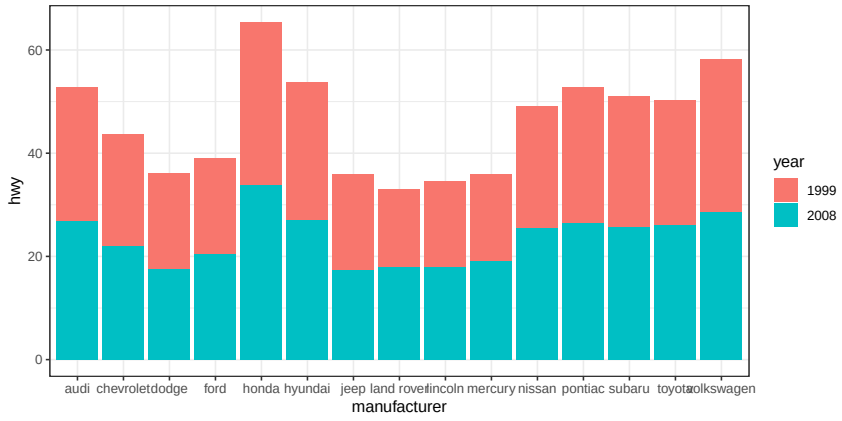
```
ggplot(mpg, aes(manufacturer, hwy)) +
  geom_bar(stat = "summary", fun.y = "mean")  +
  ylab('Highway mileage')
```



3.5.1 Adding on variables

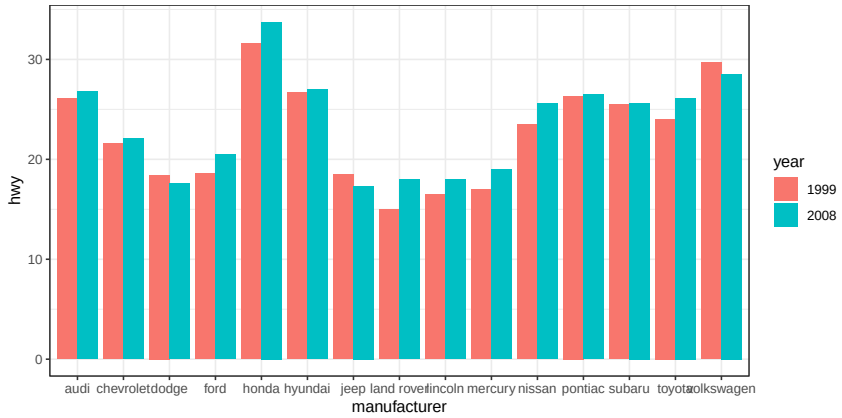
What if we wanted to add another variable into the mix? Maybe the *year* of the car is also important to consider. We have a few options here. First, you could map the variable to another **aesthetic**.

```
# first, year needs to be converted to a factor
mpg$year <- factor(mpg$year)

ggplot(mpg, aes(manufacturer, hwy, fill = year)) +
  geom_bar(stat = "summary", fun.y = "mean")
```

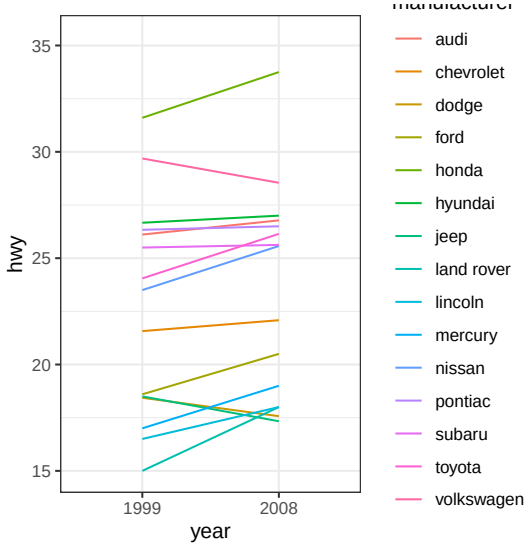


By default, the bars are *stacked* on top of one another. If you want to separate them, you can change the `position` argument form its default to "dodge".

```
ggplot(mpg, aes(manufacturer, hwy, fill=year)) +
  geom_bar(stat = "summary",
           fun.y = "mean",
           position = "dodge")
```

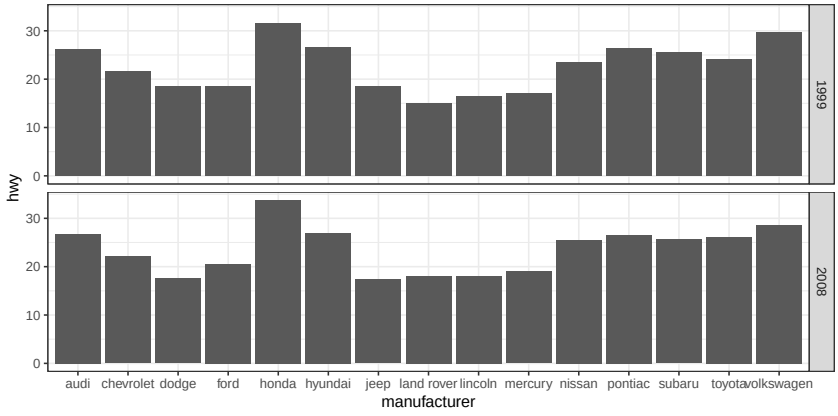



```
ggplot(mpg, aes(year, hwy,
                group=manufacturer,
                color=manufacturer)) +
  geom_line(stat = "summary", fun.y = "mean")
```



For a less visually cluttered plot, let's try **facetting**. This creates *subplots* for each value of the `year` variable.

```
ggplot(mpg, aes(manufacturer, hwy)) +
  # split up the bar plot into two by year
  facet_grid(year ~ .) +
  geom_bar(stat = "summary",
           fun.y = "mean")
```

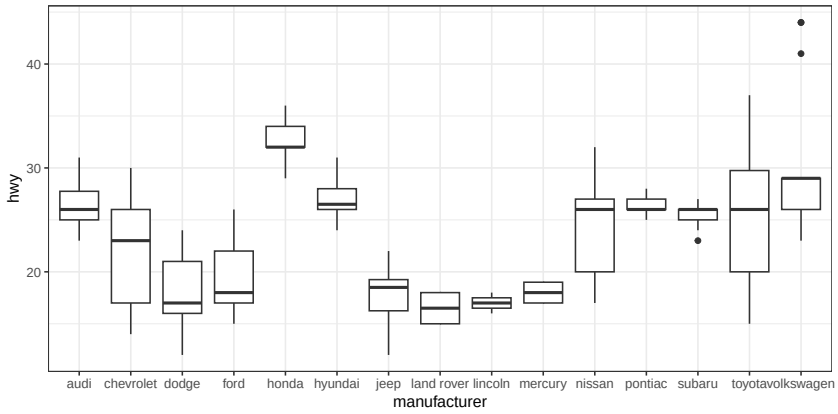


3.5.2 Plotting dispersion

Instead of looking at just the means, we can get a sense of the entire distribution of mileage values for each manufacturer.

3.5.2.1 Box plot

```
ggplot(mpg, aes(manufacturer, hwy)) +
  geom_boxplot()
```



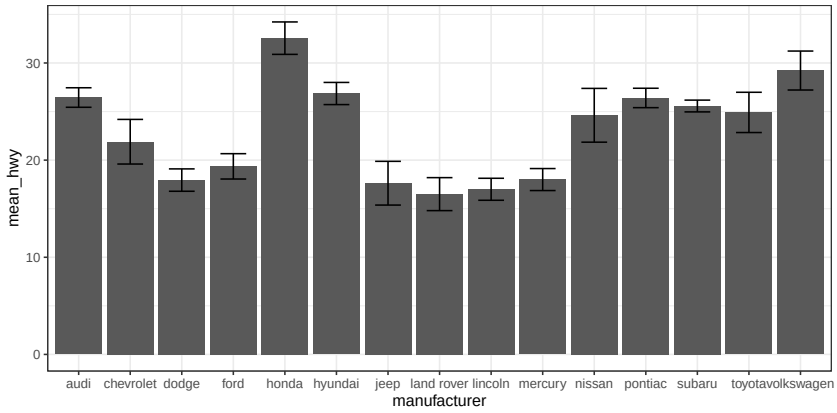
A **box plot** (or box and whiskers plot) uses quartiles to give us a sense of spread. The thickest line, somewhere inside the box, represents the *median*. The upper and lower bounds of the box (the *hinges*) are the first and third quartiles (can you use them to approximate the interquartile range?). The lines extending from the hinges are the remaining data points, excluding **outliers**, which are plotted as individual points.

3.5.2.2 Error bars

Now, let's do something a bit more complex, but much more useful – let's create our own summary of the data, so we can choose which summary statistic to plot and also compute a measure of dispersion of our choosing.

```
# summarise data
mpg_summary <- mpg %>%
  group_by(manufacturer) %>%
  summarise(n = n(),
            mean_hwy = mean(hwy),
```

```
            sd_hwy = sd(hwy))

# compute confidence intervals for the error bars
# (we'll talk about this later in the course!)

limits <- aes(
  # compute the lower limit of the error bar
  ymin = mean_hwy - 1.96 * sd_hwy / sqrt(n),
  # compute the upper limit
  ymax = mean_hwy + 1.96 * sd_hwy / sqrt(n))

# now we're giving ggplot the mean for each group,
# instead of the datapoints themselves

ggplot(mpg_summary, aes(manufacturer, mean_hwy)) +
  # we set stat = "identity" on the summary data
  geom_bar(stat = "identity") +
  # we create error bars using the limits we computed above
  geom_errorbar(limits, width=0.5)
```

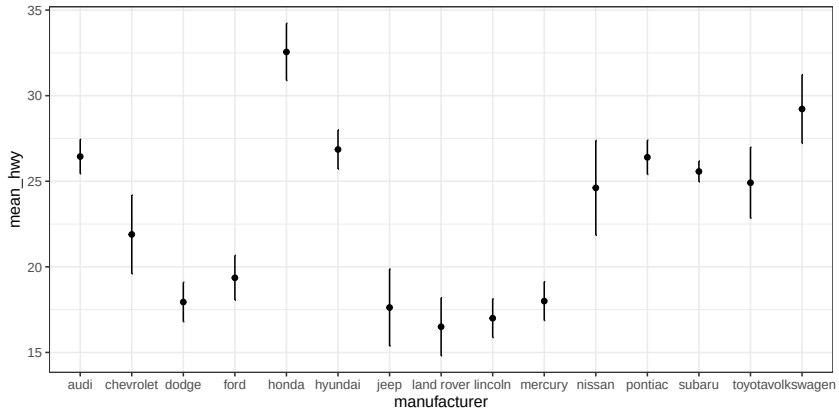


Error bars don't always mean the same thing – it's important to determine whether you're looking at e.g. standard error or confidence intervals (which we'll talk more about later in the course).

3.5.2.2.1 Minimizing non-data ink The plot we just created is nice and all, but it's tough to look at. The bar plots add a lot of ink that doesn't

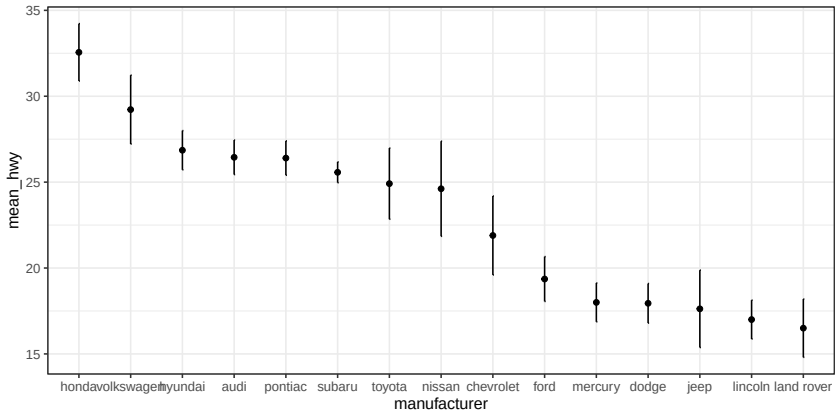
help us compare engine sizes across manufacturers. Similarly, the width of the error bars doesn't add any information. Let's tweak which *geometry* we use, and tweak the appearance of the error bars.

```
ggplot(mpg_summary, aes(manufacturer, mean_hwy)) +
  # switch to point instead of bar to minimize ink used
  geom_point() +
  # remove the horizontal parts of the error bars
  geom_errorbar(limits, width = 0)
```



Looks a lot cleaner, but our points are all over the place. Let's make a final tweak to make *learning something* from this plot a bit easier.

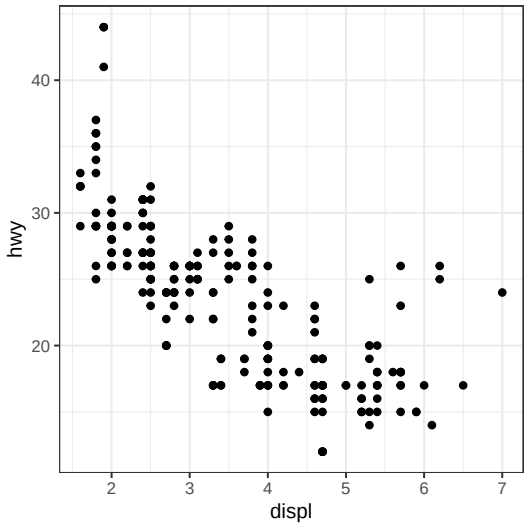
```
mpg_summary_ordered <- mpg_summary %>%
  mutate(
    # we sort manufacturers by mean engine size
    manufacturer = reorder(manufacturer, -mean_hwy)
  )

ggplot(mpg_summary_ordered, aes(manufacturer, mean_hwy)) +
  geom_point() +
  geom_errorbar(limits, width = 0)
```



3.5.3 Scatter plot

When we have multiple *continuous* variables, we can use points to plot each variable on an axis. This is known as a **scatter plot**. You've seen this example in your reading.

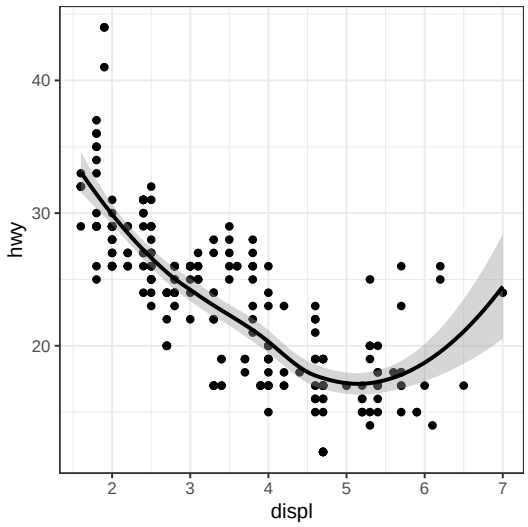
```
ggplot(mpg, aes(displ, hwy)) +
  geom_point()
```



3.5.3.1 Layers of data

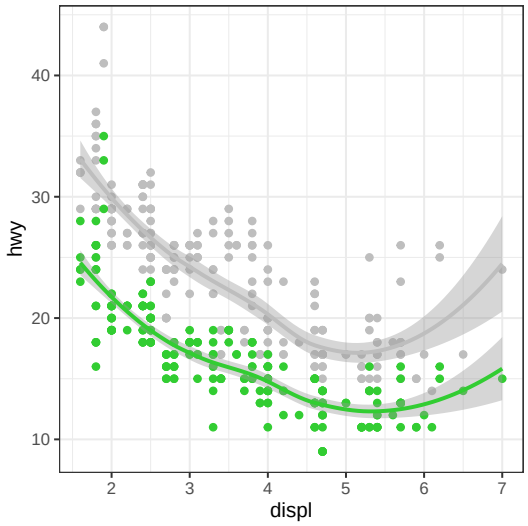
We can add layers of data onto this graph, like a *line of best fit*. We use a geometry known as a **smooth** to accomplish this.

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  geom_smooth(color = "black")
```



We can add on points and a smooth line for another set of data as well (efficiency in the city instead of on the highway).

```
ggplot(mpg) +
  geom_point(aes(displ, hwy), color = "grey") +
  geom_smooth(aes(displ, hwy), color = "grey") +
  geom_point(aes(displ, cty), color = "limegreen") +
  geom_smooth(aes(displ, cty), color = "limegreen")
```



3.6 Creating a more complex plot

In this section we will recreate Figure 3.1 from Chapter 3. Here is the code to generate the figure; we will go through each of its sections below.

```
oringDf <- read.table("data/orings.csv", sep = ",",
                      header = TRUE)

oringDf %>%
  ggplot(aes(x = Temperature, y = DamageIndex)) +
  geom_point() +
  geom_smooth(method = "loess",
              se = FALSE, span = 1) +
  ylim(0, 12) +
  geom_vline(xintercept = 27.5, size =8,
             alpha = 0.3, color = "red") +
  labs(
    y = "Damage Index",
    x = "Temperature at time of launch"
  ) +
  scale_x_continuous(breaks = seq.int(25, 85, 5)) +
  annotate(
    "text",
```

```
    angle=90,
    x = 27.5,
    y = 6,
    label = "Forecasted temperature on Jan 28",
    size = 5
  )
```

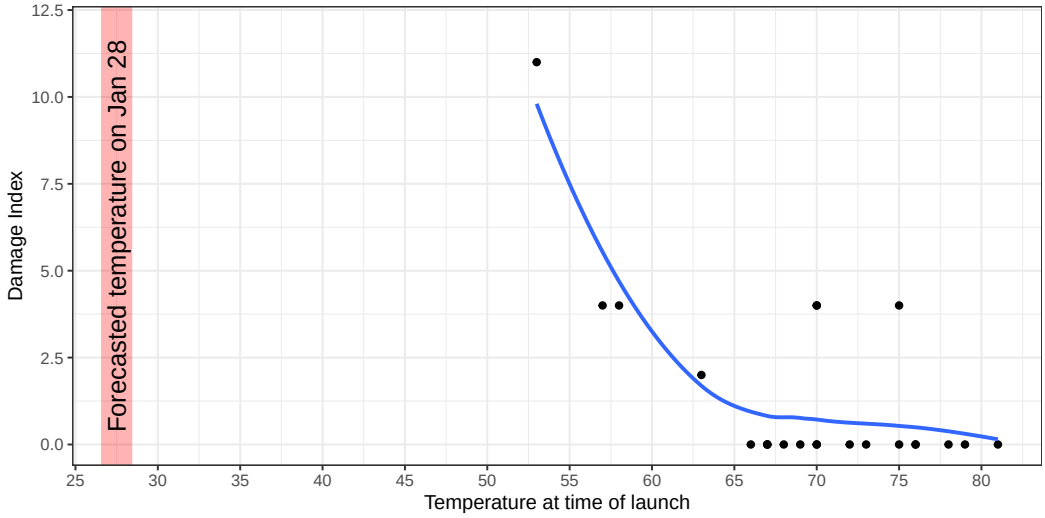


Figure 3.1: Damage index at different temperature.

3.7 Additional reading and resources

- ggplot theme reference
- knockoff tech themes

Chapter 4

Fitting simple models using R

In this chapter we will focus on how to compute the measures of central tendency and variability that were covered in the previous chapter. Most of these can be computed using a built-in R function, but we will show how to do them manually in order to give some intuition about how they work.

4.1 Mean

The mean is defined as the sum of values divided by the number of values being summed:

$$\bar{X} = \frac{\sum_{i=1}^{n} x_i}{n}$$

Let's say that we want to obtain the mean height for adults in the NHANES database (contained in the data `Height`). We would sum the individual heights (using the `sum()` function) and then divide by the number of values:

```
sum(NHANES$Height) / length(NHANES$Height)
```

```
## [1] NA
```

This returns the value NA, because there are missing values for some rows, and the `sum()` function doesn't automatically handle those. To address this,

we could filter the data frame using `drop_na()` to drop rows with NA values for this variable:

```
height_noNA <- NHANES %>%
  drop_na(Height) %>%
  pull(Height)

sum(height_noNA) / length(height_noNA)
```

```
## [1] 160.3516
```

There is, of course, a built-in function in R called `mean()` that will compute the mean. Like the `sum()` function, `mean()` will return NA if there are any NA values in the data:

```
mean(NHANES$Height)
```

```
## [1] NA
```

The `mean()` function includes an optional argument called `na.rm` that will remove NA values if it is set to TRUE:

```
mean(NHANES$Height, na.rm=TRUE)
```

```
## [1] 160.3516
```

4.2 Median

The median is the middle value after sorting the entire set of values. Let's use the cleand-up `height_noNA` variable created above to determine this for the NHANES height data. First we sort the data in order of their values:

```
height_sorted <- sort(height_noNA)
```

Next we find the median value. If there is an odd number of values in the list, then this is just the value in the middle, whereas if the number of values is even then we take the average of the two middle values. We can determine whether the number of items is even by dividing the length by two and seeing if there is a remainder; we do this using the `%%` operator, which is known as the *modulus* and returns the remainder:

```
5 %% 2
```

```
## [1] 1
```

Here we will test whether the remainder is equal to one; if it is, then we will take the middle value, otherwise we will take the average of the two middle values. We can do this using an if/else structure, which executes different processes depending on which of the arguments are true:

```
if (logical value) {
  functions to perform if logical value is true
} else {
  functions to perform if logical value is false

}
```

Let's do this with our data. To find the middle value when the number of items is odd, we will divide the length and then round up, using the `ceiling()` function:

```
if (length(height_sorted) %% 2 == 1){
  # length of vector is odd
  median_height <-
    height_sorted[ceiling(length(height_sorted) / 2)]
} else {
  median_height <-
    (height_sorted[length(height_sorted) / 2] +
        height_sorted[1 + length(height_sorted) / (2)])/2
}

median_height
```

```
## [1] 165.1
```

We can compare this to the result from the built-in median function:

```
median(height_noNA)
```

```
## [1] 165.1
```

4.3 Mode

The mode is the most frequent value that occurs in a variable. R has a function called `mode()` but if you look at the help page you will see that it doesn't actually copute the mode. In fact, R doesn't have a built-in function to compute the mode, so we need to create one. Let start with some toy data:

```
mode_test = c('a', 'b', 'b', 'c', 'c', 'c')
mode_test
```

```
## [1] "a" "b" "b" "c" "c" "c"
```

We can see by eye that the mode is "a" since it occurs more often than the others. To find it computationally, let's first get the unique values

To do this, we first create a table with the counts for each value, using the `table()` function:

```
mode_table <- table(mode_test)
mode_table
```

```
## mode_test
## a b c
## 1 2 3
```

Now we need to find the maximum value. We do this by comparing each value to the maximum of the table; this will work even if there are multiple values with the same frequency (i.e. a tie for the mode).

```
table_max <- mode_table[mode_table == max(mode_table)]
table_max
```

```
## c
## 3
```

This variable is a special kind of value called a *named vector*, and its name contains the value that we need to identify the mode. We can pull it out using the `names()` function:

```
my_mode <- names(table_max)[1]
my_mode
```

```
## [1] "c"
```

Let's wrap this up into our own custom function:

```
getmode <- function(v, print_table=FALSE) {
  mode_table <- table(v)
  if (print_table){
    print(kable(mode_table))
  }
  table_max <- mode_table[mode_table == max(mode_table)]
  return(names(table_max))
}
```

We can then apply this to real data. Let's apply this to the `MaritalStatus` variable in the NHANES dataset:

```
getmode(NHANES$MaritalStatus)
```

```
## [1] "Married"
```

4.4 Variability

Let's first compute the *variance*, which is the average squared difference between each value and the mean. Let's do this with our cleaned-up version of the height data, but instead of working with the entire dataset, let's take a random sample of 150 individuals:

```
height_sample <- NHANES %>%
  drop_na(Height) %>%
  sample_n(150) %>%
  pull(Height)
```

First we need to obtain the sum of squared errors from the mean. In R, we can square a vector using `**2`:

```
SSE <- sum((height_sample - mean(height_sample))**2)
SSE
```

```
## [1] 63419.37
```

Then we divide by N - 1 to get the estimated variance:

```
var_est <- SSE/(length(height_sample) - 1)
var_est
```

```
## [1] 425.6333
```

We can compare this to the built-in `var()` function:

```
var(height_sample)
```

```
## [1] 425.6333
```

We can get the *standard deviation* by simply taking the square root of the variance:

```
sqrt(var_est)
```

```
## [1] 20.63088
```

Which is the same value obtained using the built-in `sd()` function:

```
sd(height_sample)
```

```
## [1] 20.63088
```

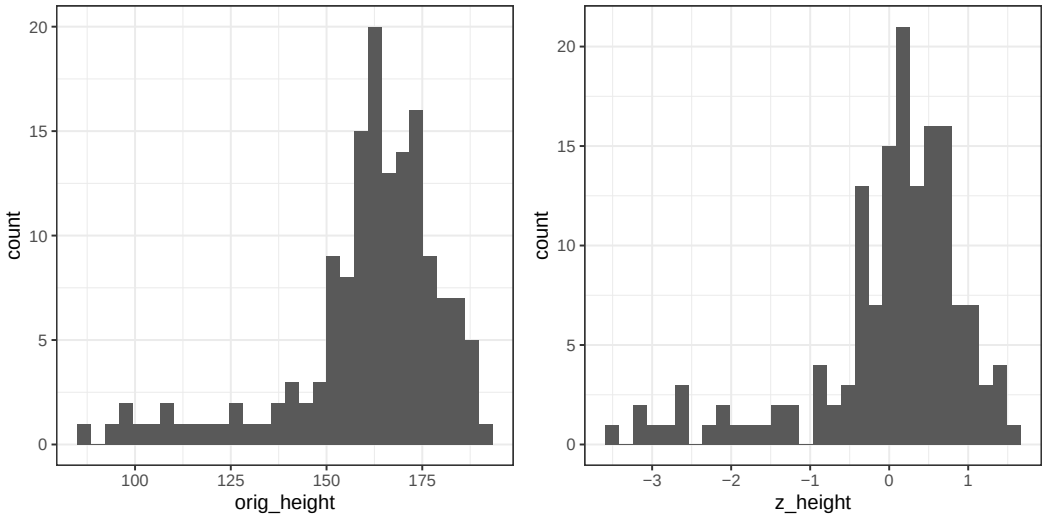
4.5 Z-scores

A Z-score is obtained by first subtracting the mean and then dividing by the standard deviation of a distribution. Let's do this for the `height_sample` data.

```
mean_height <- mean(height_sample)
sd_height <- sd(height_sample)

z_height <- (height_sample - mean_height)/sd_height
```

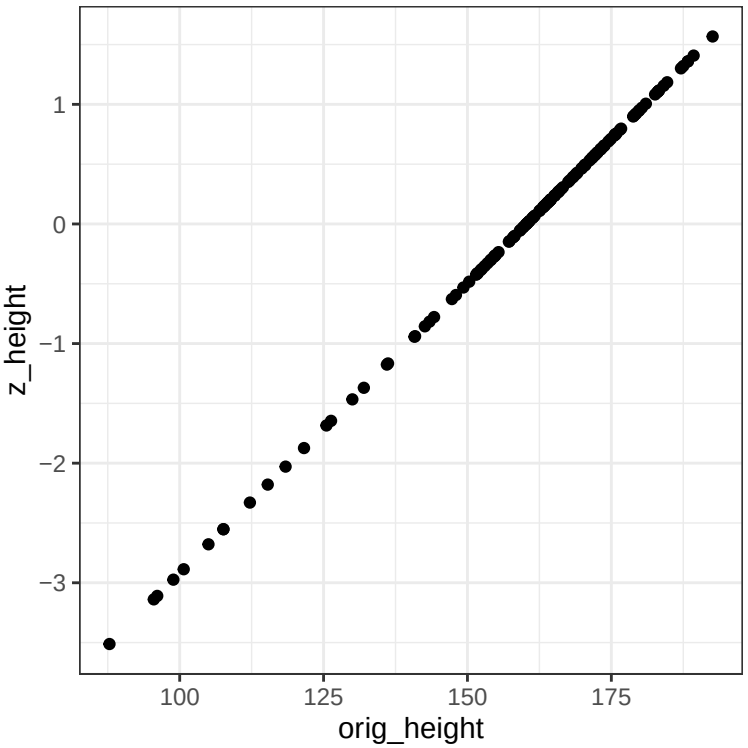
Now let's plot the histogram of Z-scores alongside the histogram for the original values. We will use the `plot_grid()` function from the `cowplot` library to plot the two figures alongside one another. First we need to put the values into a data frame, since `ggplot()` requires the data to be contained in a data frame.

```
height_df <- data.frame(orig_height=height_sample,
                        z_height=z_height)

# create individual plots
plot_orig <- ggplot(height_df, aes(orig_height)) +
  geom_histogram()
plot_z <- ggplot(height_df, aes(z_height)) +
  geom_histogram()

# combine into a single figure
plot_grid(plot_orig, plot_z)
```



You will notice that the shapes of the histograms are similar but not exactly the same. This occurs because the binning is slightly different between the two sets of values. However, if we plot them against one another in a scatterplot, we will see that there is a direct linear relation between the two sets of values:

```
ggplot(height_df, aes(orig_height, z_height)) +
  geom_point()
```


z_height
1
0
−1
−2
−3
100
125
150
175
orig_height

Chapter 5

Probability in R (with Lucy King)

In this chapter we will go over probability computations in R.

5.1 Basic probability calculations

Let's create a vector of outcomes from one to 6, using the `seq()` function to create such a sequence:

```
outcomes <- seq(1, 6)
outcomes
```

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

Now let's create a vector of logical values based on whether the outcome in each position is equal to 1. Remember that `==` tests for equality of each element in a vector:

```
outcome1isTrue <- outcomes == 1
outcome1isTrue
```

```
## [1]  TRUE FALSE FALSE FALSE FALSE FALSE
```

Remember that the simple probability of an outcome is number of occurrences of the outcome divided by the total number of events. To compute a probability, we can take advantage of the fact that TRUE/FALSE are equivalent

to 1/0 in R. The formula for the mean (sum of values divided by the number of values) is thus exactly the same as the formula for the simple probability! So, we can compute the probability of the event by simply taking the mean of the logical vector.

```
p1isTrue <- mean(outcome1isTrue)
p1isTrue
```

```
## [1] 0.1666667
```

5.2 Empirical frequency (Section 5.2)

Let's walk through how we computed empirical frequency of rain in San Francisco.

First we load the data:

```
# we will remove the STATION and NAME variables
# since they are identical for all rows
SFrain <- read_csv("data/SanFranciscoRain/1329219.csv") %>%
  dplyr::select(-STATION, -NAME)

glimpse(SFrain)
```

```
## Rows: 365
## Columns: 2
## $ DATE <date> 2017-01-01, 2017-01-02, 2017-01-03, 2017-01-~
## $ PRCP <dbl> 0.05, 0.10, 0.40, 0.89, 0.01, 0.00, 0.82, 1.4~
```

We see that the data frame contains a variable called `PRCP` which denotes the amount of rain each day. Let's create a new variable called `rainToday` that denotes whether the amount of precipitation was above zero:

```
SFrain <-
  SFrain %>%
  mutate(rainToday = as.integer(PRCP > 0))

glimpse(SFrain)
```

```
## Rows: 365
```

```
## Columns: 3
## $ DATE      <date> 2017-01-01, 2017-01-02, 2017-01-03, 201~
## $ PRCP      <dbl> 0.05, 0.10, 0.40, 0.89, 0.01, 0.00, 0.82~
## $ rainToday <int> 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0~
```

Now we will summarize the data to compute the probability of rain:

```
pRainInSF <-
  SFrain %>%
  summarize(
    pRainInSF = mean(rainToday)
  ) %>%
  pull()

pRainInSF
```

```
## [1] 0.2
```

5.3 Conditional probability (Section 5.3)

Let's determine the conditional probability of someone being unhealthy, given that they are over 70 years of age, using the NHANES dataset. Let's create a new data frame that only contains people over 70 years old.

```
healthDataFrame <-
  NHANES %>%
  mutate(
    Over70 = Age > 70,
    Unhealthy = DaysPhysHlthBad > 0
  ) %>%
  dplyr::select(Unhealthy, Over70) %>%
  drop_na()

glimpse(healthDataFrame)
```

```
## Rows: 4,891
## Columns: 2
## $ Unhealthy <lgl> FALSE, FALSE, FALSE, TRUE, FALSE, TRUE, ~
## $ Over70    <lgl> FALSE, FALSE, FALSE, FALSE, FALSE, FALSE~
```

First, what's the probability of being over 70?

```
pOver70 <-
  healthDataFrame %>%
  summarise(pOver70 = mean(Over70)) %>%
  pull()

# to obtain the specific value, we need to extract it from the data frame

pOver70
```

```
## [1] 0.1106113
```

Second, what's the probability of being unhealthy?

```
pUnhealthy <-
  healthDataFrame %>%
  summarise(pUnhealthy = mean(Unhealthy)) %>%
  pull()

pUnhealthy
```

```
## [1] 0.3567778
```

What's the probability for each combination of unhealthy/healthy and over 70/ not? We can create a new variable that finds the joint probability by multiplying the two individual binary variables together; since anything times zero is zero, this will only have the value 1 for any case where both are true.

```
pBoth <- healthDataFrame %>%
  mutate(
    both = Unhealthy*Over70
  ) %>%
  summarise(
    pBoth = mean(both)) %>%
  pull()

pBoth
```

```
## [1] 0.04252709
```

Finally, what's the probability of someone being unhealthy, given that they are over 70 years of age?

```
pUnhealthyGivenOver70 <-
  healthDataFrame %>%
  filter(Over70 == TRUE) %>% # limit to Over70
  summarise(pUnhealthy = mean(Unhealthy)) %>%
  pull()

pUnhealthyGivenOver70
```

```
## [1] 0.3844732
```

```
# compute the opposite:
# what the probability of being over 70 given that
# one is unhealthy?
pOver70givenUnhealthy <-
  healthDataFrame %>%
  filter(Unhealthy == TRUE) %>% # limit to Unhealthy
  summarise(pOver70 = mean(Over70)) %>%
  pull()

pOver70givenUnhealthy
```

```
## [1] 0.1191977
```

Chapter 6

Sampling in R

First we load the necessary libraries and set up the NHANES adult dataset

```
library(tidyverse)
library(ggplot2)
library(knitr)
library(cowplot)

set.seed(123456)
opts_chunk$set(tidy.opts=list(width.cutoff=80))
options(tibble.width = 60)


# load the NHANES data library
library(NHANES)

# create a NHANES dataset without duplicated IDs
NHANES <-
  NHANES %>%
  distinct(ID, .keep_all = TRUE)

#create a dataset of only adults
NHANES_adult <-
  NHANES %>%
  filter(
```

```
    Age >= 18
  ) %>%
  drop_na(Height)
```

6.1 Sampling error (Section 6.1)

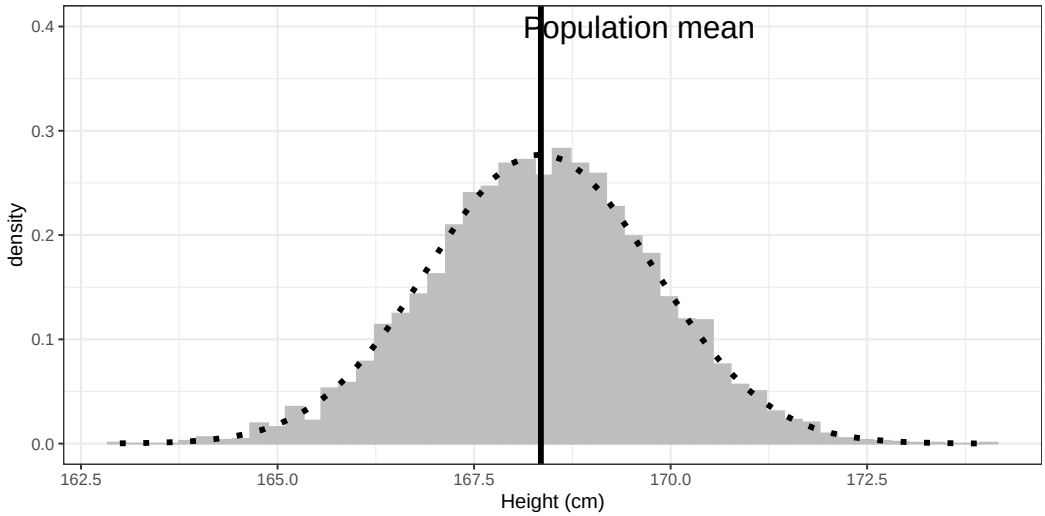
Here we will repeatedly sample from the NHANES Height variable in order to obtain the sampling distribution of the mean.

```
sampSize <- 50 # size of sample
nsamps <- 5000 # number of samples we will take

# set up variable to store all of the results
sampMeans <- tibble(meanHeight=rep(NA,nsamps))

# Loop through and repeatedly sample and compute the mean
for (i in 1:nsamps) {
  sampMeans$meanHeight[i] <- NHANES_adult %>%
    sample_n(sampSize) %>%
    summarize(meanHeight=mean(Height)) %>%
    pull(meanHeight)
}
```

Now let's plot the sampling distribution. We will also overlay the sampling distribution of the mean predicted on the basis of the population mean and standard deviation, to show that it properly describes the actual sampling distribution.

```
# pipe the sampMeans data frame into ggplot
sampMeans %>%
  ggplot(aes(meanHeight)) +
  # create histogram using density rather than count
  geom_histogram(
    aes(y = ..density..),
    bins = 50,
    col = "gray",
    fill = "gray"
  ) +
```

```
  # add a vertical line for the population mean
  geom_vline(xintercept = mean(NHANES_adult$Height),
             size=1.5) +
  # add a label for the line
  annotate(
    "text",
    x = 169.6,
    y = .4,
    label = "Population mean",
    size=6
  ) +
  # label the x axis
  labs(x = "Height (cm)") +
  # add normal based on population mean/sd
  stat_function(
      fun = dnorm, n = sampSize,
      args = list(
        mean = mean(NHANES_adult$Height),
        sd = sd(NHANES_adult$Height) / sqrt(sampSize)
      ),
      size = 1.5,
      color = "black",
      linetype='dotted'
    )
```



6.2 Central limit theorem

The central limit theorem tells us that the sampling distribution of the mean becomes normal as the sample size grows. Let's test this by sampling a clearly non-normal variable and look at the normality of the results using a Q-Q plot. We saw in Figure 6.1 that the variable `AlcoholYear` is distributed in a very non-normal way. Let's first look at the Q-Q plot for these data, to see what it looks like. We will use the `stat_qq()` function from `ggplot2` to create the plot for us.

```
# prepare the dta
NHANES_cleanAlc <- NHANES %>%
  drop_na(AlcoholYear)

ggplot(NHANES_cleanAlc, aes(sample=AlcoholYear)) +
  stat_qq() +
  # add the line for x=y
  stat_qq_line()
```

We can see from this figure that the distribution is highly non-normal, as the Q-Q plot diverges substantially from the unit line.

Now let's repeatedly sample and compute the mean, and look at the resulting Q-Q plot. We will take samples of various sizes to see the effect of sample

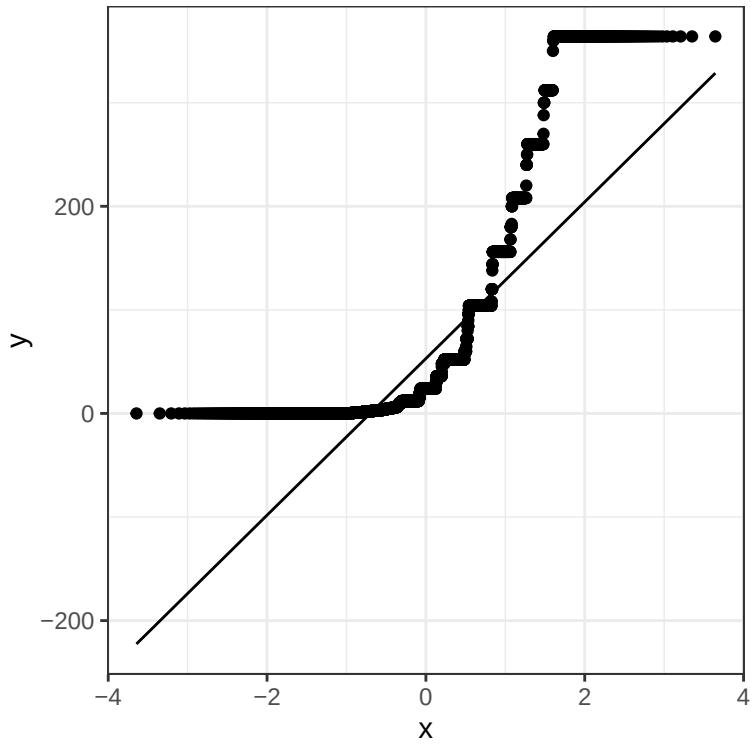


Figure 6.1: Q-Q plot for the variable `AlcoholYear`.

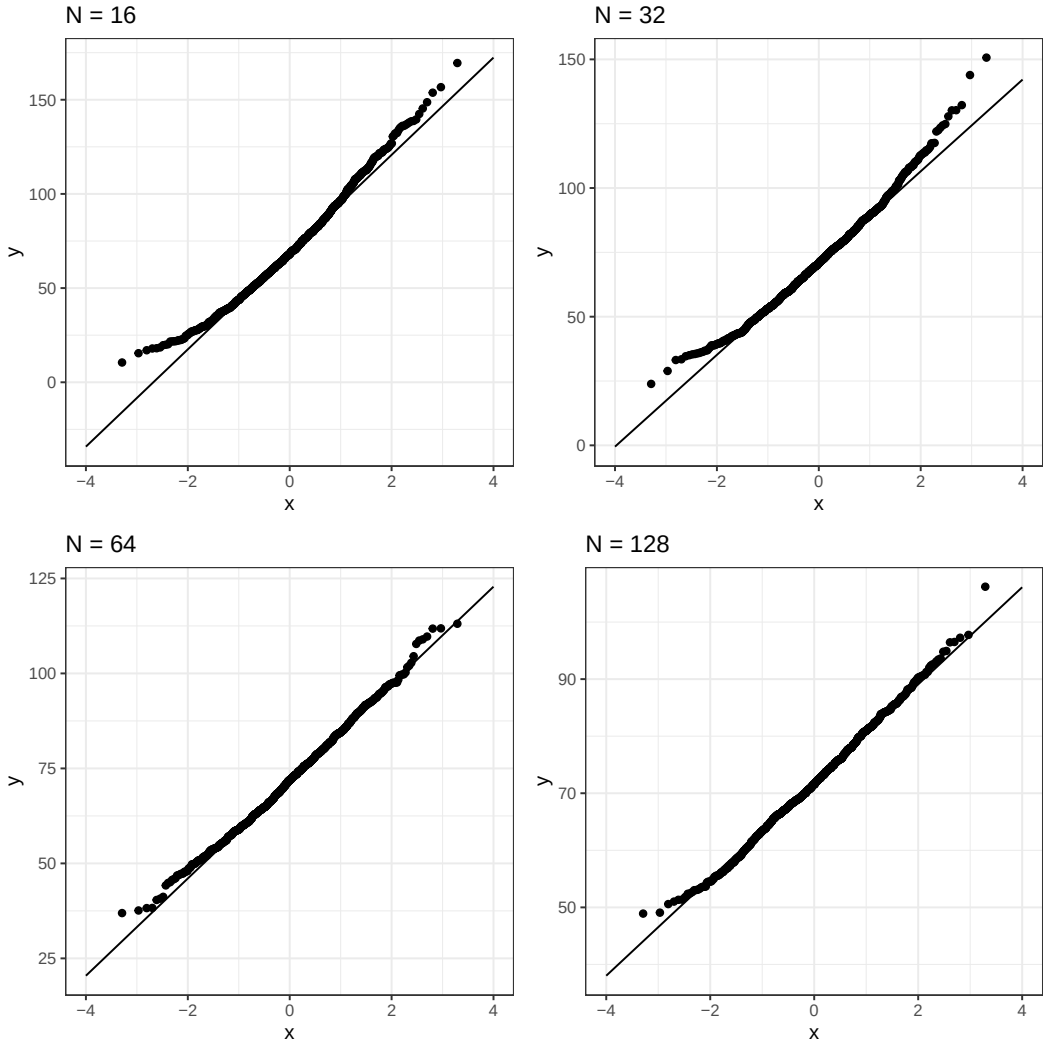
size. We will use a function from the `dplyr` package called `do()`, which can run a large number of analyses at once.

```
set.seed(12345)

sampSizes <- c(16, 32, 64, 128) # size of sample
nsamps <- 1000 # number of samples we will take

# create the data frame that specifies the analyses
input_df <- tibble(sampSize=rep(sampSizes, nsamps),
                       id=seq(nsamps * length(sampSizes)))


# create a function that samples and returns the mean
# so that we can loop over it using replicate()
get_sample_mean <- function(sampSize){
  meanAlcYear <-
    NHANES_cleanAlc %>%
    sample_n(sampSize) %>%
    summarize(meanAlcoholYear = mean(AlcoholYear)) %>%
    pull(meanAlcoholYear)
  return(tibble(meanAlcYear = meanAlcYear, sampSize=sampSize))
}

# loop through sample sizes
# we group by id so that each id will be run separately by do()
all_results = input_df %>%
  group_by(id) %>%
  # "." refers to the data frame being passed in by do()
  do(get_sample_mean(.$sampSize))
```

Now let's create separate Q-Q plots for the different sample sizes.

```
# create empty list to store plots

qqplots = list()

for (N in sampSizes){
  sample_results <-
```

```
    all_results %>%
    filter(sampSize==N)

  qqplots[[toString(N)]] <- ggplot(sample_results,
                                   aes(sample=meanAlcYear)) +
    stat_qq() +
    # add the line for x=y
    stat_qq_line(fullrange = TRUE) +
    ggtitle(sprintf('N = %d', N)) +
   xlim(-4, 4)

}

plot_grid(plotlist = qqplots)
```



This shows that the results become more normally distributed (i.e. following the straight line) as the samples get larger.

6.3 Confidence intervals (Section 6.3)

Remember that confidence intervals are intervals that will contain the population parameter on a certain proportion of times. In this example we will walk through the simulation that was presented in Section 6.3 to show that this actually works properly. Here we will use a function called `do()` that lets us

Chapter 7

Resampling and simulation in R

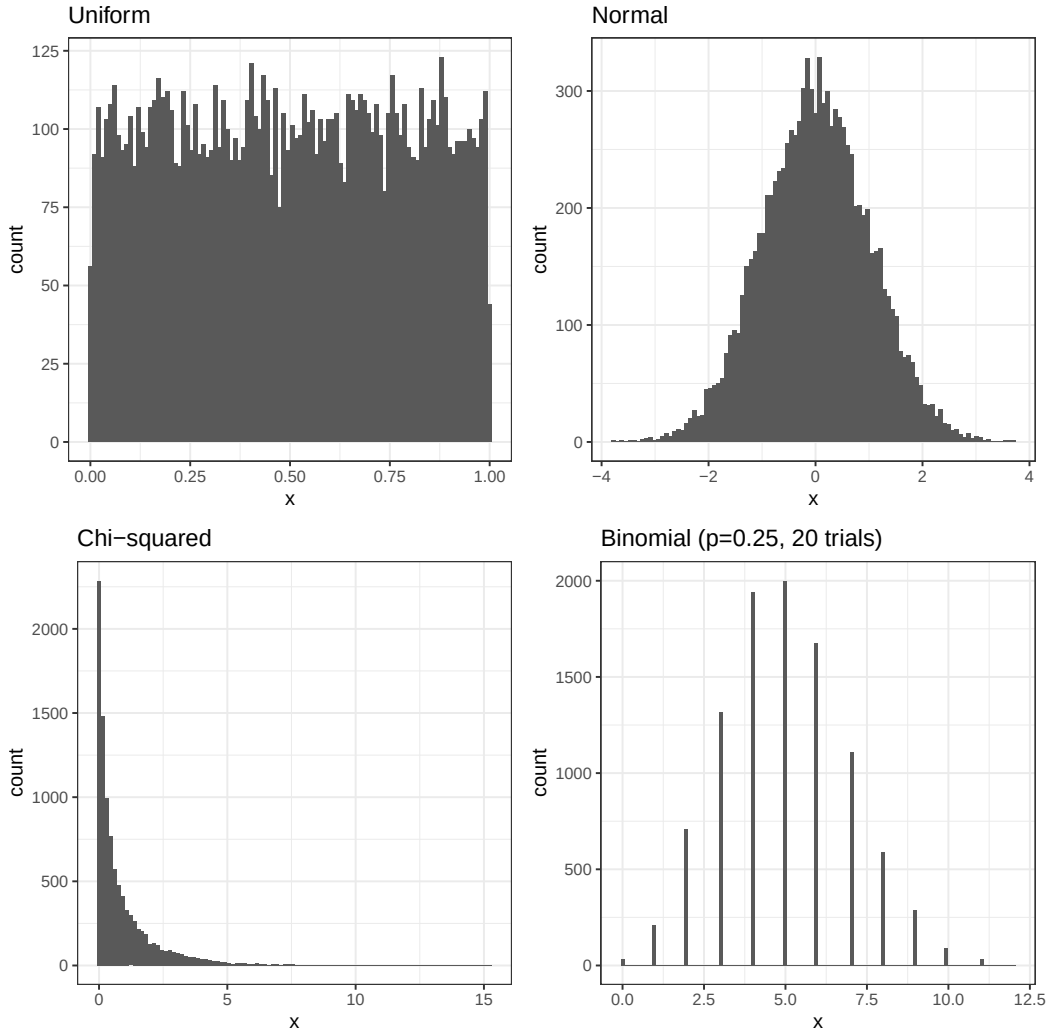
In this chapter we will use R to undestand how to resample data and perform numerical simulations.

7.1 Generating random samples (Section 7.1)

Here we will generate random samples from a number of different distributions and plot their histograms.

```
nsamples <- 10000
nhistbins <- 100

# uniform distribution

p1 <-
  tibble(
    x = runif(nsamples)
  ) %>%
  ggplot((aes(x))) +
  geom_histogram(bins = nhistbins) +
  labs(title = "Uniform")

# normal distribution
p2 <-
```

```
  tibble(
    x = rnorm(nsamples)
  ) %>%
  ggplot(aes(x)) +
  geom_histogram(bins = nhistbins) +
  labs(title = "Normal")

# Chi-squared distribution
p3 <-
  tibble(
    x = rnorm(nsamples)
  ) %>%
  ggplot(aes(x)) +
  geom_histogram(bins = nhistbins) +
  labs(title = "Normal")

# Chi-squared distribution
p3 <-
  tibble(
    x = rchisq(nsamples, df=1)
  ) %>%
  ggplot(aes(x)) +
  geom_histogram(bins = nhistbins) +
  labs(title = "Chi-squared")

# Poisson distribution
p4 <-
  tibble(
    x = rbinom(nsamples, 20, 0.25)
  ) %>%
  ggplot(aes(x)) +
  geom_histogram(bins = nhistbins) +
  labs(title = "Binomial (p=0.25, 20 trials)")


plot_grid(p1, p2, p3, p4, ncol = 2)
```

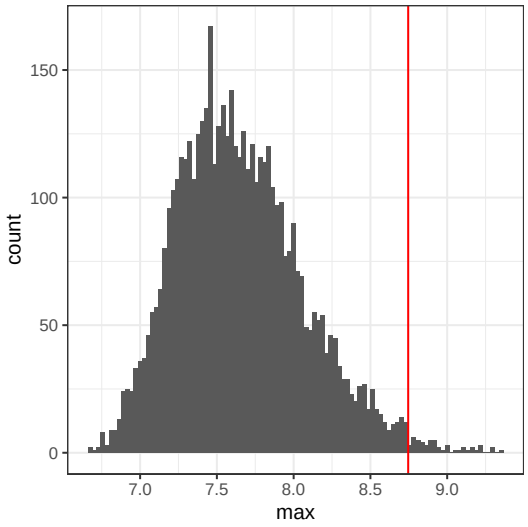


7.2 Simulating the maximum finishing time

Let's simulate 150 samples, collecting the maximum value from each sample, and then plotting the distribution of maxima.

```
# sample maximum value 5000 times and compute 99th percentile
nRuns <- 5000
sampSize <- 150

sampleMax <- function(sampSize = 150) {
```

```
  samp <- rnorm(sampSize, mean = 5, sd = 1)
  return(tibble(max=max(samp)))
}

input_df <- tibble(id=seq(nRuns)) %>%
  group_by(id)

maxTime <- input_df %>% do(sampleMax())

cutoff <- quantile(maxTime$max, 0.99)


ggplot(maxTime,aes(max)) +
  geom_histogram(bins = 100) +
  geom_vline(xintercept = cutoff, color = "red")
```



7.3 The bootstrap

The bootstrap is useful for creating confidence intervals in cases where we don't have a parametric distribution. One example is for the median; let's look at how that works. We will start by implementing it by hand, to see more closely how it works. We will start by collecting a sample of individuals

from the NHANES dataset, and the using the bootstrap to obtain confidence intervals on the median for the Height variable.

Lower CI limit	Median	Upper CI limit
161.6	167.65	171.1

Chapter 8

Hypothesis testing in R

In this chapter we will present several examples of using R to perform hypothesis testing.

8.1 Simple example: Coin-flipping (Section 8.1)

Let's say that we flipped 100 coins and observed 70 heads. We would like to use these data to test the hypothesis that the true probability is 0.5.

First let's generate our data, simulating 100,000 sets of 100 flips. We use such a large number because it turns out that it's very rare to get 70 heads, so we need many attempts in order to get a reliable estimate of these probabilties. This will take a couple of minutes to complete.

```
# simulate tossing of 100,000 flips of 100 coins to identify
# empirical probability of 70 or more heads out of 100 flips

nRuns <- 100000

# create function to toss coins
tossCoins <- function() {
  flips <- runif(100) > 0.5
  return(tibble(nHeads=sum(flips)))
```

```
}

# create an input data frame for do()
input_df <- tibble(id=seq(nRuns)) %>%
  group_by(id)

# use do() to perform the coin flips
flip_results <- input_df %>%
  do(tossCoins()) %>%
  ungroup()

p_ge_70_sim <-
  flip_results %>%
  summarise(p_gt_70 = mean(nHeads >= 70)) %>%
  pull()

p_ge_70_sim
```

```
## [1] 3e-05
```

For comparison, we can also compute the p-value for 70 or more heads based on a null hypothesis of $P_{heads} = 0.5$, using the binomial distribution.

```
# compute the probability of 69 or fewer heads,
# when P(heads)=0.5
p_lt_70 <- pbinom(69, 100, 0.5)

# the probability of 70 or more heads is simply
# the complement of p_lt_70
p_ge_70 <- 1 - p_lt_70

p_ge_70
```

```
## [1] 3.92507e-05
```

8.2 Simulating p-values

In this exercise we will perform hypothesis testing many times in order to test whether the p-values provided by our statistical test are valid. We will sample data from a normal distribution with a mean of zero, and for each sample perform a t-test to determine whether the mean is different from zero. We will then count how often we reject the null hypothesis; since we know that the true mean is zero, these are by definition Type I errors.

```
nRuns <- 5000

# create input data frame for do()
input_df <- tibble(id=seq(nRuns)) %>%
  group_by(id)

# create a function that will take a sample
# and perform a one-sample t-test

sample_ttest <- function(sampSize=32){
  tt.result <- t.test(rnorm(sampSize))
  return(tibble(pvalue = tt.result$p.value))
}

# perform simulations

sample_ttest_result <- input_df %>%
  do(sample_ttest())

p_error <-
  sample_ttest_result %>%
  ungroup() %>%
  summarize(p_error = mean(pvalue<.05)) %>%
  pull()

p_error
```

```
## [1] 0.0478
```

We should see that the proportion of samples with $p < .05$ is about 5%.

Chapter 9

Statistical power in R

In this chapter we focus on effect size and statistical power.

9.1 Computing confidence intervals

9.1.1 Theoretical

9.1.2 Bootstrap

9.2 Effect Size

9.2.1 Cohen's d

9.2.2 Pearson's r

9.2.3 Odds ratio

9.3 Power analysis

We can compute a power analysis using functions from the `pwr` package. Let's focus on the power for a t-test in order to determine a difference in the mean between two groups. Let's say that we think than an effect size of Cohen's d=0.5 is realistic for the study in question (based on previous research) and would be of scientific interest. We wish to have 80% power to find the effect

if it exists. We can compute the sample size needed for adequate power using the `pwr.t.test()` function:

```
pwr.t.test(d=0.5, power=.8)
```

```
##
##      Two-sample t test power calculation
##
##               n = 63.76561
##               d = 0.5
##       sig.level = 0.05
##           power = 0.8
##     alternative = two.sided
##
## NOTE: n is number in *each* group
```

Thus, about 64 participants would be needed in each group in order to test the hypothesis with adequate power.

9.4 Power curves

We can also create plots that can show us how the power to find an effect varies as a function of effect size and sample size. We willl use the `crossing()` function from the `tidyr` package to help with this. This function takes in two vectors, and returns a tibble that contains all possible combinations of those values.

```
effect_sizes <- c(0.2, 0.5, 0.8)
sample_sizes = seq(10, 500, 10)

#
input_df <- crossing(effect_sizes,sample_sizes)
glimpse(input_df)
```

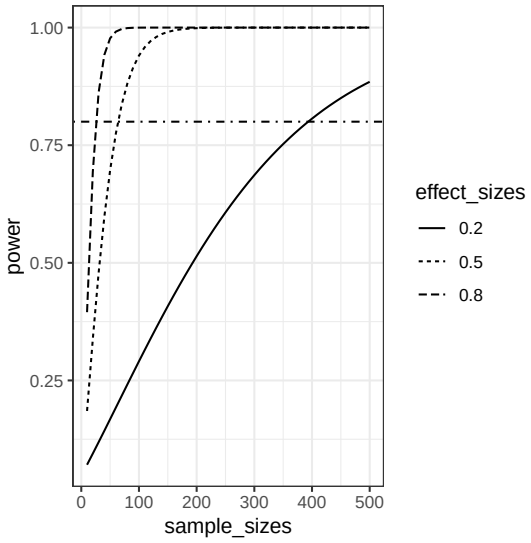
```
## Rows: 150
## Columns: 2
## $ effect_sizes <dbl> 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.~
## $ sample_sizes <dbl> 10, 20, 30, 40, 50, 60, 70, 80, 90, 1~
```

Using this, we can then perform a power analysis for each combination of effect size and sample size to create our power curves. In this case, let's say that we wish to perform a two-sample t-test.

```
# create a function get the power value and
# return as a tibble
get_power <- function(df){
  power_result <- pwr.t.test(n=df$sample_sizes,
                             d=df$effect_sizes,
                             type='two.sample')
  df$power=power_result$power
  return(df)
}

# run get_power for each combination of effect size
# and sample size

power_curves <- input_df %>%
  do(get_power(.)) %>%
  mutate(effect_sizes = as.factor(effect_sizes))
```

Now we can plot the power curves, using a separate line for each effect size.

```
ggplot(power_curves,
       aes(x=sample_sizes,
           y=power,
           linetype=effect_sizes)) +
  geom_line() +
  geom_hline(yintercept = 0.8,
             linetype='dotdash')
```



9.5 Simulating statistical power

Let's simulate this to see whether the power analysis actually gives the right answer. We will sample data for two groups, with a difference of 0.5 standard deviations between their underlying distributions, and we will look at how often we reject the null hypothesis.

```
nRuns <- 5000
effectSize <- 0.5
# perform power analysis to get sample size
pwr.result <- pwr.t.test(d=effectSize, power=.8)
# round up from estimated sample size
sampleSize <- ceiling(pwr.result$n)

# create a function that will generate samples and test for
# a difference between groups using a two-sample t-test

get_t_result <- function(sampleSize, effectSize){
  # take sample for the first group from N(0, 1)
  group1 <- rnorm(sampleSize)
  group2 <- rnorm(sampleSize, mean=effectSize)
  ttest.result <- t.test(group1, group2)
```

```
  return(tibble(pvalue=ttest.result$p.value))
}

index_df <- tibble(id=seq(nRuns)) %>%
  group_by(id)

power_sim_results <- index_df %>%
  do(get_t_result(sampleSize, effectSize))

p_reject <-
  power_sim_results %>%
  ungroup() %>%
  summarize(pvalue = mean(pvalue<.05)) %>%
  pull()

p_reject
```

```
## [1] 0.7998
```

This should return a number very close to 0.8.

Chapter 10

Bayesian statistics in R

10.1 A simple example (Section 10.1)

```
bayes_df = data.frame(prior=NA,
                      likelihood=NA,
                      marginal_likelihood=NA,
                      posterior=NA)

bayes_df$prior <- 1/1000000


nTests <- 3
nPositives <- 3
sensitivity <- 0.99
specificity <- 0.99

bayes_df$likelihood <- dbinom(nPositives, nTests, 0.99)

bayes_df$marginal_likelihood <-
  dbinom(
    x = nPositives,
    size = nTests,
    prob = sensitivity
```

```
  ) * bayes_df$prior +
  dbinom(
    x = nPositives,
    size = nTests,
    prob = 1 - specificity
  ) *
  (1 - bayes_df$prior)

bayes_df$posterior <-
  (bayes_df$likelihood * bayes_df$prior) /
  bayes_df$marginal_likelihood
```

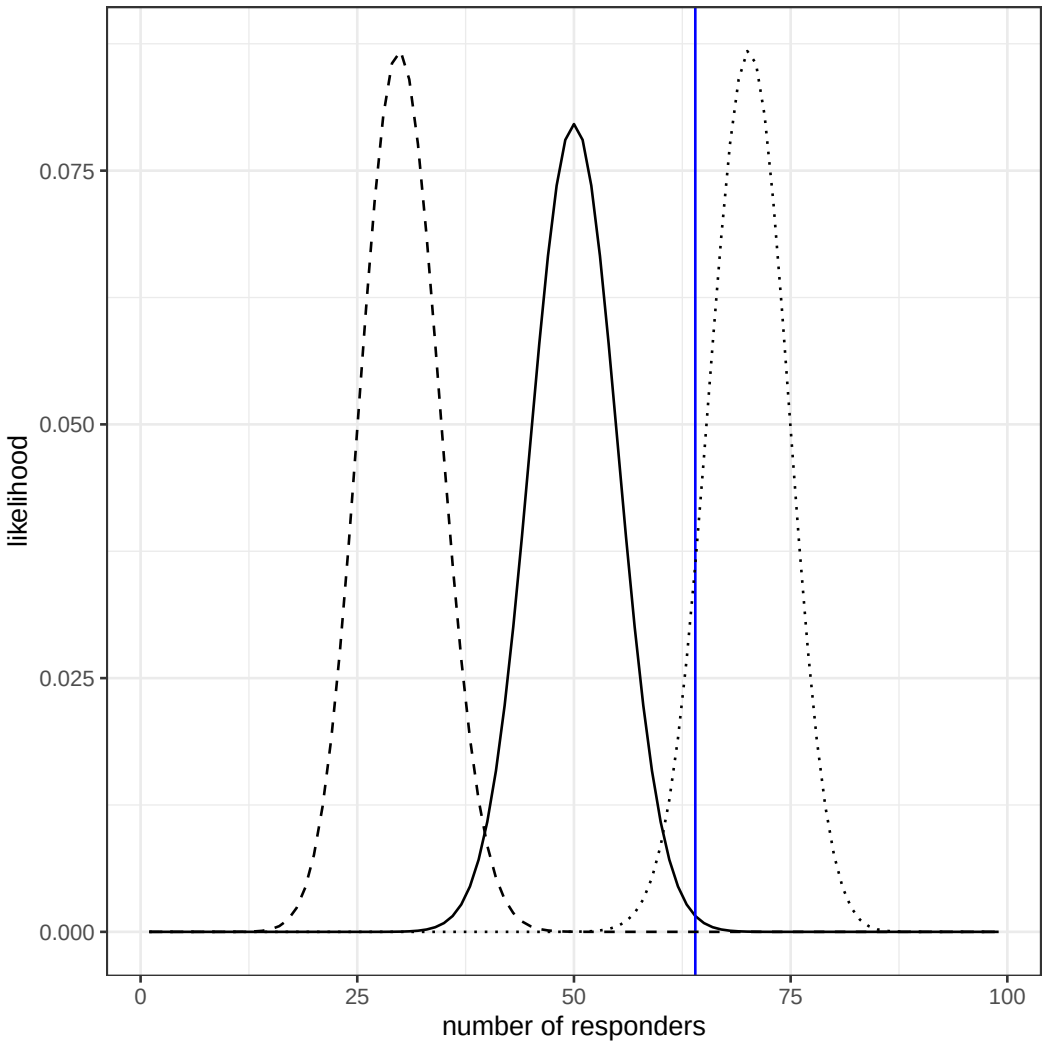
10.2 Estimating posterior distributions (Section 10.2)

```
# create a table with results
nResponders <- 64
nTested <- 100

drugDf <- tibble(
  outcome = c("improved", "not improved"),
  number = c(nResponders, nTested - nResponders)
)
```

Computing likelihood

```
likeDf <-
  tibble(resp = seq(1,99,1)) %>%
  mutate(
    presp=resp/100,
    likelihood5 = dbinom(resp,100,.5),
    likelihood7 = dbinom(resp,100,.7),
    likelihood3 = dbinom(resp,100,.3)
)

ggplot(likeDf,aes(resp,likelihood5)) +
```

```
  geom_line() +
  xlab('number of responders') + ylab('likelihood') +
  geom_vline(xintercept = drugDf$number[1],color='blue') +
  geom_line(aes(resp,likelihood7),linetype='dotted') +
  geom_line(aes(resp,likelihood3),linetype='dashed')
```



Computing marginal likelihood

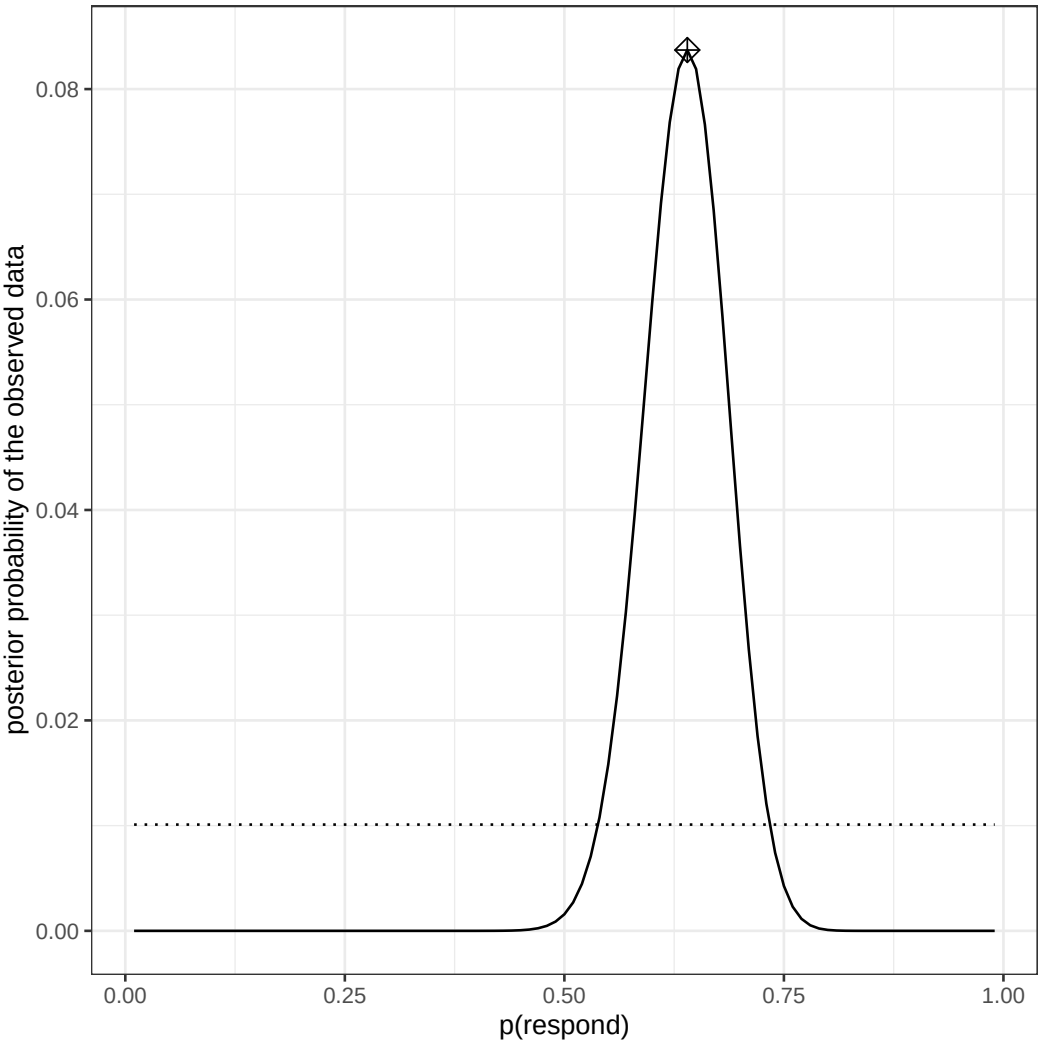
```
# compute marginal likelihood
likeDf <-
```

```
  likeDf %>%
  mutate(uniform_prior = array(1 / n()))

# multiply each likelihood by prior and add them up
marginal_likelihood <-
  sum(
    dbinom(
      x = nResponders, # the number who responded to the drug
      size = 100, # the number tested
      likeDf$presp # the likelihood of each response
    ) * likeDf$uniform_prior
  )
```

Comuting posterior

```
bayesDf <-
  tibble(
    steps = seq(from = 0.01, to = 0.99, by = 0.01)
  ) %>%
  mutate(
    likelihoods = dbinom(
      x = nResponders,
      size = 100,
      prob = steps
    ),
    priors = dunif(steps) / length(steps),
    posteriors = (likelihoods * priors) / marginal_likelihood
  )

# compute MAP estimate
MAP_estimate <-
  bayesDf %>%
  arrange(desc(posteriors)) %>%
  slice(1) %>%
  pull(steps)
```

```
ggplot(bayesDf,aes(steps,posteriors)) +
  geom_line() +
  geom_line(aes(steps,priors),
            color='black',
            linetype='dotted') +
  xlab('p(respond)') +
  ylab('posterior probability of the observed data') +
  annotate(
    "point",
    x = MAP_estimate,
    y = max(bayesDf$posteriors),
    shape=9,
    size = 3
  )
```



10.3 Bayes factors (Section 10.3)

Example showing how BFs and p-values relate

Chapter 11

Modeling categorical relationships in R

So far we have discussed the general concept of statistical modeling and hypothesis testing, and applied them to some simple analyses. In this chapter we will focus on the modeling of *categorical* relationships, by which we mean relationships between variables that are measured qualitatively. These data are usually expressed in terms of counts; that is, for each value of the variable (or combination of values of multiple variables), how many observations take that value? For example, when we count how many people from each major are in our class, we are fitting a categorical model to the data.

11.1 The Pearson Chi-squared test (Section 11.1)

11.2 Two-way tests (Section @ref(two-way-test})

Chapter 12

Modeling continuous relationships in R

12.1 Computing covariance and correlation (Section 12.1)

Let's first look at our toy example of covariance and correlation. For this example, we first start by generating a set of X values.

```
df <-
  tibble(x = c(3, 5, 8, 10, 12))
```

Then we create a related Y variable by adding some random noise to the X variable:

We compute the deviations and multiply them together to get the crossproduct:

And then we compute the covariance and correlation:

```
results_df <- tibble(
  covXY=sum(df$crossproduct) / (nrow(df) - 1),
  corXY= sum(df$crossproduct) /
    ((nrow(df) - 1) * sd(df$x) * sd(df$y)))

kable(results_df)
```

covXY	corXY
17.05	0.894782

12.2 Hate crime example

Now we will look at the hate crime data from the `fivethirtyeight` package. First we need to prepare the data by getting rid of NA values and creating abbreviations for the states. To do the latter, we use the `state.abb` and `state.name` variables that come with R along with the `match()` function that will match the state names in the `hate_crimes` variable to those in the list.

```
hateCrimes <-
  hate_crimes %>%
  mutate(state_abb = state.abb[match(state,state.name)]) %>%
  drop_na(avg_hatecrimes_per_100k_fbi, gini_index)

# manually fix the DC abbreviation
hateCrimes$state_abb[hateCrimes$state=="District of Columbia"] <- 'DC'
```

```
##
##  Pearson's product-moment correlation
##
## data:  hateCrimes$avg_hatecrimes_per_100k_fbi and hateCrimes$gini_index
## t = 3.2182, df = 48, p-value = 0.001157
## alternative hypothesis: true correlation is greater than 0
## 95 percent confidence interval:
##  0.2063067 1.0000000
## sample estimates:
##       cor
## 0.4212719
```

Remember that we can also compute the p-value using randomization. To to this, we shuffle the order of one of the variables, so that we break the link between the X and Y variables — effectively making the null hypothesis (that the correlation is less than or equal to zero) true. Here we will first create a function that takes in two variables, shuffles the order of one of them (without replacement) and then returns the correlation between that shuffled variable

and the original copy of the second variable.

Now we take the distribution of observed correlations after shuffling and compare them to our observed correlation, in order to obtain the empirical probability of our observed data under the null hypothesis.

```
mean(shuffleDist$cor >corr_results$estimate )
```

```
## [1] 0.0066
```

This value is fairly close (though a bit larger) to the one obtained using `cor.test()`.

12.3 Robust correlations (Section 12.3)

In the previous chapter we also saw that the hate crime data contained one substantial outlier, which appeared to drive the significant correlation. To compute the Spearman correlation, we first need to convert the data into their ranks, which we can do using the `order()` function:

```
hateCrimes <- hateCrimes %>%
  mutate(hatecrimes_rank = order(avg_hatecrimes_per_100k_fbi),
         gini_rank = order(gini_index))
```

We can then compute the Spearman correlation by applying the Pearson correlation to the rank variables"

```
cor(hateCrimes$hatecrimes_rank,
  hateCrimes$gini_rank)
```

```
## [1] 0.05690276
```

We see that this is much smaller than the value obtained using the Pearson correlation on the original data. We can assess its statistical signficance using randomization:

```
## [1] 0.0014
```

Here we see that the p-value is substantially larger and far from significance.

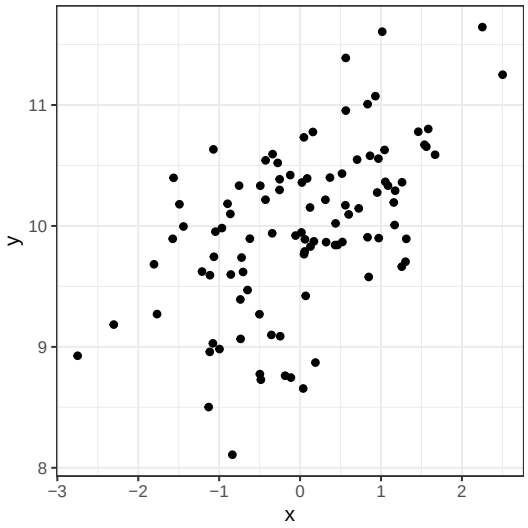
Chapter 13

The General Linear Model in R

13.1 Linear regression (Section 13.1)

To perform linear regression in R, we use the `lm()` function. Let's generate some data and use this function to compute the linear regression solution.

```
npoints <- 100
intercept = 10
# slope of X/Y relationship
slope=0.5
# this lets us control the strength of the relationship
# by varying the amount of noise added to the y variable
noise_sd = 0.6

regression_data <- tibble(x = rnorm(npoints)) %>%
  mutate(y = x*slope + rnorm(npoints)*noise_sd + intercept)

ggplot(regression_data,aes(x,y)) +
  geom_point()
```



We can then apply `lm()` to these data:

```
lm_result <- lm(y ~ x, data=regression_data)
summary(lm_result)
```

```
##
## Call:
## lm(formula = y ~ x, data = regression_data)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.55632 -0.30424 -0.00587  0.38039  1.25219
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  9.97610    0.05796 172.122  < 2e-16 ***
## x            0.37246    0.05862   6.353 6.65e-09 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.5795 on 98 degrees of freedom
## Multiple R-squared:  0.2917, Adjusted R-squared:  0.2845
## F-statistic: 40.36 on 1 and 98 DF,  p-value: 6.646e-09
```

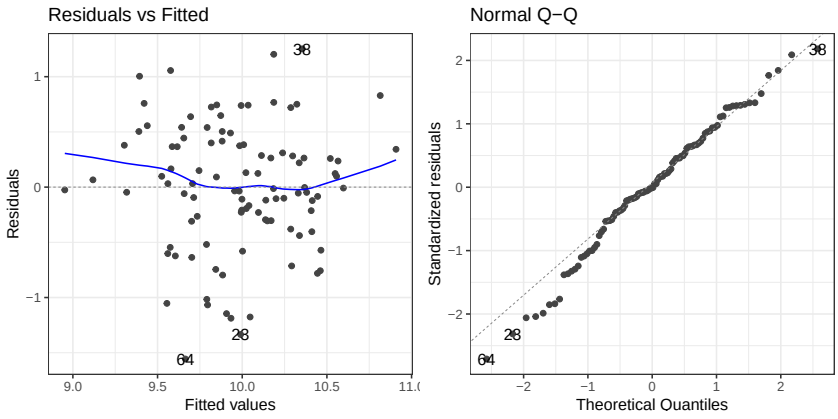
We should see three things in the `lm()` results:

- The estimate of the Intercept in the model should be very close to the intercept that we specified
- The estimate for the x parameter should be very close to the slope that we specified
- The residual standard error should be roughly similar to the noise standard deviation that we specified

13.2 Model criticism and diagnostics (Section 13.2)

Once we have fitted the model, we want to look at some diagnostics to determine whether the model is actually fitting properly. We can do this using the `autoplot()` function from the `ggfortify` package.

```
autoplot(lm_result,which=1:2)
```



The left panel in this plot shows the relationship between the predicted (or "fitted") values and the residuals. We would like to make sure that there is no clear relationship between these two (as we will see below). The right panel shows a Q-Q plot, which helps us assess whether the residuals from the model are normally distributed. In this case, they look reasonably normal, as the points don't differ too much from the unit line.

13.3 Examples of problematic model fit

Let's say that there was another variable at play in this dataset, which we were not aware of. This variable causes some of the cases to have much larger values than others, in a way that is unrelated to the X variable. We play a trick here using the `seq()` function to create a sequence from zero to one, and then threshold those 0.5 (in order to obtain half of the values as zero and the other half as one) and then multiply by the desired effect size:

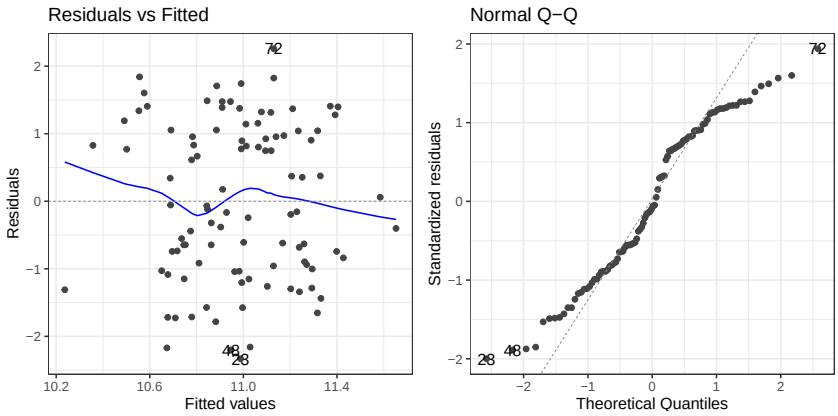
```
effsize=2
regression_data <- regression_data %>%
  mutate(y2=y + effsize*(seq(1/npoints,1,1/npoints)>0.5))

lm_result2 <- lm(y2 ~ x, data=regression_data)
summary(lm_result2)
```

```
##
## Call:
## lm(formula = y2 ~ x, data = regression_data)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -2.33243 -0.96887 -0.09385  1.04213  2.25913
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  10.9778     0.1172  93.646   <2e-16 ***
## x             0.2696     0.1186   2.273   0.0252 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.172 on 98 degrees of freedom
## Multiple R-squared:  0.0501, Adjusted R-squared:  0.04041
## F-statistic: 5.169 on 1 and 98 DF,  p-value: 0.02518
```

One thing you should notice is that the model now fits overall much worse; the R-squared is about half of what it was in the previous model, which reflects the fact that more variability was added to the data, but it wasn't accounted for in the model. Let's see if our diagnostic reports can give us

any insight:

```
autoplot(lm_result2,which=1:2)
```



The residual versus fitted graph doesn't give us much insight, but we see from the Q-Q plot that the residuals are diverging quite a bit from the unit line.

Let's look at another potential problem, in which the y variable is nonlinearly related to the X variable. We can create these data by squaring the X variable when we generate the Y variable:

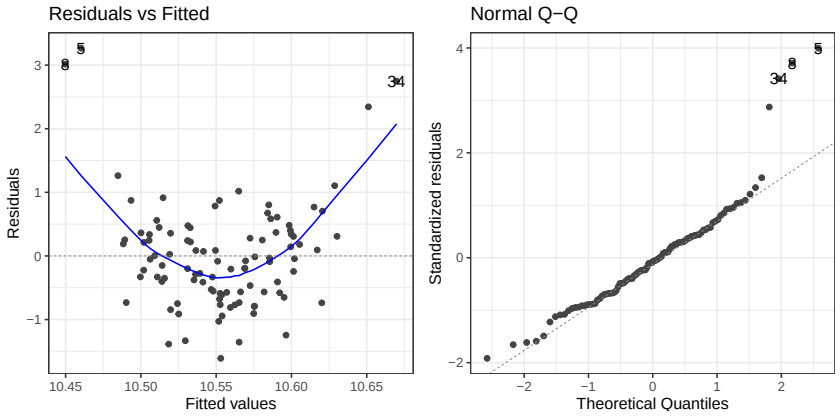
```
effsize=2
regression_data <- regression_data %>%
  mutate(y3 = (x**2)*slope + rnorm(npoints)*noise_sd + intercept)

lm_result3 <- lm(y3 ~ x, data=regression_data)
summary(lm_result3)
```

```
##
## Call:
## lm(formula = y3 ~ x, data = regression_data)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -1.6100 -0.5679 -0.0649  0.3587  3.2662
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
```

```
## (Intercept) 10.55467    0.08439 125.073   <2e-16 ***
## x           -0.04189    0.08536  -0.491    0.625
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.8438 on 98 degrees of freedom
## Multiple R-squared:  0.002452,   Adjusted R-squared:  -0.007727
## F-statistic: 0.2409 on 1 and 98 DF,  p-value: 0.6247
```

Now we see that there is no significant linear relationship between X^2 and Y/ But if we look at the residuals the problem with the model becomes clear:

```
autoplot(lm_result3,which=1:2)
```



In this case we can see the clearly nonlinear relationship between the predicted and residual values, as well as the clear lack of normality in the residuals.

As we noted in the previous chapter, the "linear" in the general linear model doesn't refer to the shape of the response, but instead refers to the fact that model is linear in its parameters — that is, the predictors in the model only get multiplied the parameters (e.g., rather than being raised to a power of the parameter). Here is how we would build a model that could account for the nonlinear relationship:

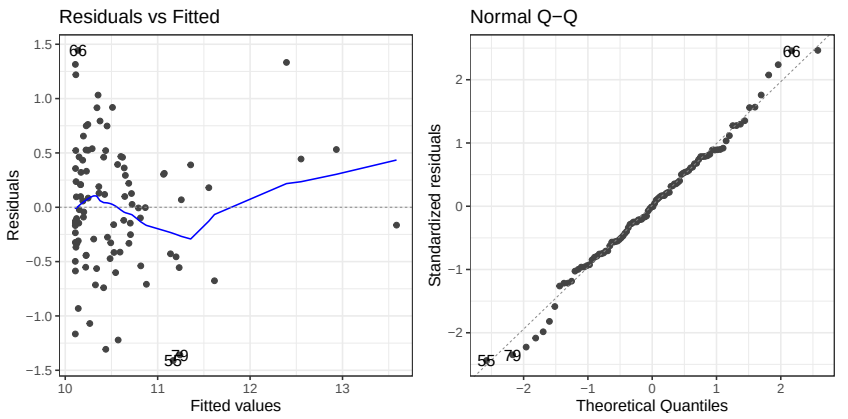
```
# create x^2 variable
regression_data <- regression_data %>%
  mutate(x_squared = x**2)

lm_result4 <- lm(y3 ~ x + x_squared, data=regression_data)
```

```
summary(lm_result4)
```

```
##
## Call:
## lm(formula = y3 ~ x + x_squared, data = regression_data)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.41009 -0.37915 -0.00483  0.39079  1.44368
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 10.10875    0.07387 136.836   <2e-16 ***
## x           -0.01178    0.05998  -0.196    0.845
## x_squared    0.45569    0.04513  10.098   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.5922 on 97 degrees of freedom
## Multiple R-squared:  0.5137, Adjusted R-squared:  0.5037
## F-statistic: 51.23 on 2 and 97 DF,  p-value: 6.536e-16
```

Now we see that the effect of X^2 is significant, and if we look at the residual plot we should see that things look much better:

```
autoplot(lm_result4,which=1:2)
```



Not perfect, but much better than before!

13.4 Extending regression to binary outcomes.

Let's say that we have a blood test (which is often referred to as a *biomarker*) and we want to know whether it predicts who is going to have a heart attack within the next year. We will generate a synthetic dataset for a population that is at very high risk for a heart attack in the next year.

```
# sample size
npatients=1000

# probability of heart attack
p_heartattack = 0.5

# true relation to biomarker
true_effect <- 0.6

# assume biomarker is normally distributed
disease_df <- tibble(biomarker=rnorm(npatients))

# generate another variable that reflects risk for
# heart attack, which is related to the biomarker
disease_df <- disease_df %>%
  mutate(risk = biomarker*true_effect + rnorm(npatients))

# create another variable that shows who has a
# heart attack, based on the risk variable
disease_df <- disease_df %>%
  mutate(
    heartattack = risk > quantile(disease_df$risk,
                                  1-p_heartattack))

glimpse(disease_df)
```

```
## Rows: 1,000
## Columns: 3
## $ biomarker   <dbl> 1.1512317, 0.6753651, 1.2141696, -0.72~
## $ risk        <dbl> 1.05356169, -0.52889832, 0.67528209, -~
```

```
## $ heartattack <lgl> TRUE, FALSE, TRUE, FALSE, FALSE, TRUE,~
```

Now we would like to build a model that allows us to predict who will have a heart attack from these data. However, you may have noticed that the heartattack variable is a binary variable; because linear regression assumes that the residuals from the model will be normally distributed, and the binary nature of the data will violate this, we instead need to use a different kind of model, known as a *logistic regression* model, which is built to deal with binary outcomes. We can fit this model using the `glm()` function:

```
glm_result <- glm(heartattack ~ biomarker, data=disease_df,
                  family=binomial())
summary(glm_result)
```

```
##
## Call:
## glm(formula = heartattack ~ biomarker, family = binomial(), data = disease_df)
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept) -0.004117   0.069479  -0.059    0.953
## biomarker    0.996366   0.083418  11.944   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 1386.3  on 999  degrees of freedom
## Residual deviance: 1201.4  on 998  degrees of freedom
## AIC: 1205.4
##
## Number of Fisher Scoring iterations: 3
```

This looks very similar to the output from the `lm()` function, and it shows us that there is a significant relationship between the biomarker and heart attacks. The model provides us with a predicted probability that each individual will have a heart attack; if this is greater than 0.5, then that means that the model predicts that the individual is more likely than not to have a heart attack.

We can start by simply comparing those predictions to the actual outcomes.

```
# add predictions to data frame
disease_df <- disease_df %>%
  mutate(prediction = glm_result$fitted.values>0.5,
         heartattack = heartattack)

# create table comparing predicted to actual outcomes
CrossTable(disease_df$prediction,
           disease_df$heartattack,
           prop.t=FALSE,
           prop.r=FALSE,
           prop.chisq=FALSE)
```

```
##
##
##    Cell Contents
## |-------------------------|
## |                       N |
## |           N / Col Total |
## |-------------------------|
##
##
## Total Observations in Table:  1000
##
##
##                      | disease_df$heartattack
## disease_df$prediction |     FALSE |      TRUE | Row Total |
## ----------------------|-----------|-----------|-----------|
##                 FALSE |       332 |       157 |       489 |
##                       |     0.664 |     0.314 |           |
## ----------------------|-----------|-----------|-----------|
##                  TRUE |       168 |       343 |       511 |
##                       |     0.336 |     0.686 |           |
## ----------------------|-----------|-----------|-----------|
##          Column Total |       500 |       500 |      1000 |
##                       |     0.500 |     0.500 |           |
## ----------------------|-----------|-----------|-----------|
```

```
##
##
```

This shows us that of the 500 people who had heart attacks, the model corrected predicted a heart attack for 343 of them. It also predicted heart attacks for 168 people who didn't have them, and it failed to predict a heart attack for 157 people who had them. This highlights the distinction that we mentioned before between statistical and practical significance; even though the biomarker shows a highly significant relationship to heart attacks, it's ability to predict them is still relatively poor. As we will see below, it gets even worse when we try to generalize this to a new group of people.

13.5 Cross-validation (Section 13.5)

Cross-validation is a powerful technique that allows us to estimate how well our results will generalize to a new dataset. Here we will build our own crossvalidation code to see how it works, continuing the logistic regression example from the previous section.

In cross-validation, we want to split the data into several subsets and then iteratively train the model while leaving out each subset (which we usually call *folds*) and then test the model on that held-out fold Let's write our own code to do this splitting; one relatively easy way to this is to create a vector that contains the fold numbers, and then randomly shuffle it to create the fold assigments for each data point.

```
nfolds <- 4 # number of folds

# we use the kronecker() function to repeat the folds
fold <-  kronecker(seq(nfolds),rep(1,npatients/nfolds))
# randomly shuffle using the sample() function
fold <- sample(fold)

# add variable to store CV predictions
disease_df <- disease_df %>%
  mutate(CVpred=NA)

# now loop through folds and separate training and test data
```

```
for (f in seq(nfolds)){
  # get training and test data
  train_df <- disease_df[fold!=f,]
  test_df <- disease_df[fold==f,]
  # fit model to training data
  glm_result_cv <- glm(heartattack ~ biomarker, data=train_df,
                       family=binomial())
  # get probability of heart attack on test data
  pred <- predict(glm_result_cv,newdata = test_df)
  # convert to prediction and put into data frame
  disease_df$CVpred[fold==f] = (pred>0.5)

}
```

Now let's look at the performance of the model:

```
# create table comparing predicted to actual outcomes
CrossTable(disease_df$CVpred,
           disease_df$heartattack,
           prop.t=FALSE,
           prop.r=FALSE,
           prop.chisq=FALSE)
```

```
##
##
##    Cell Contents
## |-------------------------|
## |                       N |
## |           N / Col Total |
## |-------------------------|
##
##
## Total Observations in Table:  1000
##
##
##                    | disease_df$heartattack
## disease_df$CVpred |     FALSE |      TRUE | Row Total |
## ------------------|-----------|-----------|-----------|
```

```
##              FALSE |       416 |       269 |       685 |
##                    |     0.832 |     0.538 |           |
## -------------------|-----------|-----------|-----------|
##               TRUE |        84 |       231 |       315 |
##                    |     0.168 |     0.462 |           |
## -------------------|-----------|-----------|-----------|
##      Column Total |       500 |       500 |      1000 |
##                    |     0.500 |     0.500 |           |
## -------------------|-----------|-----------|-----------|
##
##
```

Now we see that the model only accurately predicts less than half of the heart attacks that occurred when it is predicting to a new sample. This tells us that this is the level of prediction that we could expect if were to apply the model to a new sample of patients from the same population.

Chapter 14

Comparing means in R

14.1 Testing the value of a single mean (Section 14.1)

In this example, we will show multiple ways to test a hypothesis about the value of a single mean. As an example, let's test whether the mean systolic blood pressure (BP) in the NHANES dataset (averaged over the three measurements that were taken for each person) is greater than 120 mm, which is the standard value for normal systolic BP.

First let's perform a power analysis to see how large our sample would need to be in order to detect a small difference (Cohen's d = .2).

```
pwr.result <- pwr.t.test(d=0.2, power=0.8,
           type='one.sample',
           alternative='greater')
pwr.result
```

```
##
##      One-sample t test power calculation
##
##               n = 155.9256
##               d = 0.2
##       sig.level = 0.05
##           power = 0.8
```

```
##     alternative = greater
```

Based on this, we take a sample of 156 individuals from the dataset.

```
NHANES_BP_sample <- NHANES_adult %>%
  drop_na(BPSysAve) %>%
  dplyr::select(BPSysAve) %>%
  sample_n(pwr.result$n)

print('Mean BP:')
```

```
## [1] "Mean BP:"
```

```
meanBP <- NHANES_BP_sample %>%
  summarize(meanBP=mean(BPSysAve)) %>%
  pull()
meanBP
```

```
## [1] 123.1097
```

First let's perform a sign test to see whether the observed mean of 123.11 is significantly different from zero. To do this, we count the number of values that are greater than the hypothesized mean, and then use a binomial test to ask how surprising that number is if the true proportion is 0.5 (as it would be if the distribution were centered at the hypothesized mean).

```
NHANES_BP_sample <- NHANES_BP_sample %>%
  mutate(BPover120=BPSysAve>120)

nOver120 <- NHANES_BP_sample %>%
  summarize(nOver120=sum(BPover120)) %>%
  pull()

binom.test(nOver120, nrow(NHANES_BP_sample), alternative='greater')
```

```
##
##  Exact binomial test
##
## data:  nOver120 and nrow(NHANES_BP_sample)
## number of successes = 84, number of trials = 155, p-value = 0.1676
## alternative hypothesis: true probability of success is greater than 0.5
```

```
## 95 percent confidence interval:
##  0.4727095 1.0000000
## sample estimates:
## probability of success
##              0.5419355
```

This shows no significant difference. Next let's perform a one-sample t-test:

```
t.test(NHANES_BP_sample$BPSysAve, mu=120, alternative='greater')
```

```
##
##  One Sample t-test
##
## data:  NHANES_BP_sample$BPSysAve
## t = 2.2045, df = 154, p-value = 0.01449
## alternative hypothesis: true mean is greater than 120
## 95 percent confidence interval:
##  120.7754      Inf
## sample estimates:
## mean of x
##  123.1097
```

Here we see that the difference is not statistically signficant. Finally, we can perform a randomization test to test the hypothesis. Under the null hypothesis we would expect roughly half of the differences from the expected mean to be positive and half to be negative (assuming the distribution is centered around the mean), so we can cause the null hypothesis to be true on average by randomly flipping the signs of the differences.

```
nruns = 5000

# create a function to compute the
# t on the shuffled values
shuffleOneSample <- function(x,mu) {
  # randomly flip signs
  flip <- runif(length(x))>0.5
  diff <- x - mu
  diff[flip]=-1*diff[flip]
  # compute and return correlation
  # with shuffled variable
```

```
  return(tibble(meanDiff=mean(diff)))
}

index_df <- tibble(id=seq(nruns)) %>%
  group_by(id)

shuffle_results <- index_df %>%
  do(shuffleOneSample(NHANES_BP_sample$BPSysAve,120))

observed_diff <- mean(NHANES_BP_sample$BPSysAve-120)
p_shuffle <- mean(shuffle_results$meanDiff>observed_diff)
p_shuffle
```

```
## [1] 0.014
```

This gives us a very similar p value to the one observed with the standard t-test.

We might also want to quantify the degree of evidence in favor of the null hypothesis, which we can do using the Bayes Factor:

```
ttestBF(NHANES_BP_sample$BPSysAve,
        mu=120,
        nullInterval = c(-Inf, 0))
```

```
## Bayes factor analysis
## --------------
## [1] Alt., r=0.707 -Inf<d<0    : 0.02856856 ±0.04%
## [2] Alt., r=0.707 !(-Inf<d<0) : 1.848811   ±0%
##
## Against denominator:
##   Null, mu = 120
## ---
## Bayes factor type: BFoneSample, JZS
```

This tells us that our result doesn't provide particularly strong evidence for either the null or alternative hypothesis; that is, it's inconclusive.

14.2 Comparing two means (Section 14.2)

To compare two means from independent samples, we can use the two-sample t-test. Let's say that we want to compare blood pressure of smokers and non-smokers; we don't have an expectation for the direction, so we will use a two-sided test. First let's perform a power analysis, again for a small effect:

```
power_results_2sample <- pwr.t.test(d=0.2, power=0.8,
                                    type='two.sample'
                                    )
power_results_2sample
```

```
##
##      Two-sample t test power calculation
##
##               n = 393.4057
##               d = 0.2
##       sig.level = 0.05
##           power = 0.8
##     alternative = two.sided
##
## NOTE: n is number in *each* group
```

This tells us that we need 394 subjects in each group, so let's sample 394 smokers and 394 nonsmokers from the NHANES dataset, and then put them into a single data frame with a variable denoting their smoking status.

```
nonsmoker_df <- NHANES_adult %>%
  dplyr::filter(SmokeNow=="Yes") %>%
  drop_na(BPSysAve) %>%
  dplyr::select(BPSysAve,SmokeNow) %>%
  sample_n(power_results_2sample$n)

smoker_df <- NHANES_adult %>%
  dplyr::filter(SmokeNow=="No") %>%
  drop_na(BPSysAve) %>%
  dplyr::select(BPSysAve,SmokeNow) %>%
  sample_n(power_results_2sample$n)
```

```
sample_df <- smoker_df %>%
  bind_rows(nonsmoker_df)
```

Let's test our hypothesis using a standard two-sample t-test. We can use the formula notation to specify the analysis, just like we would for `lm()`.

```
t.test(BPSysAve ~ SmokeNow, data=sample_df)
```

```
##
##  Welch Two Sample t-test
##
## data:  BPSysAve by SmokeNow
## t = 4.2105, df = 774.94, p-value = 2.848e-05
## alternative hypothesis: true difference in means between group No and group
## 95 percent confidence interval:
##  2.850860 7.831074
## sample estimates:
##  mean in group No mean in group Yes
##          125.1603          119.8193
```

This shows us that there is a significant difference, though the direction is surprising: Smokers have *lower* blood pressure!

Let's look at the Bayes factor to quantify the evidence:

```
sample_df <- sample_df %>%
  mutate(SmokeNowInt=as.integer(SmokeNow))
ttestBF(formula=BPSysAve ~ SmokeNowInt,
        data=sample_df)
```

```
## Bayes factor analysis
## --------------
## [1] Alt., r=0.707 : 440.269 ±0%
##
## Against denominator:
##   Null, mu1-mu2 = 0
## ---
## Bayes factor type: BFindepSample, JZS
```

This shows that there is very strong evidence against the null hypothesis of

no difference.

14.3 The t-test as a linear model (Section 14.3)

We can also use `lm()` to implement these t-tests.

The one-sample t-test is basically a test for whether the intercept is different from zero, so we use a model with only an intercept and apply this to the data after subtracting the null hypothesis mean (so that the expectation under the null hypothesis is an intercept of zero):

```
NHANES_BP_sample <- NHANES_BP_sample %>%
  mutate(BPSysAveDiff = BPSysAve-120)
lm_result <- lm(BPSysAveDiff ~ 1, data=NHANES_BP_sample)
summary(lm_result)
```

```
##
## Call:
## lm(formula = BPSysAveDiff ~ 1, data = NHANES_BP_sample)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -36.11 -13.11  -1.11   9.39  67.89
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)    3.110      1.411   2.205    0.029 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 17.56 on 154 degrees of freedom
```

You will notice that this p-value is twice as big as the one obtained from the one-sample t-test above; this is because that was a one-tailed test, while `lm()` is performing a two-tailed test.

We can also run the two-sample t-test using `lm()`:

```
lm_ttest_result <- lm(BPSysAve ~ SmokeNow, data=sample_df)
summary(lm_ttest_result)
```

```
##
## Call:
## lm(formula = BPSysAve ~ SmokeNow, data = sample_df)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -45.16 -11.16  -2.16   8.84 101.18
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  125.160      0.897  139.54  < 2e-16 ***
## SmokeNowYes   -5.341      1.268   -4.21 2.84e-05 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 17.78 on 784 degrees of freedom
## Multiple R-squared:  0.02211,    Adjusted R-squared:  0.02086
## F-statistic: 17.73 on 1 and 784 DF,  p-value: 2.844e-05
```

This gives the same p-value for the SmokeNowYes variable as it did for the two-sample t-test above.

14.4 Comparing paired observations (Section 14.4)

Let's look at how to perform a paired t-test in R. In this case, let's generate some data for a set of individuals on two tests, where each indivdual varies in their overall ability, but there is also a practice effect such that performance on the second test is generally better than the first.

First, let's see how big of a sample we will require to find a medium (d=0.5) sized effect. Let's say that we want to be extra sure in our results, so we will find the sample size that gives us 95% power to find an effect if it's there:

```
paired_power <- pwr.t.test(d=0.5, power=0.95, type='paired', alternative='greater')
paired_power
```

```
##
##      Paired t test power calculation
##
##               n = 44.67998
##               d = 0.5
##       sig.level = 0.05
##           power = 0.95
##     alternative = greater
##
## NOTE: n is number of *pairs*
```

Now let's generate a dataset with the required number of subjects:

```
subject_id <- seq(paired_power$n)
# we code the tests as 0/1 so that we can simply
# multiply this by the effect to generate the data
test_id <- c(0,1)
repeat_effect <- 5
noise_sd <- 5

subject_means <- rnorm(paired_power$n, mean=100, sd=15)
paired_data <- crossing(subject_id,test_id) %>%
  mutate(subMean=subject_means[subject_id],
         score=subject_means +
           test_id*repeat_effect +
           rnorm(paired_power$n, mean=noise_sd))
```

Let's perform a paired t-test on these data. To do that, we need to separate the first and second test data into separate variables, which we can do by converting our *long* data frame into a *wide* data frame.

```
paired_data_wide <- paired_data %>%
  spread(test_id, score) %>%
  rename(test1=`0`,
         test2=`1`)
```

```
glimpse(paired_data_wide)
```

```
## Rows: 44
## Columns: 4
## $ subject_id <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, ~
## $ subMean    <dbl> 116.45972, 94.94520, 103.11749, 91.0816~
## $ test1      <dbl> 120.53690, 107.58688, 101.69392, 94.071~
## $ test2      <dbl> 104.44096, 101.07566, 101.86726, 106.67~
```

Now we can pass those new variables into the `t.test()` function:

```
paired_ttest_result <- t.test(paired_data_wide$test1,
                              paired_data_wide$test2,
                              type='paired')
paired_ttest_result
```

```
##
##  Welch Two Sample t-test
##
## data:  paired_data_wide$test1 and paired_data_wide$test2
## t = -1.2799, df = 73.406, p-value = 0.2046
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -10.536522   2.295414
## sample estimates:
## mean of x mean of y
##  107.5863  111.7068
```

This analysis is a bit trickier to perform using the linear model, because we need to estimate a separate intercept for each subject in order to account for the overall differences between subjects. We can't do this using `lm()` but we can do it using a function called `lmer()` from the `lme4` package. To do this, we need to add `(1|subject_id)` to the formula, which tells `lmer()` to add a separate intercept ("1") for each value of `subject_id`.

```
paired_test_lmer <- lmer(score ~ test_id + (1|subject_id),
                         data=paired_data)
summary(paired_test_lmer)
```

```
## Linear mixed model fit by REML. t-tests use Satterthwaite's method [
```

```
## lmerModLmerTest]
## Formula: score ~ test_id + (1 | subject_id)
##    Data: paired_data
##
## REML criterion at convergence: 718.6
##
## Scaled residuals:
##      Min       1Q   Median       3Q      Max
## -2.54238 -0.62142 -0.09286  0.73490  2.97925
##
## Random effects:
##  Groups     Name        Variance Std.Dev.
##  subject_id (Intercept)   0       0.0
##  Residual               228      15.1
## Number of obs: 88, groups:  subject_id, 44
##
## Fixed effects:
##             Estimate Std. Error      df t value Pr(>|t|)
## (Intercept)  107.586      2.277  86.000   47.26   <2e-16 ***
## test_id        4.121      3.220  86.000    1.28    0.204
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Correlation of Fixed Effects:
##         (Intr)
## test_id -0.707
## optimizer (nloptwrap) convergence code: 0 (OK)
## boundary (singular) fit: see help('isSingular')
```

This gives a similar answer to the standard paired t-test. The advantage is that it's more flexible, allowing us to perform *repeated measures* analyses, as we will see below.

14.5 Analysis of variance (Section 14.5)

Often we want to compare several different means, to determine whether any of them are different from the others. In this case, let's look at the data from

NHANES to determine whether Marital Status is related to sleep quality. First we clean up the data:

```
NHANES_sleep_marriage <-
  NHANES_adult %>%
  dplyr::select(SleepHrsNight, MaritalStatus, Age) %>%
  drop_na()
```

In this case we are going to treat the full NHANES dataset as our sample, with the goal of generalizing to the entire US population (from which the NHANES dataset is mean to be a representative sample). First let's look at the distribution of the different values of the `MaritalStatus` variable:

```
NHANES_sleep_marriage %>%
  group_by(MaritalStatus) %>%
  summarize(n=n()) %>%
  kable()
```

MaritalStatus	n
Divorced	437
LivePartner	370
Married	2434
NeverMarried	889
Separated	134
Widowed	329

There are reasonable numbers of most of these categories, but let's remove the `Separated` category since it has relatively few members:

```
NHANES_sleep_marriage <-
  NHANES_sleep_marriage %>%
  dplyr::filter(MaritalStatus!="Separated")
```

Now let's use `lm()` to perform an analysis of variance. Since we also suspect that Age is related to the amount of sleep, we will also include Age in the model.

```
lm_sleep_marriage <- lm(SleepHrsNight ~ MaritalStatus + Age,
                        data=NHANES_sleep_marriage)
summary(lm_sleep_marriage)
```

```
##
## Call:
## lm(formula = SleepHrsNight ~ MaritalStatus + Age, data = NHANES_sleep_marriage)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -5.0160 -0.8797  0.1065  1.0821  5.2821
##
## Coefficients:
##                            Estimate Std. Error t value Pr(>|t|)
## (Intercept)                6.517579   0.098024  66.490  < 2e-16 ***
## MaritalStatusLivePartner   0.143733   0.098692   1.456 0.145360
## MaritalStatusMarried       0.234936   0.070937   3.312 0.000934 ***
## MaritalStatusNeverMarried 0.251721   0.084036   2.995 0.002756 **
## MaritalStatusWidowed       0.263035   0.103270   2.547 0.010896 *
## Age                        0.003180   0.001415   2.248 0.024643 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.365 on 4453 degrees of freedom
## Multiple R-squared:  0.004583,   Adjusted R-squared:  0.003465
## F-statistic:   4.1 on 5 and 4453 DF,  p-value: 0.001024
```

This tells us that there is a highly significant effect of marital status (based on the F test), though it accounts for a very small amount of variance (less than 1%).

It's also useful to look in more detail at which groups differ from which others, which we can do by examining the *estimated marginal means* for each group using the `emmeans()` function.

```
# compute the differences between each of the means
leastsquare <- emmeans(lm_sleep_marriage,
                       pairwise ~ MaritalStatus,
                       adjust="tukey")

# display the results by grouping using letters

cld(leastsquare$emmeans,
```

```
    alpha=.05,
    Letters=letters)
```

```
##  MaritalStatus emmean     SE   df lower.CL upper.CL .group
##  Divorced        6.67 0.0656 4453     6.54     6.80  a
##  LivePartner     6.81 0.0725 4453     6.67     6.95  ab
##  Married         6.90 0.0280 4453     6.85     6.96   b
##  NeverMarried    6.92 0.0501 4453     6.82     7.02   b
##  Widowed         6.93 0.0823 4453     6.77     7.09  ab
##
## Confidence level used: 0.95
## P value adjustment: tukey method for comparing a family of 5 estimates
## significance level used: alpha = 0.05
## NOTE: If two or more means share the same grouping symbol,
##       then we cannot show them to be different.
##       But we also did not show them to be the same.
```

The letters in the `group` column tell us which individual conditions differ from which others; any pair of conditions that don't share a group identifier (in this case, the letters `a` and `b`) are significantly different from one another. In this case, we see that Divorced people sleep less than Married or Widowed individuals; no other pairs differ significantly.

14.5.1 Repeated measures analysis of variance

The standard analysis of variance assumes that the observations are independent, which should be true for different people in the NHANES dataset, but may not be true if the data are based on repeated measures of the same individual. For example, the NHANES dataset involves three measurements of blood pressure for each individual. If we want to test whether there are any differences between those, then we would need to use a *repeated measures* analysis of variance. We can do this using `lmer()` as we did above. First, we need to create a "long" version of the dataset.

```
NHANES_bp_all <- NHANES_adult %>%
  drop_na(BPSys1,BPSys2,BPSys3) %>%
  dplyr::select(BPSys1,BPSys2,BPSys3, ID) %>%
  gather(test, BPsys, -ID)
```

Then we fit a model that includes a separate intercept for each individual.

```
repeated_lmer <-lmer(BPsys ~ test + (1|ID), data=NHANES_bp_all)
summary(repeated_lmer)
```

```
## Linear mixed model fit by REML. t-tests use Satterthwaite's method [
## lmerModLmerTest]
## Formula: BPsys ~ test + (1 | ID)
##    Data: NHANES_bp_all
##
## REML criterion at convergence: 89301
##
## Scaled residuals:
##     Min      1Q  Median      3Q     Max
## -4.5475 -0.5125 -0.0053  0.4948  4.1339
##
## Random effects:
##  Groups   Name        Variance Std.Dev.
##  ID       (Intercept) 280.9    16.761
##  Residual              16.8     4.099
## Number of obs: 12810, groups:  ID, 4270
##
## Fixed effects:
##              Estimate Std. Error        df t value Pr(>|t|)
## (Intercept)  122.0037     0.2641 4605.7049  462.04   <2e-16 ***
## testBPSys2    -0.9283     0.0887 8538.0000  -10.47   <2e-16 ***
## testBPSys3    -1.6216     0.0887 8538.0000  -18.28   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Correlation of Fixed Effects:
##             (Intr) tsBPS2
## testBPSys2 -0.168
## testBPSys3 -0.168  0.500
```

This shows us that the second and third tests are significant different from the first test (which was automatically assigned as the baseline by `lmer()`). We might also want to know whether there is an overall effect of test. We can determine this by comparing the fit of our model to the fit of a model that

does not include the test variable, which we will fit here. We then compare the models using the `anova()` function, which performs an F test to compare the two models.

```
repeated_lmer_baseline <-lmer(BPsys ~ (1|ID), data=NHANES_bp_all)
anova(repeated_lmer,repeated_lmer_baseline)
```

```
## Data: NHANES_bp_all
## Models:
## repeated_lmer_baseline: BPsys ~ (1 | ID)
## repeated_lmer: BPsys ~ test + (1 | ID)
##                         npar   AIC   BIC logLik deviance  Chisq Df Pr(>Chisq
## repeated_lmer_baseline    3 89630 89652 -44812    89624
## repeated_lmer             5 89304 89341 -44647    89294 330.15  2  < 2.2e-1
##
## repeated_lmer_baseline
## repeated_lmer          ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

This shows that blood pressure differs significantly across the three tests.

Chapter 15

Practical statistical modeling in R

Chapter 16

References