R for Researchers: An Introduction

Tyson S. Barrett, PhD

2019-05-13

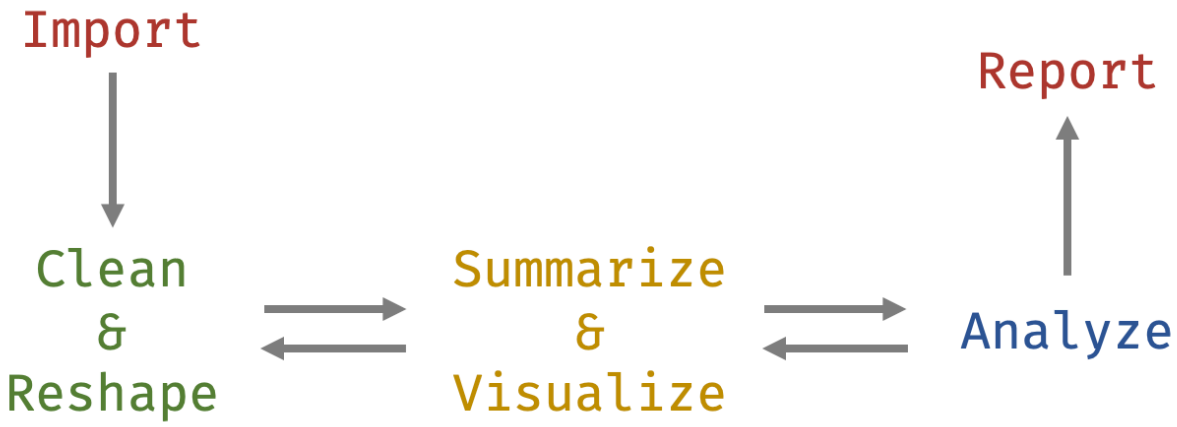
Contents

Preface

Welcome to *R for Researchers: An Introduction*! This book is designed to help you learn to use R and RStudio for working with and analyzing your data. This book is written for researchers in what I'm calling the *human sciences*—the sciences dealing directly with human well-being (e.g. health, behavior, education, psychology, and sociology). Although different in substantive interests, the data situations faced, and analysis approaches used, in these fields are similar. As such, the topics addressed should be useful for researchers working with quantitative data in each of these areas.

My approach herein does not focus on the theoretical aspects of statistics or data science. Instead, it focuses on the application of the methods necessary for a researcher in the humans sciences to independently work with and analyze human data.

This book can work as a companion to other statistical texts or can be a standalone text for learning R. As a companion, Part II (explained below) can be particularly helpful getting started using R for statistical analyses. As a standalone, the entire book helps a researcher work with data throughout the "data analysis lifecycle"—from importing, cleaning, and summarizing the data to analyzing and reporting the results (see figure below).

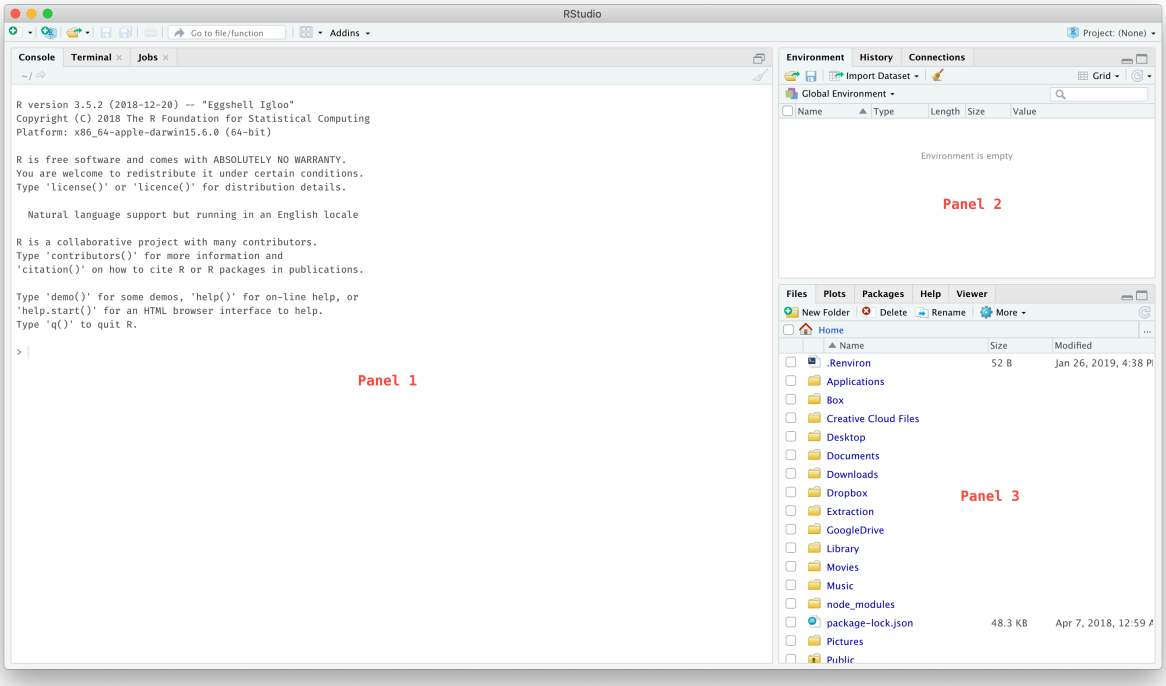


Preparation

Before beginning the book, you will need to download the `R` software www.r-project.org and then the RStudio software www.rstudio.com. `R` will just need to be installed on your computer but you won't interact with it directly (you won't need to open it up). `RStudio`[1] is actually how we will end up using `R`. You can think of `R` as the computer processor while `RStudio` is the keyboard and screen. `R` actually does the work but you'll only use it by using `RStudio`.

Once they are both installed, when you open up `RStudio` you should see the following three panels:

[1] Get the free version of `Rstudio`. Believe me, it doesn't feel like it should be free software.



If you do, then congrats! You are one huge step on the path to using `R` in your research. If it isn't going so smoothly, helps on installing the software can be found on www.rstudio.com, www.r-bloggers.com, and www.statmethods.net. Don't give up it if you run into any obstacles here. But once it is installed and you can open `RStudio`, you are good to go. The remainder of the book will be about actually using it (the fun part!).

> In general, I will refer to anything you do in `RStudio` as using `R`. So, when I say things like: "To make a plot in `R`, we are going to ...", I do not mean to open `R` and start doing stuff there. Instead, I mean to keep using `RStudio` but will not refer to the things we do as in `RStudio` but as in `R`.

The book is divided into three parts.

Part I

1. Chapter 1: The Basics
2. Chapter 2: Working with and Cleaning Your Data
3. Chapter 3: Understanding Your Data (summary statistics, ggplot2)

Part II

4. Chapter 4: Basic Statistical Analyses (T-tests, ANOVA, Linear Regression)
5. Chapter 5: Generalized Linear Models
6. Chapter 6: Multilevel Modeling
7. Chapter 7: Other Modeling Techniques

Part III

8. Chapter 8: Advanced data manipulation
9. Chapter 9: Reproducible Workflow

10. Chapter 10: Where to go from here

At the end of the book, you should be able to: 1) use `R` to perform your data cleaning and data analyses and 2) understand online helps (e.g. www.stackoverflow.com, www.r-bloggers.com) so your potential in `R` becomes nearly limitless.

Enjoy![2]

[2]Note that to return to Tyson's blog, you can click here

Chapter 1: The Basics

> "Success is neither magical nor mysterious. Success is the natural consequence of consistently applying the basic fundamentals." — Jim Rohn

R is an open source statistical software made by statisticians. This means it generally speaks the language of statistics. This is very helpful when it comes running analyses but can be confusing when starting to understand the code.

This book was written to help my students begin to use R for research across the health, behavioral, educational, and psychological sciences. The best way to begin to learn R (in my opinion) is by jumping right into it and using it. To do so, as we learn new concepts, we will go through two main sections:

1. Introduction to the concept with examples of its use
2. Application of the concept through projects provided at tysonbarrett.com

As we start working with data in R, we will go through several main themes, including importing and cleaning data, reshaping and otherwise wrangling[3] data, assessing data via visualizations and tables, and running common statistical tests.

This chapter will provide the background and foundation to start using R. This background revolves around data frames and functions—two general types of objects.

Objects

R is built on a few different types of virtual objects. An object, just like in the physical world, is something you can do things with. In the real world, we have objects that we use regularly. For example, we have chairs. Chairs are great for some things (sitting on, sleeping on, enjoying the beach) and horrible at others (playing basketball, flying, floating in the ocean). Similarly, in R each type of object is useful for certain things. The data types that we will discuss below are certain types of objects.

Because this is so analogous to the real world, it becomes quite natural to work with. You can have many objects in the computer's memory, which allows flexbility in analyzing many different things simply within a single R session.[4] The main object types that you'll work with are presented in the following table (others exist that will come up here and there).

Object	Description
Vector	A single column of data ('a variable')
Data Frame	Multiple vectors put together with observations as rows and variables as columns (much like a Spreadsheet
Function	Takes input and produces output—the workhorse of R
Operator	A special type of function (e.g. '<-')

[3] The hip term for working with data including selecting variables, filtering observations, creating new variables, etc.

[4] An R session is any time you open R do work and then close R. Unless you are saving your workspace (which, in general you shouldn't do), it starts the slate clean–no objects are in memory and no packages are loaded. This is why we use scripts. Also, it makes your workflow extra transparent and replicable.

Figure 1: Chair

Each of these objects will be introduced in this chapter, highlighting their definition and use. For your work, the first objects you work with will be data in various forms. Below, we explain the different data types and how they can combine into what is known as a `data.frame`.

> Early Advice: Don't get overwhelmed. It may feel like there is a lot to learn, but taking things one at a time will work surprisingly quickly. I've designed this book to discuss what you need to know from the beginning. Other topics that are not discussed are things you can learn later and do not need to be of your immediate concern.

Data Types

To begin understanding data in `R`, you must know about vectors. Vectors are, in essence, a single column of data—a variable. In `R` there are three main vector data types (variable types) that you'll work with in research:

- numeric
- factor
- character

The first, **numeric**, is just that: numbers. In `R`, you can make a numeric variable with the code below:

```
x <- c(10.1, 2.1, 4.6, 2.3, 8.9, 4.5, 7.2, 7.9, 3.6, 2.0)
```

The `c()` is a **function**[5] that stands for "concatenate" which basically glues the values inside the paratheses together into one. We use `<-` to put it into `x`. So in this case, `x` (which we could have named anything) is saving those values so we can work with them[6]. If we ran this code, the `x` object would be in the working memory of `R` and will stay there unless we remove it or until the end of the `R` session (i.e., we close `R`).

A **factor** variable is a categorical variable (i.e., only a limited number of options exist). For example, race/ethnicity is a factor variable.

```
race <- c(1, 3, 2, 1, 1, 2, 1, 3, 4, 2)
```

The code above actually produces a numeric vector (since it was only provided numbers). We can quickly tell `R` that it is indeed supposed to be a factor.

```
race <- factor(race,
               labels = c("white", "black", "hispanic", "asian"))
```

The `factor()` function tells `R` that the first thing—`race`—is actually a factor. The additional argument `labels` tells `R` what each of the values means. If we print out `race` we see that `R` has replaced the numeric values with the labels.

```
race
```

```
##  [1] white    hispanic black    white    white    black    white
##  [8] hispanic asian    black
## Levels: white black hispanic asian
```

Finally, and maybe less relevantly, there are character variables. These are words (known as strings). In research this is often where subjects give open responses to a question. These can be used somewhat like factors in a lot of situations.

```
ch <- c("I think this is great.",
        "I would suggest you learn R.",
        "You seem quite smart.")
```

[5] `R` is all about functions. Functions tell `R` what to do with the data. You'll see many more examples throughout the book.

[6] This is a great feature of `R`. It is called "object oriented" which basically means `R` creates objects to work with. I discuss this more in 1.2. Also, the fact that I said `x` "saves" the information is not entirely true but is useful to think this way.

When we combine multiple variables into one, we create a **data.frame**. A data frame is like a spreadsheet table, like the ones you have probably seen in Microsoft's Excel and IBM's SPSS. Here's a simple example:

```
df <- data.frame(x, race)
df
```

```
##       x     race
## 1  10.1    white
## 2   2.1 hispanic
## 3   4.6    black
## 4   2.3    white
## 5   8.9    white
## 6   4.5    black
## 7   7.2    white
## 8   7.9 hispanic
## 9   3.6    asian
## 10  2.0    black
```

We can do quite a bit with the `data.frame` that we called `df`[7]. Once again, we could have called this data frame anything (e.g., `jimmy`, `susan`, `rock`, `scissors` all would work), although I recommend short names. If we hadn't already told `R` that `race` was a factor, we could do this once it is inside of `df` by:

```
df$race <- factor(df$race,
                  labels = c("white", "black", "hispanic", "asian"))
```

In the above code, the `$` reaches into `df` to grab a variable (or column). There are other ways of doing this as is almost always the case in `R`. For example, you may see individuals using:

```
df[["race"]] <- factor(df[["race"]] ,
                       labels = c("white", "black", "hispanic", "asian"))
```

`df[["race"]]` grabs the `race` variable just like `df$race`. Although there are many ways of doing this, we are going to focus on the modern and intuitive ways of doing this more often.

In my experience, the first element of uncomfort that researchers face is the fact that our data is essentially invisible to us. It is "housed" in `df` (not totally accurate but can be thought of this way for now) but it is hard to know exactly what is in `df`. First, we can use:

```
View(df)
```

which prints out the data in spreadsheet-type form to see the data. We can also get nice summaries using the following functions.

```
names(df)
```

```
## [1] "x"    "race"
```

```
str(df)
```

```
## 'data.frame':    10 obs. of  2 variables:
##  $ x   : num  10.1 2.1 4.6 2.3 8.9 4.5 7.2 7.9 3.6 2
##  $ race: Factor w/ 4 levels "white","black",..: 1 3 2 1 1 2 1 3 4 2
```

```
summary(df)
```

```
##        x              race
##  Min.   : 2.00   white   :4
##  1st Qu.: 2.62   black   :3
##  Median : 4.55   hispanic:2
```

[7] I used this name since `df` is common in online helps and other resources.

```
##  Mean   : 5.32   asian   :1
##  3rd Qu.: 7.72
##  Max.   :10.10
```

Functions

Earlier we mentioned that `c()` was a "function." Functions are how we do things with our data. There are probably hundreds of thousands of functions at your reach. In fact, you can create your own! We'll discuss that more in later chapters.

For now, know that each named function has a name (the function name of `c()` is "c"), arguments, and output of some sort. Arguments are the information that you provide the function between the parenthases (e.g. we gave `c()` a bunch of numbers; we gave `factor()` two arguments—the variable and the labels for the variable's levels). Output from a function varies to a massive degree but, in general, the output is what you are using the function for (e.g., for `c()` we wanted to create a vector—a variable—of data where the arguments we gave it are glued together into a variable).

At any point, by typing:

```
?functionname
```

we get information in the "Help" window. It provides information on how to use the function, including arguments, output, and examples.

Sometimes `R` by itself does not have a function that you need. We can download and use functions in **packages**[8]. We will be using several functions from several packages. To install them, we can use the `install.packages()` function,

```
install.packages("rio")
```

This puts the package called `rio` on your computer for you but in order to use the functions it provides, we use

```
library(rio)
```

We now can use the functions from `rio` in this `R` session (i.e. from when you open `R` to when you close it).

After a quick note about operators, you will be shown several functions for both importing and saving data. Note that each have a name, arguments, and output of each.

Operators

A special type of function is called an operator. These take two inputs—a left hand side and a right hand side—and output some value. A very common operator is `<-`, known as the assignment operator. It takes what is on the right hand side and assigns it to the left hand side. We saw this earlier with vectors and data frames. Other operators exist, a few of which we will introduce in the following chapter. But again, an operator is just a special function.

Importing Data

Most of the time you'll want to import data into `R` rather than manually entering it line by line, variable by variable.

We will rely on the `rio` package to do the majority of our data importing. In general, researchers in our fields use `.csv`, `.txt`, `.sav`, and `.xlxs`. For the most part, importing will be straightforward. Regardless of

[8] A package is an extension to `R` that gives you more functions–abilities–to work with data. Anyone can write a package, although to get it on the Comprehensive R Archive Network (CRAN) it needs to be vetted to a large degree. In fact, after some practice, you could write a package to help you more easily do your work.

the file type, we will use `rio::import()`. This notation means that the package `rio` has a function called `import()`.

```
## Import a CSV file located in the same folder as your RMarkdown file
our_data <- rio::import("your_data_file.csv")
```

This will work for `.csv`, `.txt`, `.sav`, `.xlsx`, and many others files. In the example, we assigned our imported file to `df` using `<-`. This is necessary to let `R` remember the data for it to use for other stuff. The object `our_data` now contains the data saved in the CSV file called `"your_data_file.csv"`. Note that at the end of the lines you see that I left a **comment** using `#`. I used two for stylistic purposes but only one is necessary. Anything after a `#` is not read by the computer; it's just for us humans.

> Heads up! It is important to know where your data file is located. In my experience, this is where students struggle the most at the very beginning. If the file is in the same folder as the RStudio project, we can use the `here::here()` function to point to the project's folder. More will be discussed about RStudio projects after the discussion on saving data.

If you have another type of data file to import that `rio::import()` doesn't work well with, online helps found on sites like www.stackoverflow.com and www.r-bloggers.com often have the solution.

Saving Data

Occassionally you'll want to save some data that you've worked with (usually this is not necessary). When necessary, you can use `rio::export()`.

```
rio::export(df, file="file.csv")  ## to create a CSV data file
```

`R` automatically saves missing data as `NA` since that is what it is in `R`. But often when we write a CSV file, we might want it as blank or some other value. If that's the case, we can add another argument `na = " "` after the file argument.

Again, if you ever have questions about the specific arguments that a certain function has, you can simply run `?functionname`. So, if you were curious about the different arguments in `export` simply run: `?export`. In the pane with the files, plots, packages, etc. a document will show up to give you more informaton.

Apply it

This link contains a folder complete with an Rstudio project file, an RMarkdown file, and a few data files. Download it and unzip it to do the following steps.

Step 1

Open the `Chapter1.Rproj` file. This will open up RStudio for you.

Step 2

Once RStudio has started, in the panel on the lower-right, there is a `Files` tab. Click on that to see the project folder. You should see the data files and the `Chapter1.Rmd` file. Click on the `Chapter1.Rmd` file to open it. It should open in the upper-left panel. It has words and code already put in it. Starting at the first code chunk (has "'{r} right above it), click on the first line and run the code. The code can be run using "Cmd +"Enter" or the Run button at the top of RStudio.

Step 3

Run the code to import the CSV file. At the end of that code chunk there is a link that is simply `df`. Run that line. It should print out some of the data in `df`. If you see the data printed out, you have done these

steps correctly. Feel free to play around with this file and import the other data file types and run other code we've shown in this chapter.

Conclusions

R is designed to be flexible and do just about anything with data that you'll need to do as a researcher. With this chapter under your belt, you can now read basic R code, import and save your data. The next chapter will introduce the "tidyverse" of methods that can help you join, reshape, summarize, group, and much more.

Chapter 2: Working with and Cleaning Your Data

> "Organizing is what you do before you do something, so that when you do it, it is not all mixed up." — A. A. Milne

In order to work with and clean your data in the most modern and straightforward way, we are going to be using the "tidyverse" group of methods. The tidyverse[9] is a group of packages[10] that provide a simple syntax that can do many basic (and complex) data manipulating. They form a sort of "grammar" of data manipulation that simplifies both the coding approach and the way researchers think about working with data.

The group of packages can be downloaded via:

```
install.packages("tidyverse")
```

After downloading it, to use its functions, simply use:

```
library(tidyverse)
```

Note that when we loaded tidyverse it loaded several packages and told you of "conflicts". These conflicts are where two or more functions across different packages have the same name. These functions with the same name will almost invariably differ in what they do. In this situation, the last loaded package is the one that `R` will use by default. For example, if we loaded two packages—`awesome` and `amazing`—and both had the function `make_really_cool` and we loaded `awesome` and then `amazing` as so:

```
library(awesome)
library(amazing)
```

`R` will automatically use the function from `amazing`. We can still access the `awesome` version of the function (because, again, even though the name is the same, they won't necessarily do the same things for you). We can do this by:

```
awesome::make_really_cool(args)
```

In essence, the `::` grabs the function from inside of the package and let's you use that.

That's a bit of an aside, but know that you can always get at a function even if it is "masked" from your current session.

Tidy Methods

I'm introducing this to you for a couple reasons.

[9] Hadley Wickham (2016). tidyverse: Easily Install and Load 'Tidyverse' Packages. R package version 1.0.0. https://CRAN.R-project.org/package=tidyverse

[10] Remember, a package is an extension to `R` that gives you more functions that you can easily load into `R`.

1. It simplifies the code and makes the code more readable. It is often worthwhile to make sure the code is readable for, as the saying goes, there are always at least two collaborators on any project: you and future you.
2. It is the cutting edge of all things `R`. The most influential individuals in the `R` world, including the makers and maintainers of `RStudio`, use these methods and syntax.

The majority of what you'll need to do with data as a researcher will be covered by these functions. The goal of these packages is to help tidy up your data. *Tidy data* is based on columns being variables and rows being observations. This depends largely on your data and research design but the definition is still the same—columns are variables and rows are observations. It is the form that data needs to be in to analyze it, whether that analysis is by graphing, modeling, or other means.

There are several methods that help create tidy data:

1. Piping
2. Selecting and Filtering
3. Mutate and Transmute
4. Grouping and Summarizing
5. Reshaping
6. Joining (merging)

> Heads up! Understanding these tools requires an understanding of what ways data can be moved around. For example, reshaping can refer to moving data into a more wide-format or long-format, can refer to summarizing or aggregating, and can refer to joining or binding. All of these are necessary to work with data flexibly. Because of this, we suggest taking your time to fully understand what each function is doing with the data.

Much of these may be things you have done in other tools such as spreadsheets. The copy-and-paste approach is seriously error prone and is not reproducible. Taking your time to learn these methods will be well worth it.

Tidy (Long) Form

Before diving in, I want to provide some examples of tidy data (also known as "long form") and how they are different from another form, often called "wide form."

Only when the data is cross-sectional and each individual is a row does this distinction not matter much. Otherwise, if there are multiple measurement periods per individual (longitudinal design), or there are multiple individuals per cluster (clustered design), the distinction between wide and long is very important for modeling and visualization.

Wide Form

Wide form generally has one unit (i.e. individual) per row. This generally looks like:

```
##    ID Var_Time1 Var_Time2
## 1   1    1.0403     0.861
## 2   2   -0.1874     0.740
## 3   3   -1.1360     0.461
## 4   4   -0.6775     0.321
## 5   5   -0.8864     0.485
## 6   6   -0.0565     0.978
## 7   7   -0.8965     0.365
## 8   8   -0.1870     0.713
## 9   9   -1.4083     0.578
## 10 10    0.1281     0.653
```

Notice that each row has a unique ID, and some of the columns (all in this case) are variables at specific time points. This means that the variable is split up and is not a single column. This is a common way to collect and store data in our fields and so your data likely looks similar to this.

Long Form

In contrast, long format has the lowest nested unit as a single row. This means that a single ID can span multiple rows, usually with a unique time point for each row as so:

```
##   ID Time   Var
## 1  1    1 0.345
## 2  1    2 0.147
## 3  1    3 0.505
## 4  1    4 0.202
## 5  2    1 0.824
## 6  2    2 0.970
## 7  3    1 0.463
## 8  3    2 0.674
## 9  3    3 0.243
```

Notice that a single ID spans multiple columns and that each row has only one time point. Here, time is nested within individuals making it the lowest unit. Therefore, each row corresponds to a single time point. Generally, this is the format we want for most modeling techniques and most visualizations.

This is a very simple example of the differences between wide and long format. Although much more complex examples are probably available to you, we will start with this for now. Chapter 8 will delve into more complex data designs and how to reshape those. Toward the end of this chapter, we show reshaping with more simple data structures.

Data Used for Examples

To help illustrate each aspect of working with and cleaning your data, we are going to use real data from the National Health and Nutrition Examiniation Survey (NHANES). I've provided this data at https://tysonstanley.github.io/assets/Data/NHANES.zip. I've cleaned it up somewhat already.

Let's quickly read that data in so we can use it throughout the remainder of this chapter.

```
library(rio)
dem_df <- import("NHANES_demographics_11.xpt")
med_df <- import("NHANES_MedHeath_11.xpt")
men_df <- import("NHANES_MentHealth_11.xpt")
act_df <- import("NHANES_PhysActivity_11.xpt")
```

Now we have four separate, but related, data sets in memory:

1. `dem_df` containing demographic information
2. `med_df` containing medical health information
3. `men_df` containing mental health information
4. `act_df` containing activity level information

Since all of them have all-cap variable names, we are going to quickly change this with a little trick:

```
names(dem_df) <- tolower(names(dem_df))
names(med_df) <- tolower(names(med_df))
names(men_df) <- tolower(names(men_df))
names(act_df) <- tolower(names(act_df))
```

This takes the names of the data frame (on the right hand side), changes them to lower case and then reassigns them to the names of the data frame.[11]

We will now go through each aspect of the tidy way of working with data using these four data sets.

Piping

Let's introduce a few major themes in this tidyverse. First, the pipe operator – `%>%`. It helps simplify the code and makes things more readable. It takes what is on the left hand side and puts it in the right hand side's function.

```
dem_df %>% head()
```

So the above code takes the data frame `dem_df` and puts it into the `head` function. This does the same thing as `head(df)`.

To illustrate its strength, consider the following example. We take `dem_df`, group it by gender, and then count.

```
dem_df %>%
  group_by(riagendr) %>%
  summarize(mean_age = mean(ridageyr, na.rm = TRUE))
```

```
## # A tibble: 2 x 2
##   riagendr mean_age
##      <dbl>    <dbl>
## 1        1     31.1
## 2        2     31.7
```

Without using the pipe, we would need to use something like:

```
summarize(group_by(dem_df, riagendr), mean_age = mean(ridageyr, na.rm = TRUE))
```

```
## # A tibble: 2 x 2
##   riagendr mean_age
##      <dbl>    <dbl>
## 1        1     31.1
## 2        2     31.7
```

Although it is fewer lines, this is notably harder to understand. Instead of reading from left to right, top to bottom, we must read inside out, which is unnatural for English reading individuals.

Select Variables and Filter Observations

We often want to subset our data in some way before we do many of our analyses. We may want to do this for a number of reasons (e.g., easier cognitively to think about the data, the analyses depend on the subsetting). The code below show the two main ways to subset your data:

1. *selecting* variables and
2. *filtering* observations.

To select three variables (i.e. gender ["riagendr"], age ["ridageyr"], and ethnicity ["ridreth1"]) we:

```
selected_dem <- dem_df %>%
  select(riagendr, ridageyr, ridreth1)
```

[11] Note that these are not particularly helpful names, but they are the names provided in the original data source. If you have questions about the data, visit http://wwwn.cdc.gov/Nchs/Nhanes/Search/Nhanes11_12.aspx.

Now, `selected_dem` has three variables and all the observations.

We can also filter (i.e. take out observations we don't want):

```
filtered_dem <- dem_df %>%
  filter(riagendr == 1)
```

Since when `riagendr == 1` the individual is male, `filtered_dem` only has male participants. We can add multiple filtering options as well:

```
filtered_dem <- dem_df %>%
  filter(riagendr == 1 & ridageyr > 16)
```

We now have only males that are older than 16 years old. We used `&` to say we want **both** conditions to be met. Alternatively, we could use:

```
filtered_dem <- dem_df %>%
  filter(riagendr == 1 | ridageyr > 16)
```

By using `|` we are saying we want males **or** individuals older than 16. In other words, if either are met, that observation will be kept.

Finally, we can do all of these in one step:

```
filtered_selected_dem <- dem_df %>%
  select(riagendr, ridageyr, ridreth1) %>%
  filter(riagendr == 1 & ridageyr > 16)
```

where we use two `%>%` operators to grab `dem_df`, select the three variables, and then filter the rows that we want.

Mutate Variables

Earlier we showed how to create a new variable or clean an existing one. We used the base `R` way for these. However, a great way to add and change variables is using `mutate()`. We grab `df`, and create a variable called `citizen` using the factored version of `dmdcitzn` and assign it back into `df`.

```
## Our Grouping Variable as a factor
dem_df <- dem_df %>%
  mutate(citizen = factor(dmdcitzn))
```

The benefit is in the readability of the code. We repeat things like the name of the data frame much less. I highly recommend working this way.

Grouping and Summarizing

A major aspect of analysis is comparing groups. Lucky for us, this is very simple in `R`. I call it the three step summary:

1. Data
2. Group by
3. Summarize

```
## Three step summary:
dem_df %>%                          ## 1. Data
  group_by(citizen) %>%             ## 2. Group by
  summarize(N = n())                ## 3. Summarize
```

```
## Warning: Factor `citizen` contains implicit NA, consider using
```

```
## `forcats::fct_explicit_na`
```

```
## # A tibble: 4 x 2
##   citizen     N
##   <fct>   <int>
## 1 1        8685
## 2 2        1040
## 3 7          26
## 4 <NA>        5
```

The output is very informative. The first column is the grouping variable and the second is the N (number of individuals) by group. We can quickly see that there are four levels, currently, to the citizen variable. After some reading of the documentation we see that `1 = Citizen` and `2 = Not a Citizen`. A value of `7` it turns out is a placeholder value for missing. And finally we have an NA category. It's unlikely that we want those to be included in any analyses, unless we are particularly interested in the missingness on this variable. So let's do some simple cleaning to get this where we want it. To do this, we will use the `furniture` package.

```
install.packages("furniture")
```

```
dem_df <- dem_df %>%
  mutate(citizen = washer(citizen, 7),                ## Changes all 7's to NA's
         citizen = washer(citizen, 2, value = 0))     ## Changes all 2's to 0's
```

Now, our citizen variable is cleaned, with `0` meaning not a citizen and `1` meaning citizen. Let's rerun the code from above with the three step summary:

```
## Three step summary:
dem_df %>%                              ## 1. Data
  group_by(citizen) %>%                 ## 2. Group by
  summarize(N = n())                    ## 3. Summarize
```

```
## Warning: Factor `citizen` contains implicit NA, consider using
## `forcats::fct_explicit_na`
```

```
## # A tibble: 3 x 2
##   citizen     N
##   <fct>   <int>
## 1 0        1040
## 2 1        8685
## 3 <NA>       31
```

Its clear that the majority of the subjects are citizens. We can also check multiple variables at the same time, just separating them with a comma in the `summarize` function.

```
## Three step summary:
dem_df %>%                              ## 1. Data
  group_by(citizen) %>%                 ## 2. Group by
  summarize(N = n(),                    ## 3. Summarize
            Age = mean(ridageyr, na.rm=TRUE))
```

```
## Warning: Factor `citizen` contains implicit NA, consider using
## `forcats::fct_explicit_na`
```

```
## # A tibble: 3 x 3
##   citizen     N   Age
##   <fct>   <int> <dbl>
## 1 0        1040  37.3
## 2 1        8685  30.7
## 3 <NA>       31  40.4
```

We used the `n()` function (which gives us counts) and the `mean()` function which, shockingly, gives us the mean. Note that if there are `NA`'s in the variable, the mean (and most other functions like it) will give the result `NA`. To have `R` ignore these, we tell the `mean` function to remove the `NA`'s when you compute this using `na.rm=TRUE`.

We can also use `furniture::table1()` to do the three-step summary. By default, `table1()` will give you the mean and SD of numeric variables and counts and percentages for factors.

```
## Three step summary:
dem_df %>%                          ## 1. Data
  group_by(citizen) %>%             ## 2. Group by
  table1(Age = ridageyr,            ## 3. Summarize
         Sex = factor(riagendr))
```

```
## Warning: Factor `citizen` contains implicit NA, consider using
## `forcats::fct_explicit_na`
```

```
## Using dplyr::group_by() groups: citizen
```

```
##
##
##             citizen
##       0           1
##       n = 1040    n = 8685
##  Age
##       37.3 (18.5) 30.7 (25.1)
##  Sex
##     1 523 (50.3%) 4315 (49.7%)
##     2 517 (49.7%) 4370 (50.3%)
##
```

This pattern of grouping and summarizing is something that will follow us throughout the book. It's a great way to get to know your data well and to make decisions on what to do next with your data.

Reshaping

Earlier we described the differences between wide and long (or tidy) form. Now that you have an idea of the differences, we are going to introduce how to change from one to the other. Several functions exist for just this purpose, including `gather()` from the `tidyr` package and `reshape()` in the default `stats` package. Since these can be limited in certain situations, we are going to teach two functions that can be used in nearly any reshaping situation. For now, though, we will keep in simple.

First, we will go from wide to long form using `long()` from the `furniture` package.[12] We are using the ficticious data for the example of wide format from above.

```
#library(furniture)
df_wide <- data.frame("ID"=c(1:10),
                      "Var_Time1"=rnorm(10),
                      "Var_Time2"=runif(10))
df_long <- long(df_wide,
                c("Var_Time1", "Var_Time2"),
                v.names = "Var")
```

```
## id = ID
```

[12] Although the furniture package isn't in the `tidyverse`, it is a valuable package to use with the other `tidyverse` packages.

```
df_long
```

```
##       ID time     Var
## 1.1    1    1  0.6603
## 2.1    2    1  1.6265
## 3.1    3    1  0.4962
## 4.1    4    1  0.1802
## 5.1    5    1 -1.3975
## 6.1    6    1 -1.0848
## 7.1    7    1  1.2837
## 8.1    8    1  1.0592
## 9.1    9    1  0.6406
## 10.1 10    1 -0.5674
## 1.2    1    2  0.0259
## 2.2    2    2  0.2379
## 3.2    3    2  0.9808
## 4.2    4    2  0.7983
## 5.2    5    2  0.3400
## 6.2    6    2  0.6250
## 7.2    7    2  0.8894
## 8.2    8    2  0.3452
## 9.2    9    2  0.7314
## 10.2 10    2  0.5624
```

We provided the data (`df_wide`), the time varying variables (`c("Var_Time1", "Var_Time2")`), and told it what we should name the value variable (`v.names = "Var"`). Note that the function guessed, based on its name, that the variable `ID` was the identifying variable.

This function automatically assumes each observation is a time point, thus the `time` variable. We can easily change that by adding the argument `timevar` and giving it a name (e.g., `timevar = "cluster"`).

And now we will go from long to wide using `wide()` from the same package.

```
df_long <- data.frame("ID"=c(1,1,1,1,2,2,3,3,3),
                      "Time"=c(1,2,3,4,1,2,1,2,3),
                      "Var"=runif(9))
df_wide <- wide(df_long,
                v.names = c("Var"),
                timevar = "Time")
```

```
## id = ID
```

```
df_wide
```

```
##   ID   Var.1 Var.2 Var.3 Var.4
## 1  1 0.95875 0.603 0.308 0.792
## 5  2 0.00205 0.283    NA    NA
## 7  3 0.74267 0.840 0.852    NA
```

Here, we provided the data `df_long` and the variable name (`Var`) that had the values and (`Time`) that contained the time labels (in this case, just numbers). With a little bit of code we can move data around without any copy-pasting that is so error-prone. Again, note that the function guessed, based on its name, that the variable `ID` was the identifying variable.

Joining (merging)

The final topic in the chapter is joining data sets. This is common in many situations including large surveys (e.g., a demographics set, physical activity set, family characteristics set), health records (financial data, doctors notes, diagnosis data), and longitudinal studies (data from wave I, data from wave II).

We currently have 4 data sets that have mostly the same people in them but with different variables. One tells us about the demographics; another gives us information on mental health. We may have questions that ask whether a demographic characteristics is related to a mental health factor. This means we need to merge, or join, our data sets.[13]

When we merge a data set, we combine them based on some ID variable(s). Here, this is simple since each individual is given a unique identifier in the variable `seqn`. Within the `dplyr` package (part of the tidyverse) there are four main joining functions: `inner_join`, `left_join`, `right_join` and `full_join`. Each join combines the data in slightly different ways.

The figure below shows each type of join in Venn Diagram form. Each are discussed below.

Inner Join

Here, only those individuals that are in both data sets that you are combining will remain. So if person "A" is in data set 1 and not in data set 2 then he/she will not be included.

```
inner_join(df1, df2, by="IDvariable")
```

Left or Right Join

This is similar to inner join but now if the individual is in data set 1 then `left_join` will keep them even if they aren't in data set 2. `right_join` means if they are in data set 2 then they will be kept whether or not they are in data set 1.

```
left_join(df1, df2, by="IDvariable")   ## keeps all in df1
right_join(df1, df2, by="IDvariable")  ## keeps all in df2
```

Full Join

This one simply keeps all individuals that are in either data set 1 or data set 2.

```
full_join(df1, df2, by="IDvariable")
```

Each of the left, right and full joins will have missing values placed in the variables where that individual wasn't found. For example, if person "A" was not in df2, then in a full join they would have missing values in the df1 variables.

For our NHANES example, we will use `full_join` to get all the data sets together. Note that in the code below we do all the joining in the same overall step.

```
df <- dem_df %>%
  full_join(med_df, by="seqn") %>%
  full_join(men_df, by="seqn") %>%
  full_join(act_df, by="seqn")
```

So now `df` is the the joined data set of all four. We started with `dem_df` joined it with `med_df` by `seqn` then joined that joined data set with `men_df` by `seqn`, and so on.

For many analyses in later chapters, we will use this new `df` object that is the combination of all the data sets that we had before. Whenever this is the case, this data will be explicitly referenced. Note, that in

[13] Note that this is different than adding new rows but not new variables. Merging requires that we have at least some overlap of individuals in both data sets.

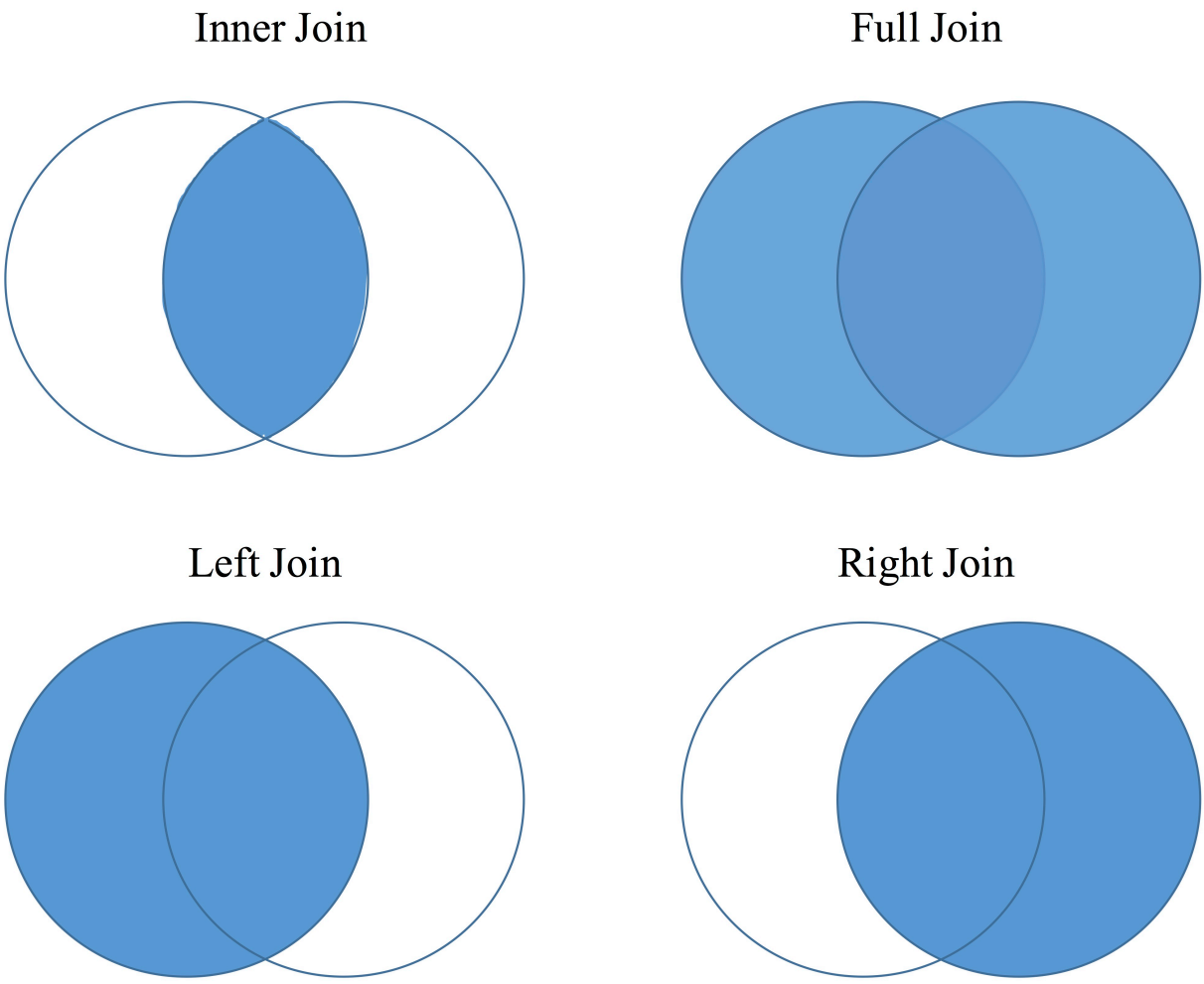


Figure 2: Joining

addition to what was shown in this chapter, a few other cleaning tasks were done on the data. This final version of `df` can be found at tysonstanley.github.io/R.

Wrap it up

Let's put all the pieces together that we've learned in the chapter together on this new `df` data frame we just created. Below, we using piping, selecting, filtering, grouping, and summarizing.

First, let's create an overall depression variable that is the sum of all the depression items (we could do IRT or some other way of combining them but that is not the point here). Below, we do three things:

1. We clean up each depression item using `furniture::washer()` since both "7" and "9" are placeholders for missing values. We take override the original variable with the cleaned one.
2. Next, we sum all the items into a new variable called `dep`.
3. Finally, we create a dichotomized variable called `dep2` using nested `ifelse()` functions. The `ifelse()` statements read: if condition holds (`dep` is greater than or equal to `10`), then `1`, else if condition holds (`dep` is less than `10`), then `0`, else `NA`. The basic build of the function is: `ifelse(condition, value if true, value if false)`.

In the end, we are creating a new data frame called `df2` with the new and improved `df` with the items cleaned and depression summed and dichotomized.

```
df2 <- df %>%
  mutate(dpq010 = washer(dpq010, 7,9),
         dpq020 = washer(dpq020, 7,9),
         dpq030 = washer(dpq030, 7,9),
         dpq040 = washer(dpq040, 7,9),
         dpq050 = washer(dpq050, 7,9),
         dpq060 = washer(dpq060, 7,9),
         dpq070 = washer(dpq070, 7,9),
         dpq080 = washer(dpq080, 7,9),
         dpq090 = washer(dpq090, 7,9),
         dpq100 = washer(dpq100, 7,9)) %>%
  mutate(dep = dpq010 + dpq020 + dpq030 + dpq040 + dpq050 +
               dpq060 + dpq070 + dpq080 + dpq090) %>%
  mutate(dep2 = factor(ifelse(dep >= 10, 1,
                       ifelse(dep < 10, 0, NA))))
```

After these adjustments, let's select, filter, group by, and summarize.

```
df2 %>%
  select(ridageyr, riagendr, mcq010, dep) %>%
  filter(ridageyr > 10 & ridageyr < 40) %>%
  group_by(riagendr) %>%
  summarize(asthma = mean(mcq010, na.rm=TRUE),
            depr   = mean(dep, na.rm=TRUE))
```

```
## # A tibble: 2 x 3
##   riagendr asthma  depr
##      <dbl>  <dbl> <dbl>
## 1        1   1.82  2.61
## 2        2   1.82  3.58
```

We can see that males (`riagendr = 1`) have nearly identical asthma levels but lower depression levels than their female counterparts.

General Cleaning of the Data Set

A package known as `janitor` provides some nice functions that add to the ideas presented in this chapter. Specifically, two functions are of note:

1. `clean_names()` – cleans up the names of the data frame to make them easier to type and make sure it works well with `R`.
2. `remove_empty()` – removes either empty columns or empty rows (columns or rows that have only missing values).

These can be used in tandem, like so:

```
df3 <- df2 %>%
  janitor::clean_names() %>%
  janitor::remove_empty("cols") %>%
  janitor::remove_empty("rows")
```

Ultimately, I hope you see the benefit to using these methods. With these methods, you can clean, reshape, and summarize your data. Because these are foundational, we will apply these methods throughout the book.

Apply It

This link contains a folder complete with an Rstudio project file, an RMarkdown file, and a few data files. Download it and unzip it to do the following steps.

Step 1

Open the `Chapter2.Rproj` file. This will open up RStudio for you.

Step 2

Once RStudio has started, in the panel on the lower-right, there is a `Files` tab. Click on that to see the project folder. You should see the data files and the `Chapter2.Rmd` file. Click on the `Chapter2.Rmd` file to open it. In this file, import the data, create a new variable, select three varaibles, and summarize a variable using the three step summary.

Once that code is in the file, click the `knit` button. This will create an HTML file with the code and output knitted together into one nice document. This can be read into any browser and can be used to show your work in a clean document.

Chapter 3: Exploring Your Data with Tables and Visuals

"If you can't explain it simply, you don't understand it well enough." — Albert Einstein

We are going to take what we've learned from the previous two chapters and use them together to have simple but powerful ways to understand your data. This chapter will be split into two sections:

1. Descriptive Statistics
2. Visualizations

The two go hand-in-hand in understanding what is happening in your data before you attempt any modeling procedures. We are often most interested in three things when exploring our data: *understanding distributions*, *understanding relationships*, and *looking for outliers or errors*.

Descriptive Statistics

Several methods of discovering descriptives in a succinct way have been developed for `R`. My favorite (full disclosure: it is one that I made so I may be biased) is the `table1` function in the `furniture` package.

This function has been designed to be simple and complete. It produces a well-formatted table that you can easily export and use as a table in a report or article.[14] We'll use the `df` object we created in Chapter 2.

Let's get descriptive statistics for five of the variables: asthma, race, depression, family size, and sedentary behavior.

```
#library(furniture)
df %>%
  table1(asthma, race, dep, famsize, sed)
```

```
##
##
##                     Mean/Count (SD/%)
##                     n = 4437
##  asthma
##     No Asthma       3774 (85.1%)
##     Asthma          663 (14.9%)
##  race
##     MexicanAmerican 392 (8.8%)
##     OtherHispanic   442 (10%)
##     White           1733 (39.1%)
##     Black           1161 (26.2%)
##     Other           709 (16%)
##  dep
```

[14] It is called "table1" because a nice descriptive table is often found in the first table of many academic papers.

```
##                     3.2 (4.5)
##  famsize
##                     2.7 (1.5)
##  sed
##                     369.3 (201.2)
##
```

This quickly gives you means and standard deviations or counts and percentages.

The code below shows the means/standard devaitions or counts/percentages by a grouping variable—in this case, `asthma`.

```
df %>%
  group_by(asthma) %>%
  table1(race, dep, famsize, sed)
```

```
##
##
##                            asthma
##                     No Asthma     Asthma
##                     n = 3774      n = 663
##  race
##     MexicanAmerican 350 (9.3%)    42 (6.3%)
##     OtherHispanic   380 (10.1%)   62 (9.4%)
##     White           1453 (38.5%)  280 (42.2%)
##     Black           958 (25.4%)   203 (30.6%)
##     Other           633 (16.8%)   76 (11.5%)
##  dep
##                     3.0 (4.3)     4.4 (5.2)
##  famsize
##                     2.7 (1.5)     2.6 (1.5)
##  sed
##                     365.2 (201.1) 392.4 (200.0)
##
```

We can also test for differences by group as well.

```
df %>%
  group_by(asthma) %>%
  table1(race, dep, famsize, sed,
         test = TRUE)
```

```
##
##
##                            asthma
##                     No Asthma     Asthma        P-Value
##                     n = 3774      n = 663
##  race                                           <.001
##     MexicanAmerican 350 (9.3%)    42 (6.3%)
##     OtherHispanic   380 (10.1%)   62 (9.4%)
##     White           1453 (38.5%)  280 (42.2%)
##     Black           958 (25.4%)   203 (30.6%)
##     Other           633 (16.8%)   76 (11.5%)
##  dep                                            <.001
##                     3.0 (4.3)     4.4 (5.2)
##  famsize                                        0.496
##                     2.7 (1.5)     2.6 (1.5)
```

```
##  sed                                              0.001
##                      365.2 (201.1) 392.4 (200.0)
##
```

Several other options exist for you to play around with, including obtaining medians and ranges and removing a lot of the white space of the table.

With three or four short lines of code we can get a good idea about variables that may be related to the grouping variable and any missingness in the factor variables. There's much more you can do with `table1` and there are vignettes and tutorials available to learn more.[15]

Other quick descriptive functions exist; here are a few of them.

```
## install with install.packages("psych")
psych::describe(df)
```

```
## install with install.packages("Hmisc")
Hmisc::describe(df)
```

```
## install with install.packages("janitor")
janitor::tabyl(df)
```

There is truly no shortage of descriptive information that you can obtain within `R`.

Visualizations

Understanding your data, in my experience, almost always requires visualizations[16]. If we are going to use a model of some sort, understanding the distributions and relationships beforehand are very helpful in interpreting the model and catching errors in the data. Also finding any outliers or errors that could be highly influencing the modeling should be understood beforehand.

For simple but appealing visualizations we are going to be using `ggplot2`. This package is used to produce professional level plots for many journalism organizations (e.g. five-thrity-eight). These plots are quickly presentation quality and can be used to impress your boss, your advisor, or your friends.

Using `ggplot2`

This package is included in the `tidyverse` and is built on the "Grammar of Graphics" principles of data visualization. In essence, it is built on adding layers to a plot. Below, we walk through quick ways to start to visualize our data.

First, we have a nice `qplot` function that is short for "quick plot." It quickly decides what kind of plot is useful given the data and variables you provide.

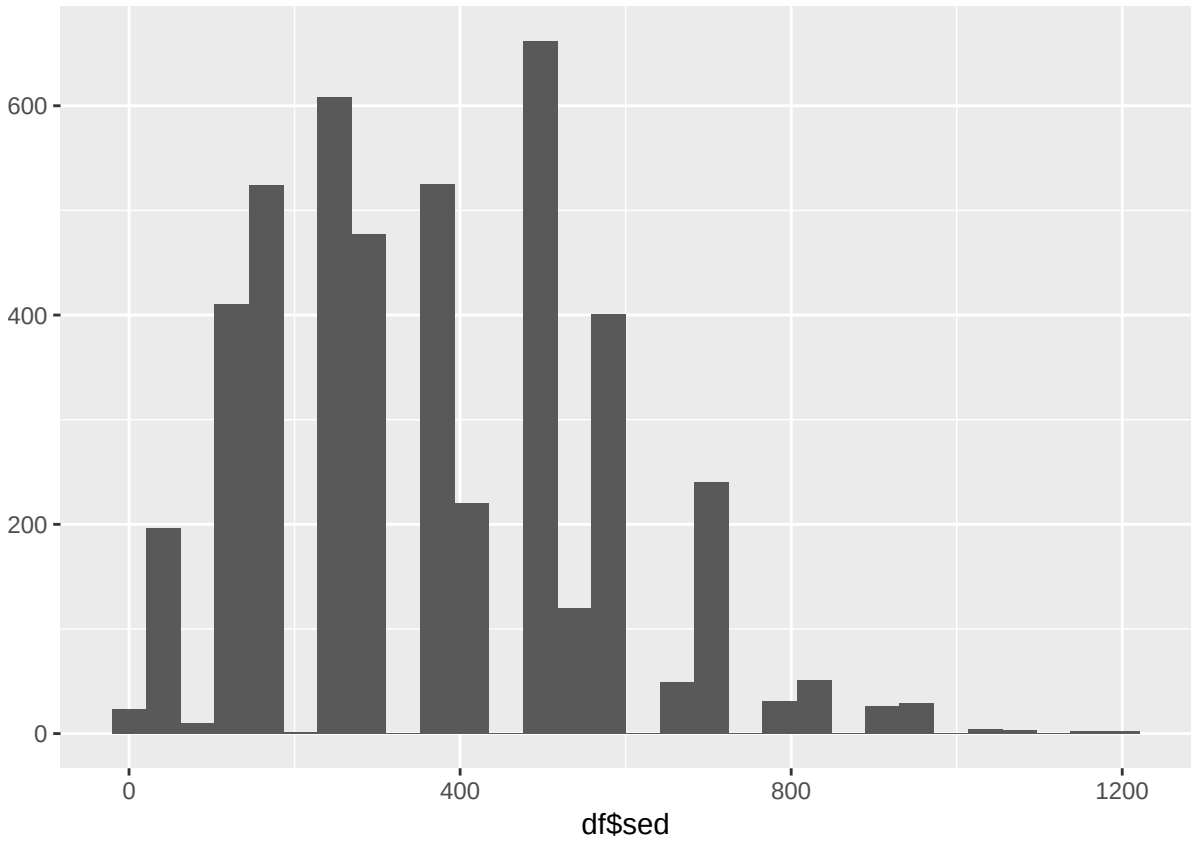
```
qplot(df$sed)          ## Makes a simple histogram
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

```
## Warning: Removed 18 rows containing non-finite values (stat_bin).
```

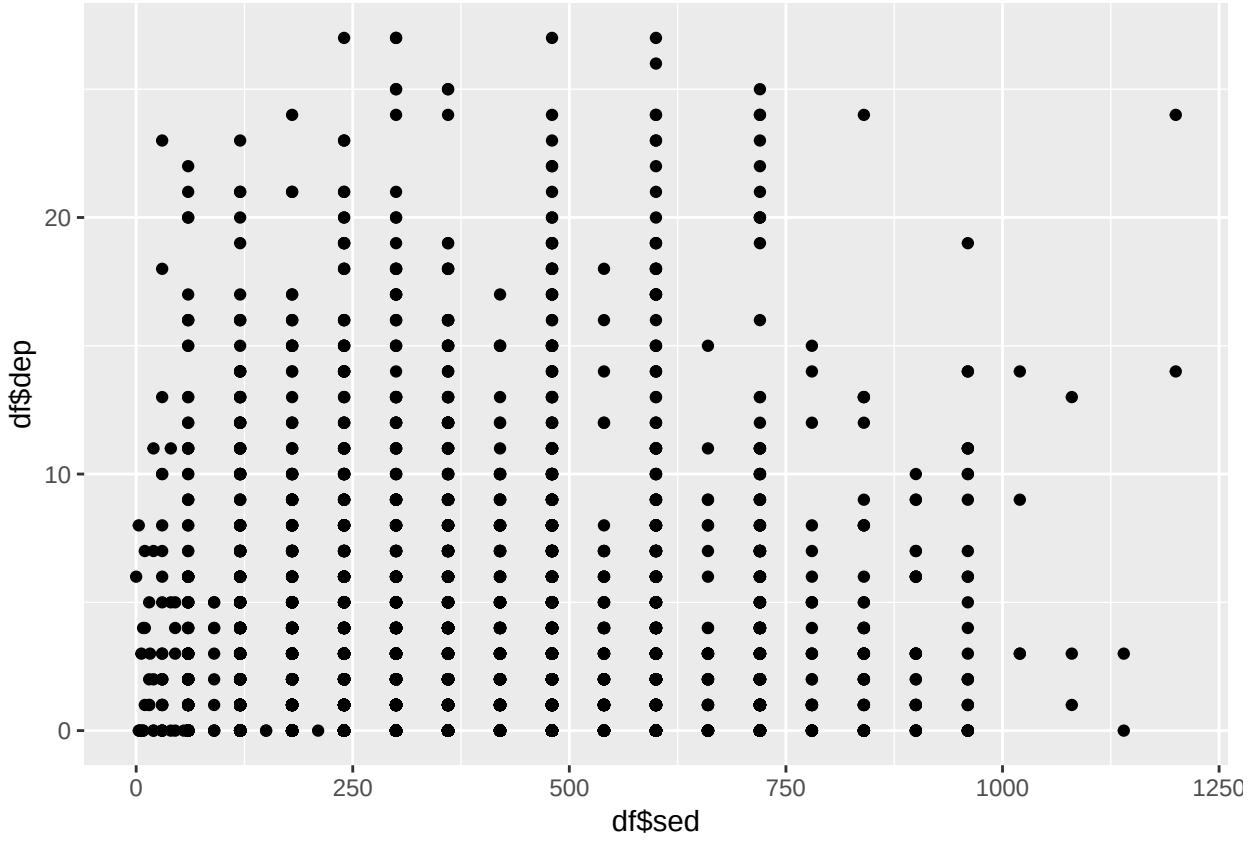
[15] tysonstanley.github.io

[16] If you'd like to learn more about data visualization, see Kieran Healy's *Data Visualization: A Practical Introduction* at http://socviz.co.




```
qplot(df$sed, df$dep)  ## Makes a scatterplot
```

```
## Warning: Removed 18 rows containing missing values (geom_point).
```



For a bit more control over the plot, you can use the `ggplot` function. The first piece is the `ggplot` piece. From there, we add layers. These layers generally start with `geom_` then have the type of plot. Below, we start with telling `ggplot` the basics of the plot and then build a boxplot.

The key pieces of ggplot:

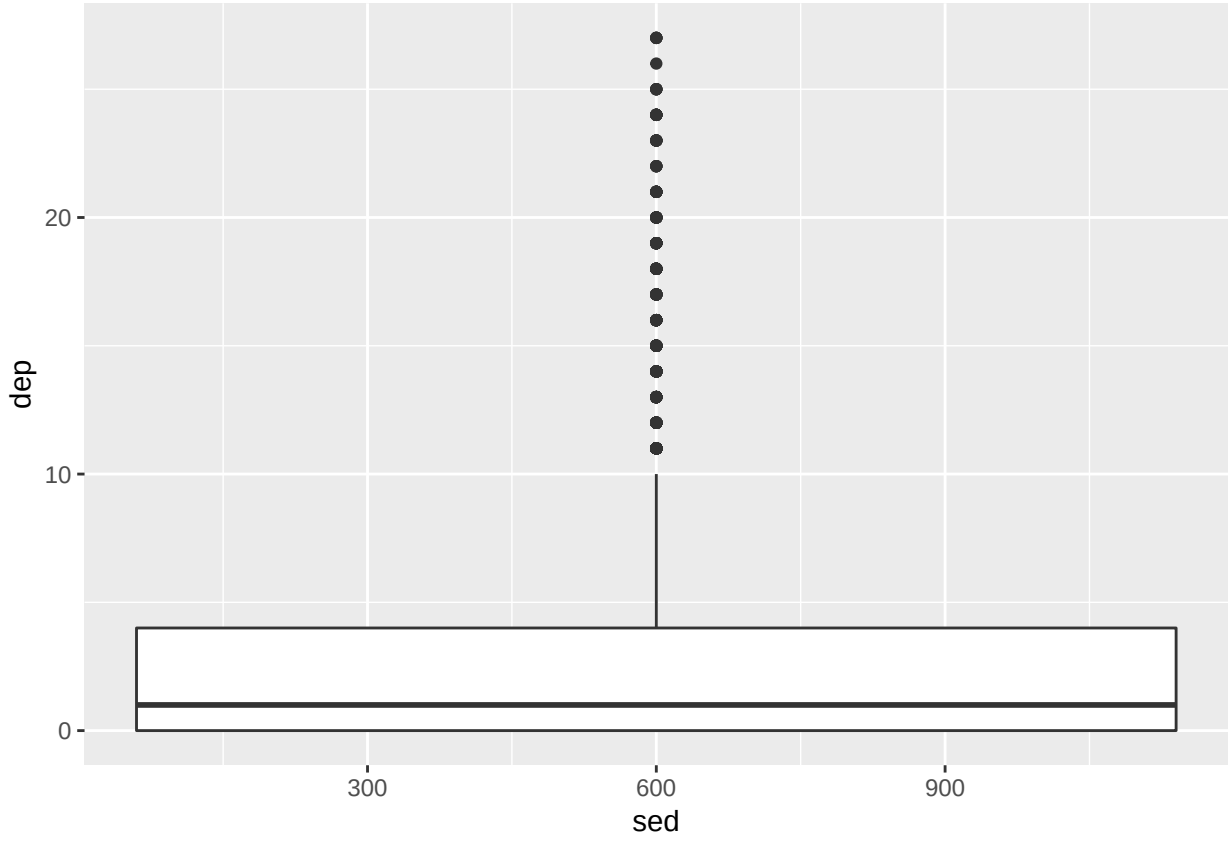
1. `aes()` is how we tell `ggplot()` to look at the variables in the data frame.
2. Within `aes()` we told it that the x-axis is the variable "C" and the y-axis is the variable "D" and then we color it by variable "C" as well (which we told specifically to the boxplot).
3. `geom_` functions are how we tell `ggplot` what to plot—in this case, a boxplot.

These same pieces will be found throughout `ggplot` plotting. In later chapters we will introduce much more in relation to these plots.

```
ggplot(df, aes(x=sed, y=dep)) +
  geom_boxplot(aes(color = sed))
```

```
## Warning: Continuous x aesthetic -- did you forget aes(group=...)?
```

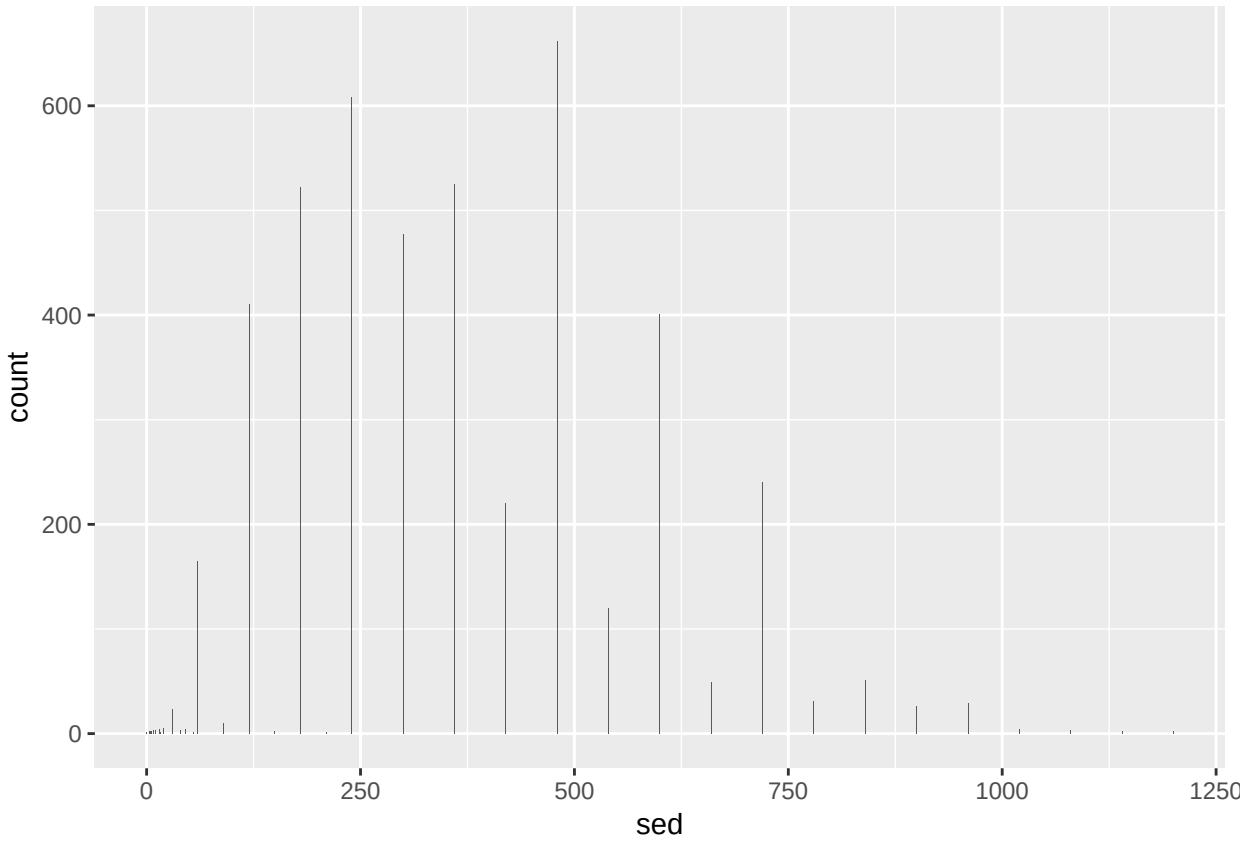
```
## Warning: Removed 18 rows containing missing values (stat_boxplot).
```



Here's a few more examples:

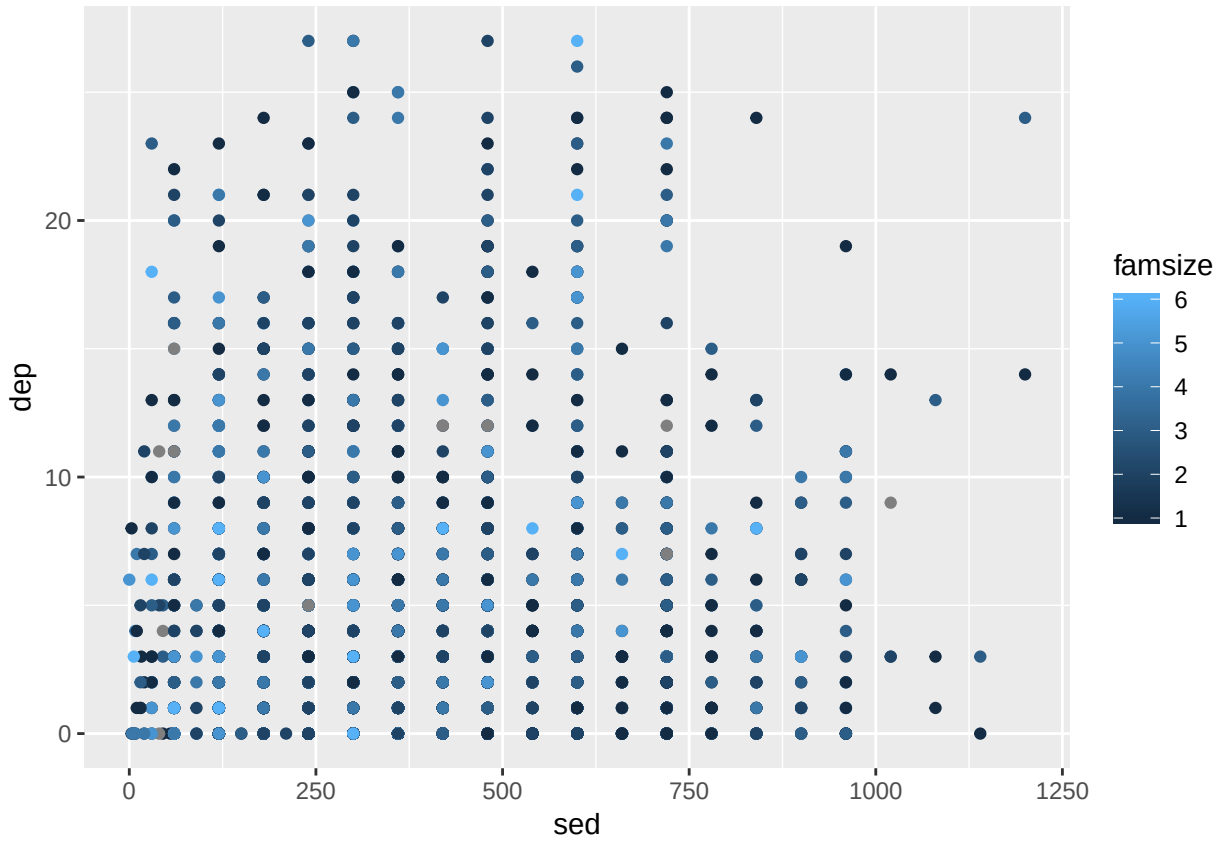
```
ggplot(df, aes(x=sed)) +
  geom_bar(stat="count")
```

```
## Warning: Removed 18 rows containing non-finite values (stat_count).
```




```
ggplot(df, aes(x=sed, y=dep)) +
  geom_point(aes(color = famsize))
```

```
## Warning: Removed 18 rows containing missing values (geom_point).
```

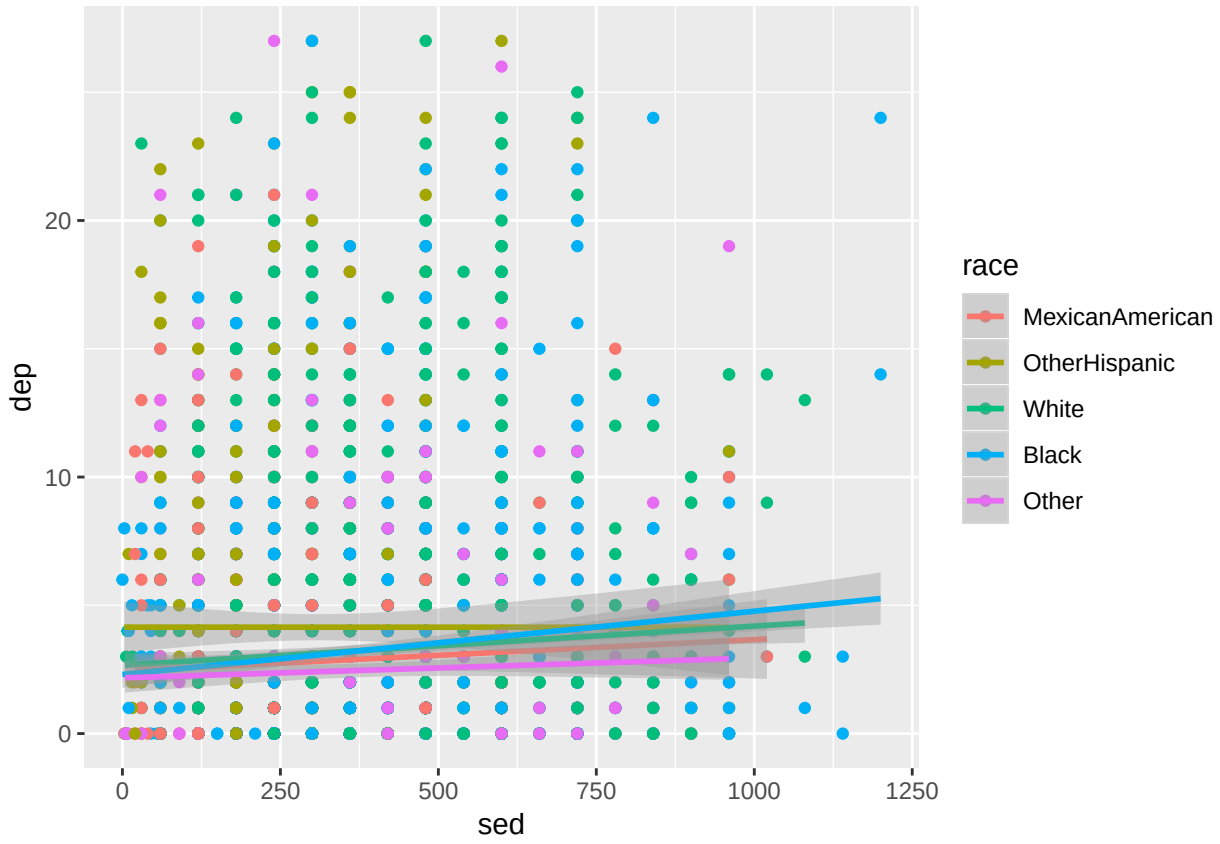


Note that the warning that says it removed a row is because we had missing values in in these variables.

```
ggplot(df, aes(x=sed,
               y=dep,
               group = race,
               color = race)) +
  geom_point() +
  geom_smooth(method = "lm")
```

```
## Warning: Removed 18 rows containing non-finite values (stat_smooth).
```

```
## Warning: Removed 18 rows containing missing values (geom_point).
```

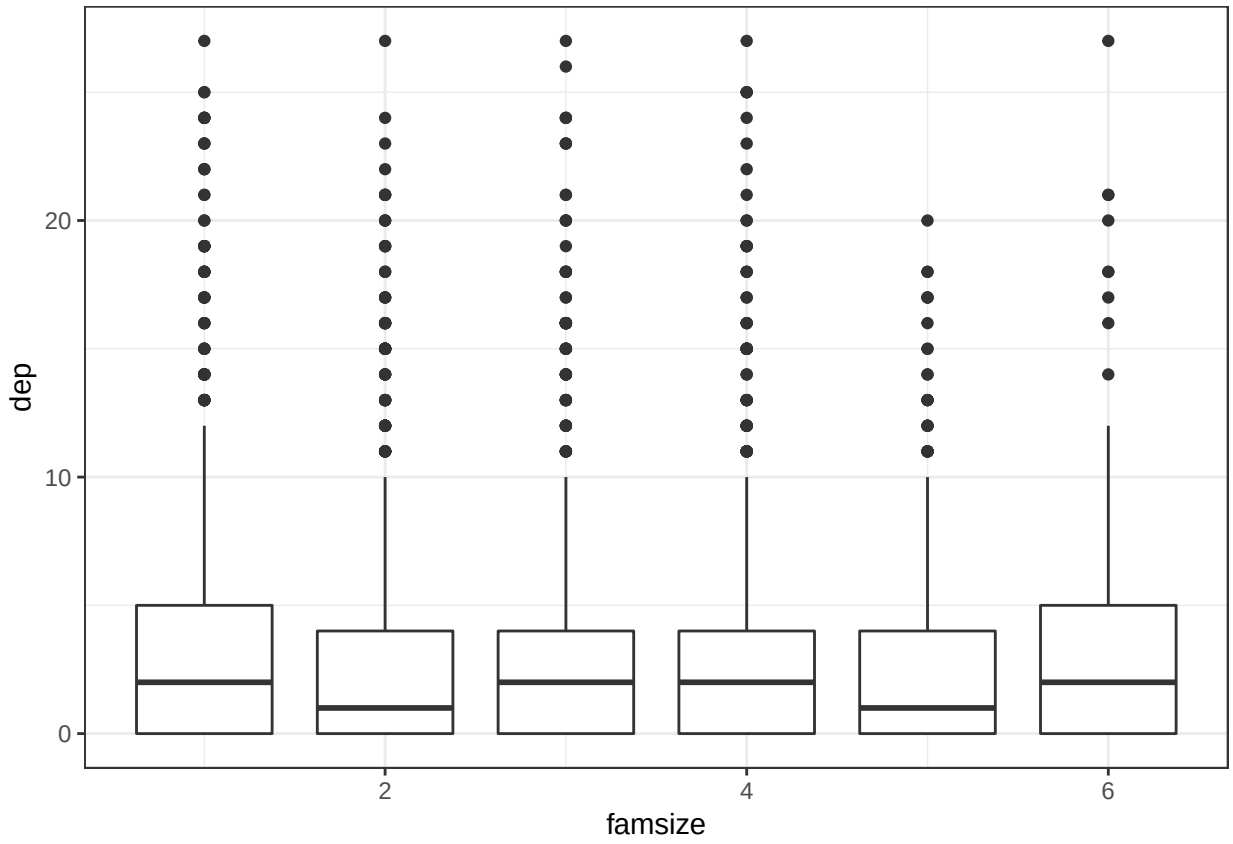


We are going to make the first one again but with some aesthetic adjustments. Notice that we just added two extra lines telling `ggplot2` how we want some things to look.[17]

```
ggplot(df, aes(x=famsize, y=dep, group = famsize)) +
  geom_boxplot(aes(color = riagendr)) +
  theme_bw() +
  scale_color_manual(values = c("dodgerblue4", "coral2"))
```

```
## Warning: Removed 178 rows containing missing values (stat_boxplot).
```

[17] This is just scratching the surface of what we can change in the plots.



The `theme_bw()` makes the background white, the `scale_color_manual()` allows us to change the colors in the plot. You can get a good idea of how many types of plots you can do by going to http://docs.ggplot2.org/current. Almost any informative plot that you need to do as a researcher is possible with `ggplot2`.

We will be using `ggplot2` extensively in the book to help understand our data and our models as well as communicate our results.

More Advanced Features of `ggplot2`

We will go through several aspects of the code that makes plotting in `R` flexible and beautiful.

1. Types of plots
2. Color schemes
3. Themes
4. Labels and titles
5. Facetting

To highlight these features we'll be using our NHANES data again; specifically, sedentary behavior, depression, asthma, family size, and race. As this is only an introduction, refer to http://docs.ggplot2.org/current/ for more information on `ggplot2`.

To begin, it needs to be understood that the first line where we actually use the `ggplot` function, will then apply to all subsequent laters (e.g., `geom_point()`). For example,

```
ggplot(df, aes(x = dep, y = sed, group = asthma))
```

means for the rest of the layers, unless we over-ride it, each will use `df` with `dep` as the x variable, `sed` as the y, and a grouping on `asthma`. So when many layers are going to use the same command put it in this so

you don't have to write the same argument many times. A common one here could be:

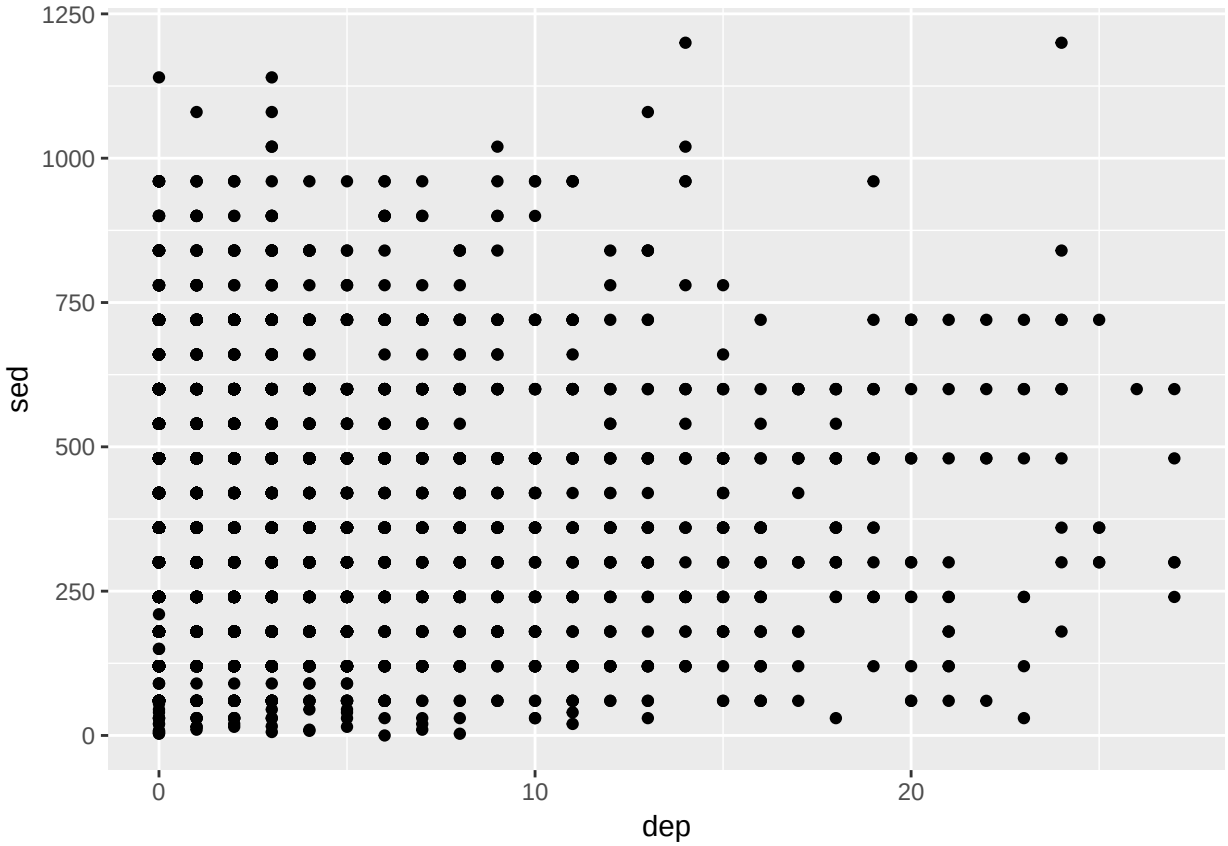
```
ggplot(df, aes(x = dep, y = sed, group = asthma, color = asthma))
```

since we often want to color by our grouping variable.

Before going forward, a nice feature of `ggplot2` allows us to use an "incomplete" plot to add on to. For example, if we have a good idea of the main structure of the plot but want to explore some changes, we can do the following:

```
p1 <- ggplot(df, aes(x = dep, y = sed, group = asthma)) +
  geom_point()
p1
```

```
## Warning: Removed 18 rows containing missing values (geom_point).
```



So now `p1` has the information for this basic, and honestly fairly uninformative, plot. We'll use this feature to build on plots that we like.

Some of our figures will also need summary data so we'll start that here as well:

```
summed_data <- df %>%
  group_by(asthma, dep2) %>%
  summarize(s_se = sd(sed, na.rm=TRUE)/sqrt(n()),
            sed  = mean(sed, na.rm=TRUE),
            N    = n())
```

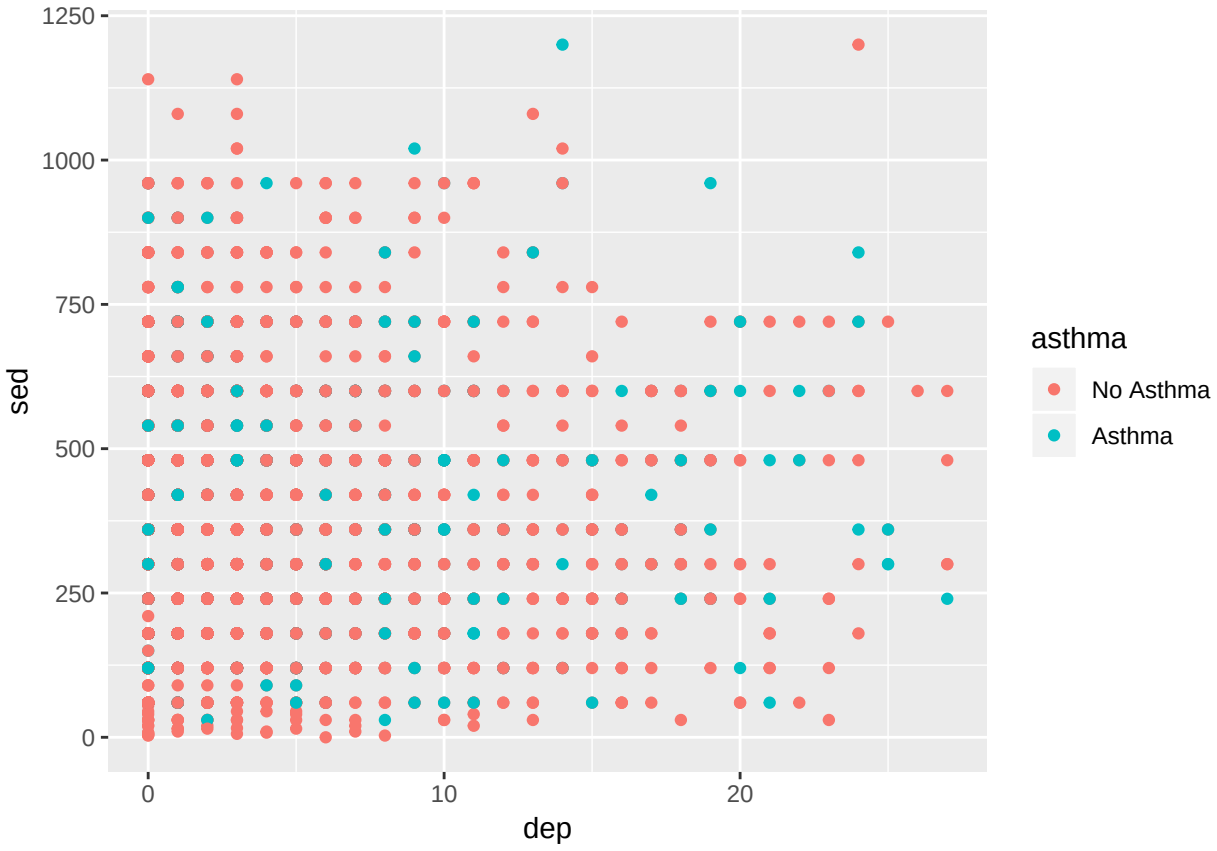
As you hopefully recognize a bit, we are summarizing the time spent being sedentary by both asthma and the dichotomous depression variables. If it doesn't make sense at first, read it line by line to see what I did. This will be useful for several types of plots.

Types of Plots

Scatterplots

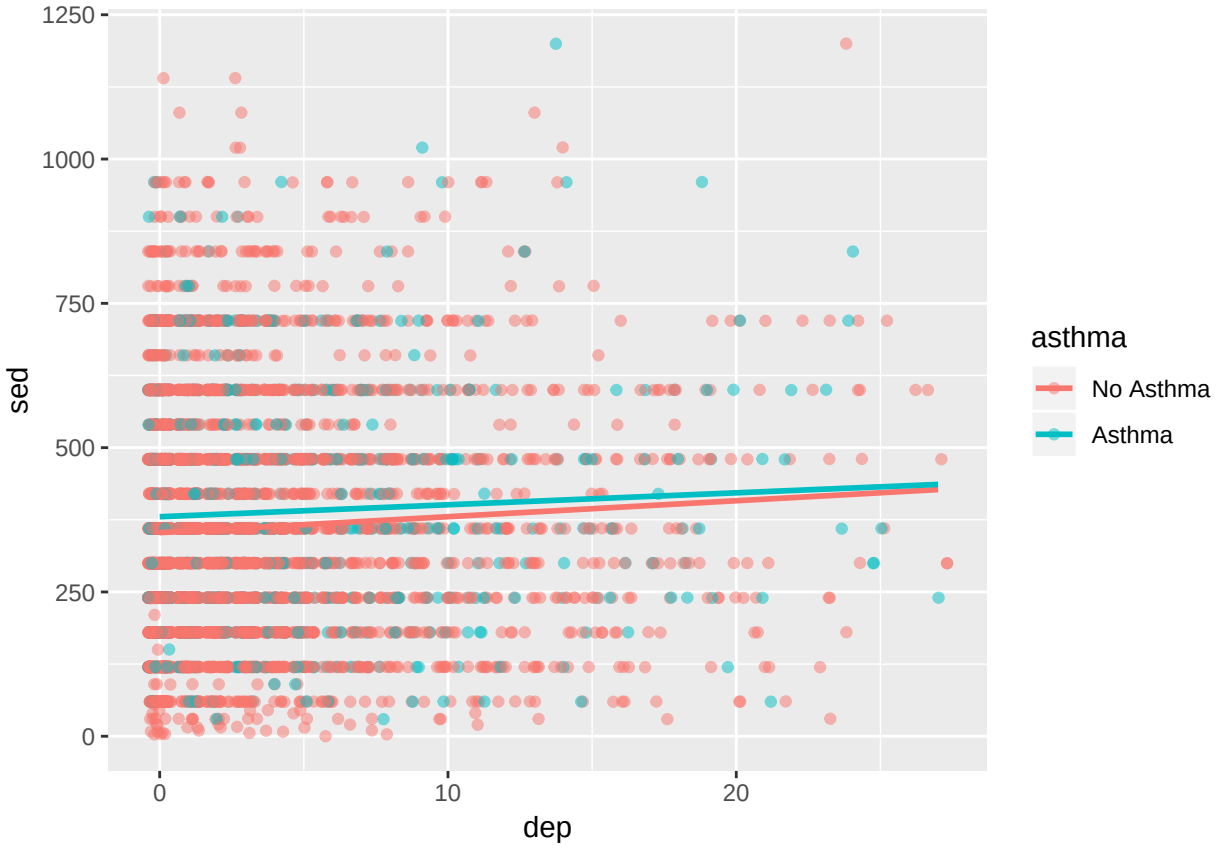
We'll start with a scatterplot–one of the most simple yet informative plots.

```
ggplot(df, aes(x = dep, y = sed, group = asthma)) +
  geom_point(aes(color = asthma))
```



It's not amazing. There looks to be a lot of overlap of the points. Also, it would be nice to know general trend lines for each group. Below, `alpha` refers to how transparent the points are, `method = "lm"` refers to how the line should be fit, and `se=FALSE` tells it not to include error ribbons.

```
ggplot(df, aes(x = dep, y = sed, group = asthma)) +
  geom_jitter(aes(color = asthma), alpha = .5) +
  geom_smooth(aes(color = asthma), method = "lm", se=FALSE)
```

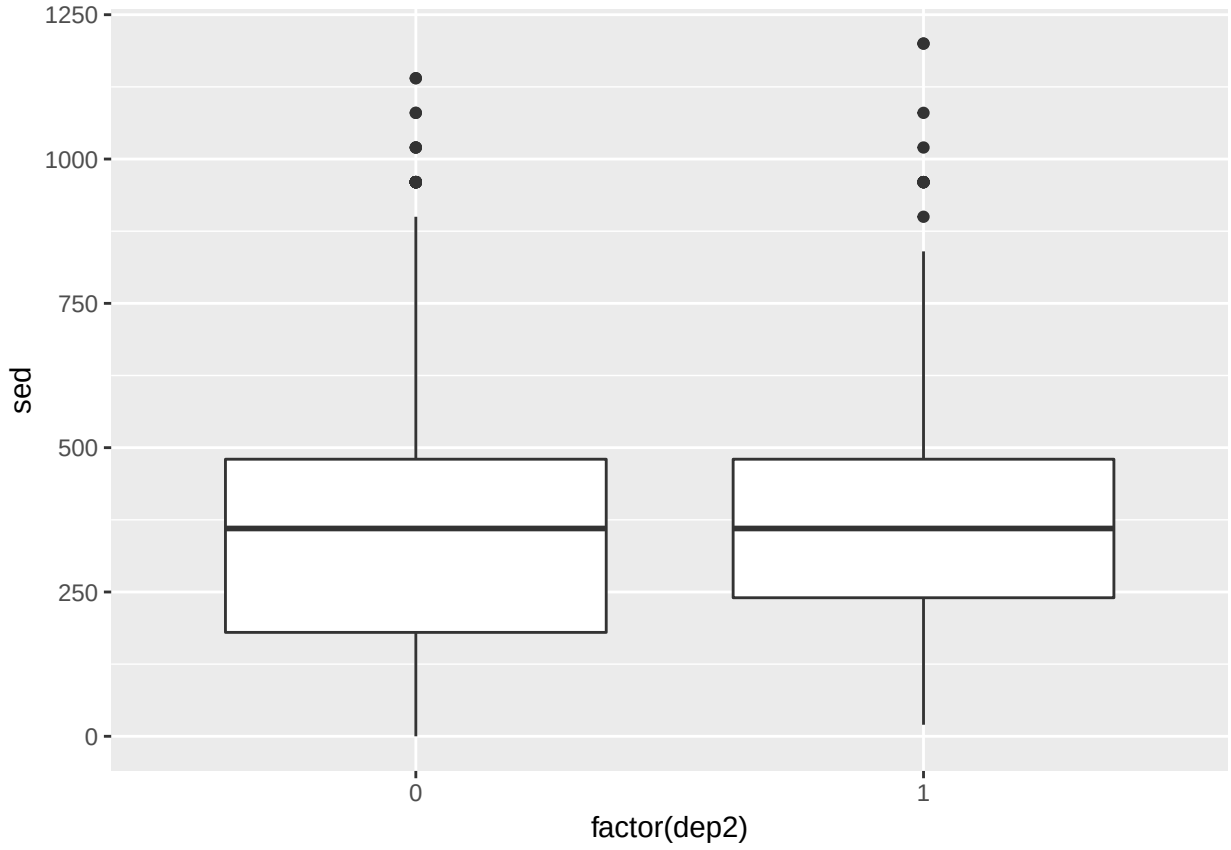


It's getting better but we could still use some more features. We'll come back to this in the next sections.

Boxplots

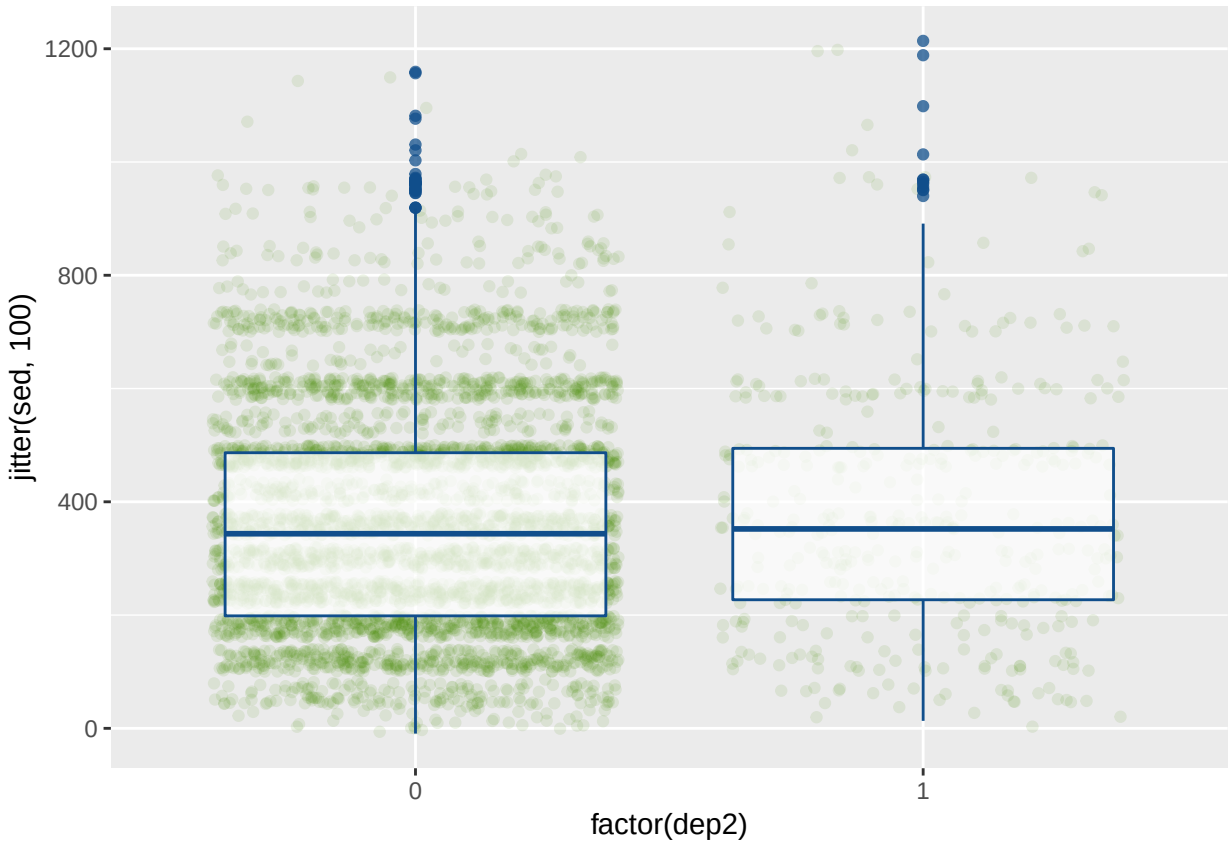
Box plots are great ways to assess the variability in your data. Below, we create a boxplot but change `p1`'s x variable so that it is the factor version of depression.

```
ggplot(df, aes(x = factor(dep2), y = sed)) +
  geom_boxplot()
```



This plot is, at best, mediocre. But there's more we can do.

```
ggplot(df, aes(x = factor(dep2), y = jitter(sed, 100))) +
  geom_jitter(alpha = .1, color = "chartreuse4") +
  geom_boxplot(alpha = .75, color = "dodgerblue4")
```

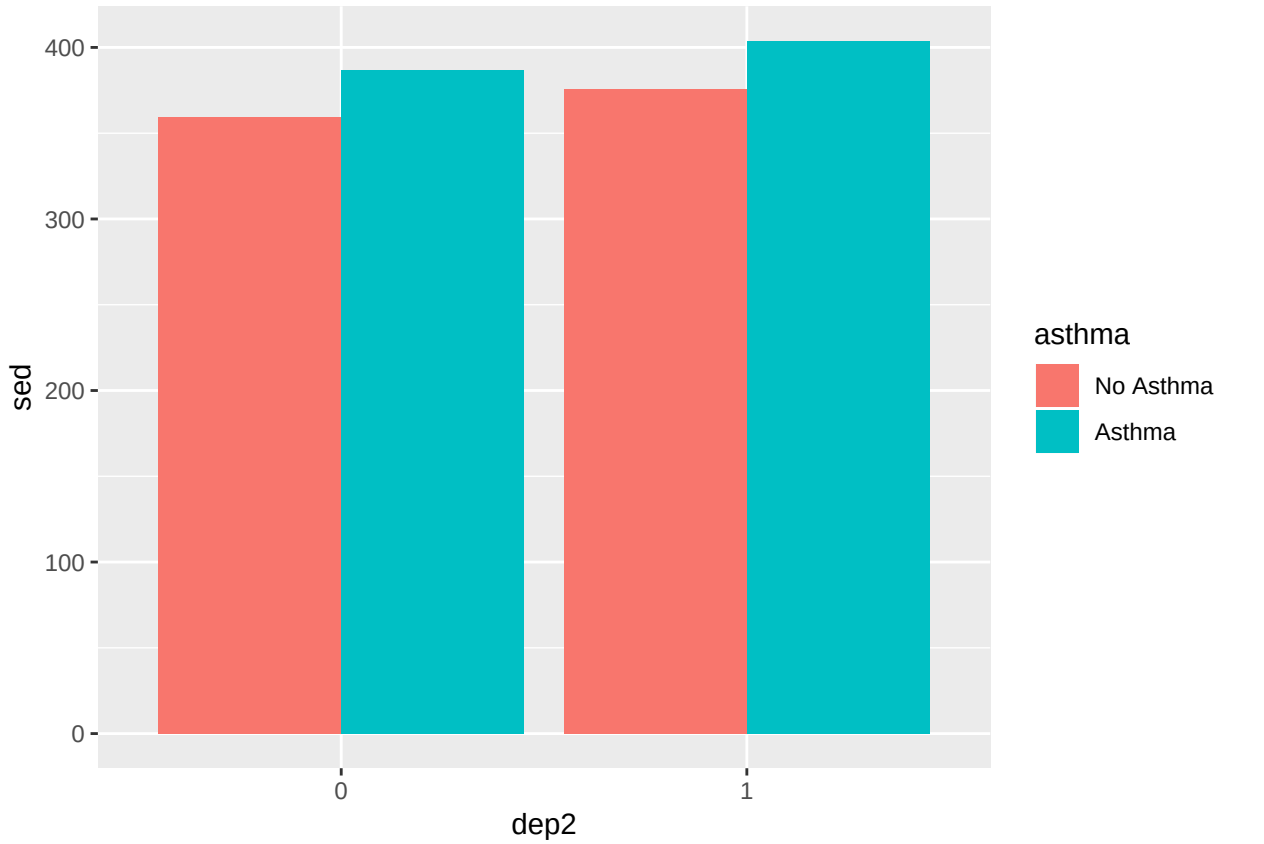


This now provides the (jittered) raw data points as well to hightlight the noise and the number of observations in each group.

Bar Plots

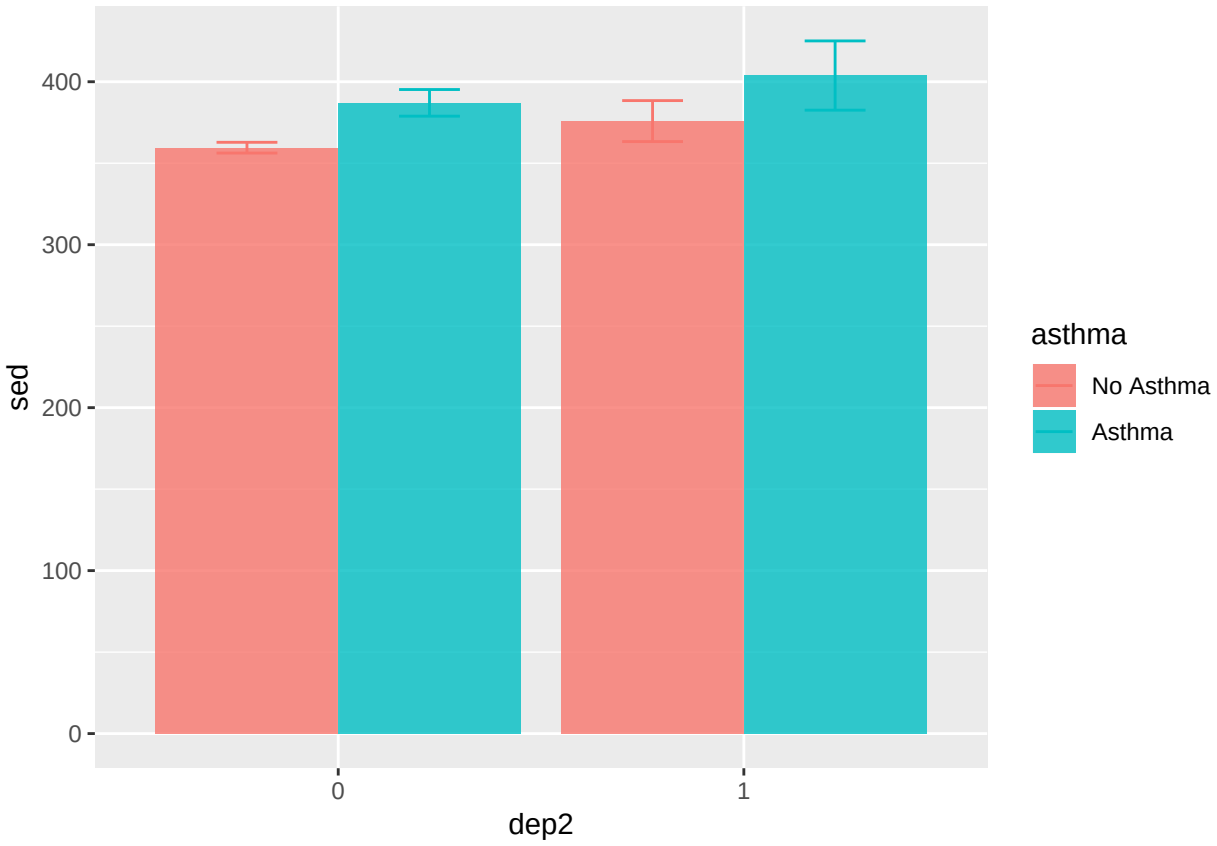
Bar plots are great ways to look at means and standard deviations for groups.

```
ggplot(summed_data, aes(x = dep2, y = sed, group = asthma)) +
  geom_bar(aes(fill = asthma), stat = "identity", position = "dodge")
```



We used `stat = "identity"` to make it based on the mean (default is `count`), and `position = "dodge"` makes it so the bars are next to each other as opposed to stacked. Let's also add error bars.

```
p = position_dodge(width = .9)
ggplot(summed_data, aes(x = dep2, y = sed, group = asthma)) +
  geom_bar(aes(fill = asthma),
           stat = "identity",
           position = p,
           alpha = .8) +
  geom_errorbar(aes(ymin = sed - s_se, ymax = sed + s_se,
                    color = asthma),
                position = p,
                width = .3)
```



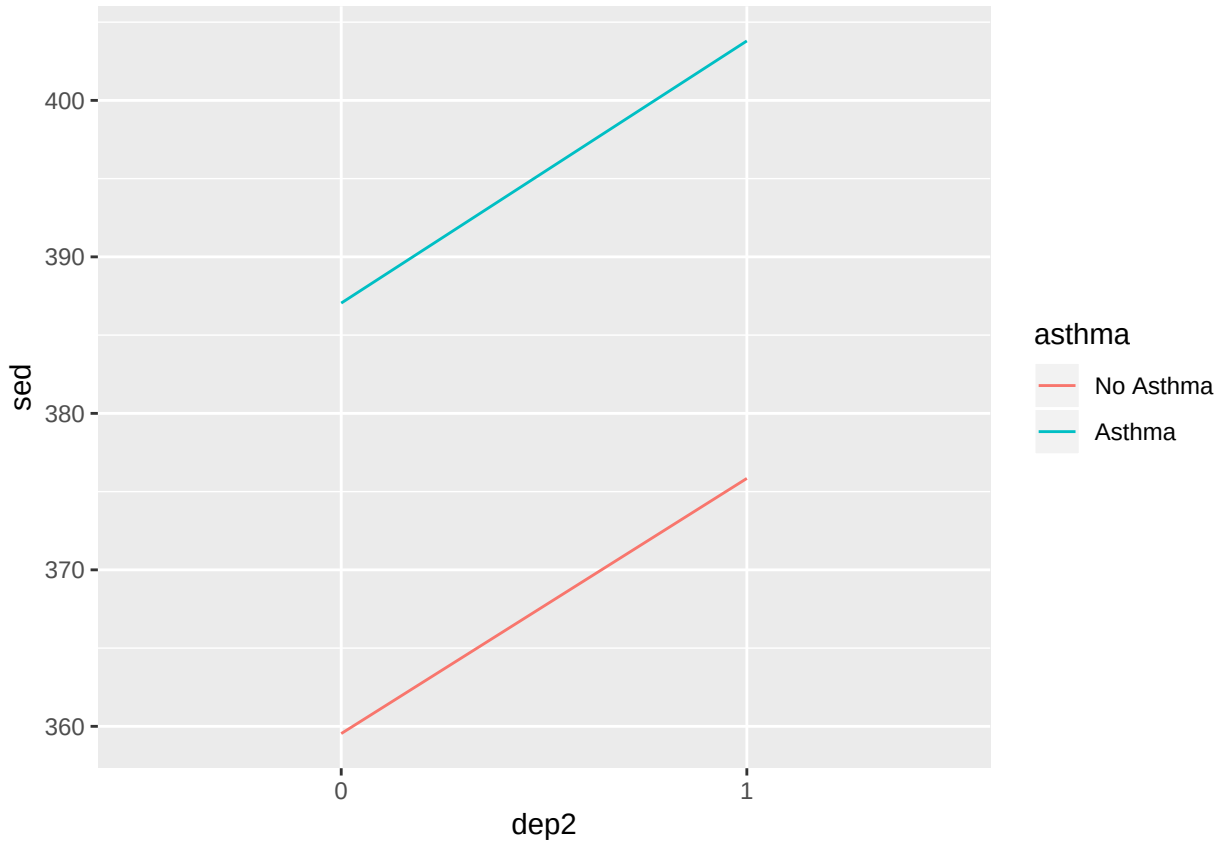
There's a lot in there but much of it is what you've seen before. For example, we use `alpha` in the `geom_bar()` to tell it to be slightly transparent so we can see the error bars better. We used the `position_dodge()` function to specify exactly how much dodge we wanted. In this way, we are able to line up the error bars and the bars. If we just use `position = "dodge"` we have less flexibility and control.

Much more can be done to clean this up, which we'll show in later sections.

Line Plots

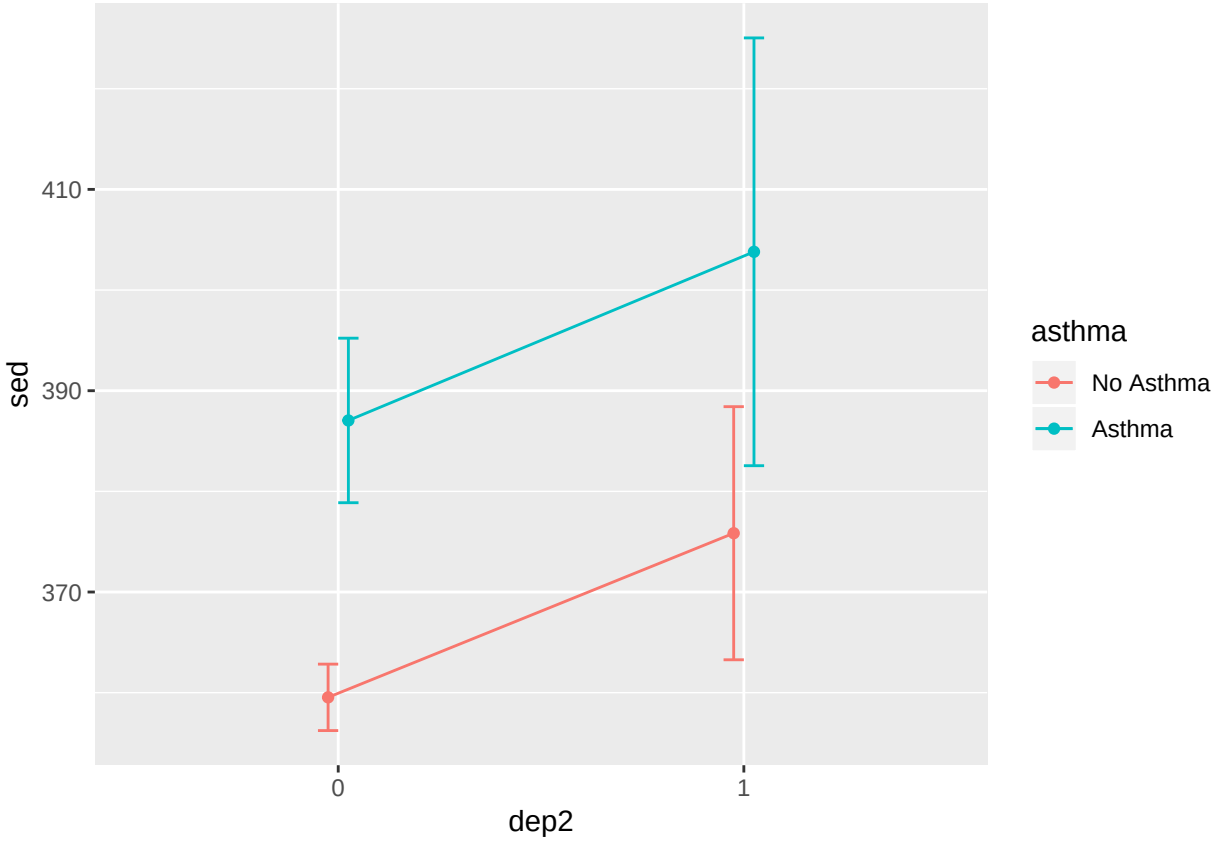
Line plots are particularly good at showing trends and relationships. Below we we use it to highlight the relationship between depression, sedentary behavior, and asthma.

```
ggplot(summed_data, aes(x = dep2, y = sed, group = asthma)) +
  geom_line(aes(color = asthma))
```



Good start, but let's add some features.

```
pos = position_dodge(width = .1)
ggplot(summed_data, aes(x = dep2, y = sed, group = asthma, color = asthma)) +
  geom_line(position = pos) +
  geom_point(position = pos) +
  geom_errorbar(aes(ymin = sed - s_se, ymax = sed + s_se),
                width = .1,
                position = pos)
```

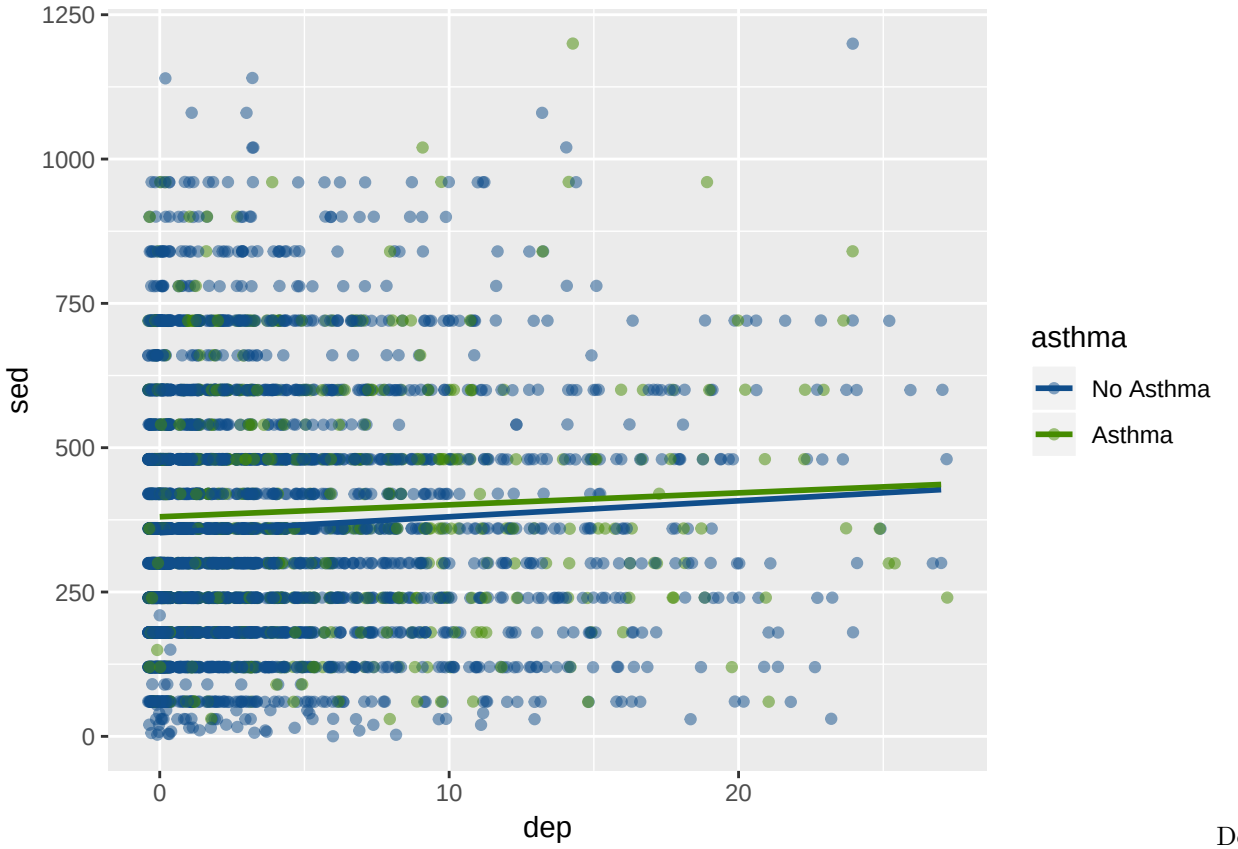


That looks a bit better. From here, let's go on to color schemes to make the plots a bit better.

Color Schemes

We'll start by using the scatterplot we made above but we will change the colors a bit using `scale_color_manual()`.

```
ggplot(df, aes(x = dep, y = sed, group = asthma)) +
  geom_jitter(aes(color = asthma), alpha = .5) +
  geom_smooth(aes(color = asthma), method = "lm", se=FALSE) +
  scale_color_manual(values = c("dodgerblue4", "chartreuse4"))
```



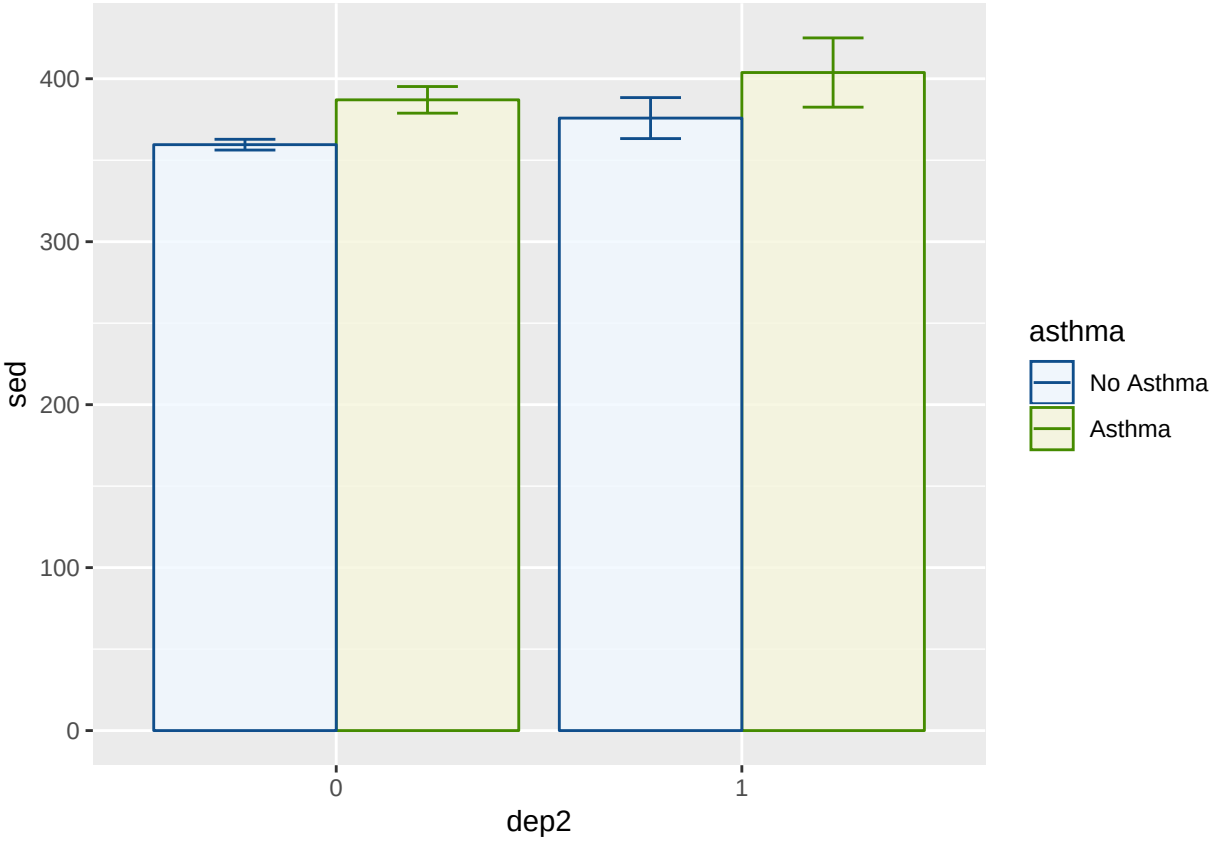
Depending on your personal taste, you can adjust it with any color. On my blog, I've posted the colors available in R (there are many).

> Advice: Don't get too lost in selecting colors but it can add a nice touch to any plot. The nuances of plot design can be invigorating but also time consuming to be smart about how long you spend using it.

Next, let's adjust the bar plot. We will also add some colors here, but we will differentiate between "color" and "fill".

1. Fill fills in the object with color. This is useful for things that are more than simply a line or a dot.
2. Color colors the object. This outlines those items that can also be filled and colors lines and dots.

```
p = position_dodge(width = .9)
ggplot(summed_data, aes(x = dep2, y = sed, group = asthma)) +
  geom_bar(aes(fill = asthma, color = asthma),
           stat = "identity",
           position = p,
           alpha = .8) +
  geom_errorbar(aes(ymin = sed - s_se, ymax = sed + s_se,
                    color = asthma),
                position = p,
                width = .3) +
  scale_color_manual(values = c("dodgerblue4", "chartreuse4")) +   ## controls the color of the error b
  scale_fill_manual(values = c("aliceblue", "beige"))
```



Just so you are aware:

- aliceblue is a lightblue
- beige is a light green
- dodgerblue4 is a dark blue
- chartreuse4 is a dark green

So the `fill` colors are light and the `color` colors are dark in this example. You, of course, can do whatever you want color-wise. I'm a fan of this style though so we will keep it for now.

These same functions can be used on the other plots as well. Feel free to give them a try. As for the book, we'll move on to the next section: Themes.

Themes

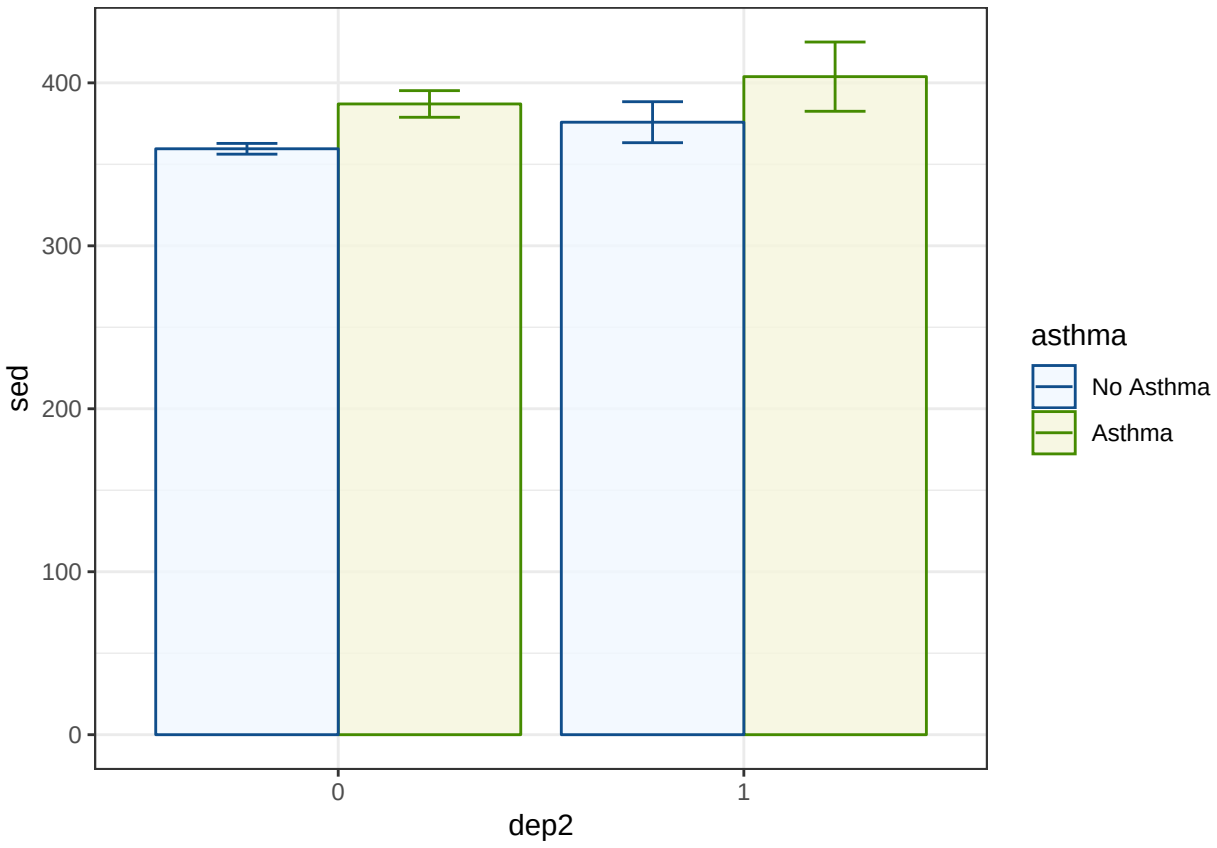
Using the plot we just made–the bar plot–we will show how theme options work. There are several built in themes that change many aspects of the plot (e.g., `theme_bw()`, `theme_classic()`, `theme_minimal()`). There are many more if you download the `ggthemes` package. Fairly simply you can create plots similar to those in newspapers and magazines.

First, we are going to save the plot to simply show the different theming options.

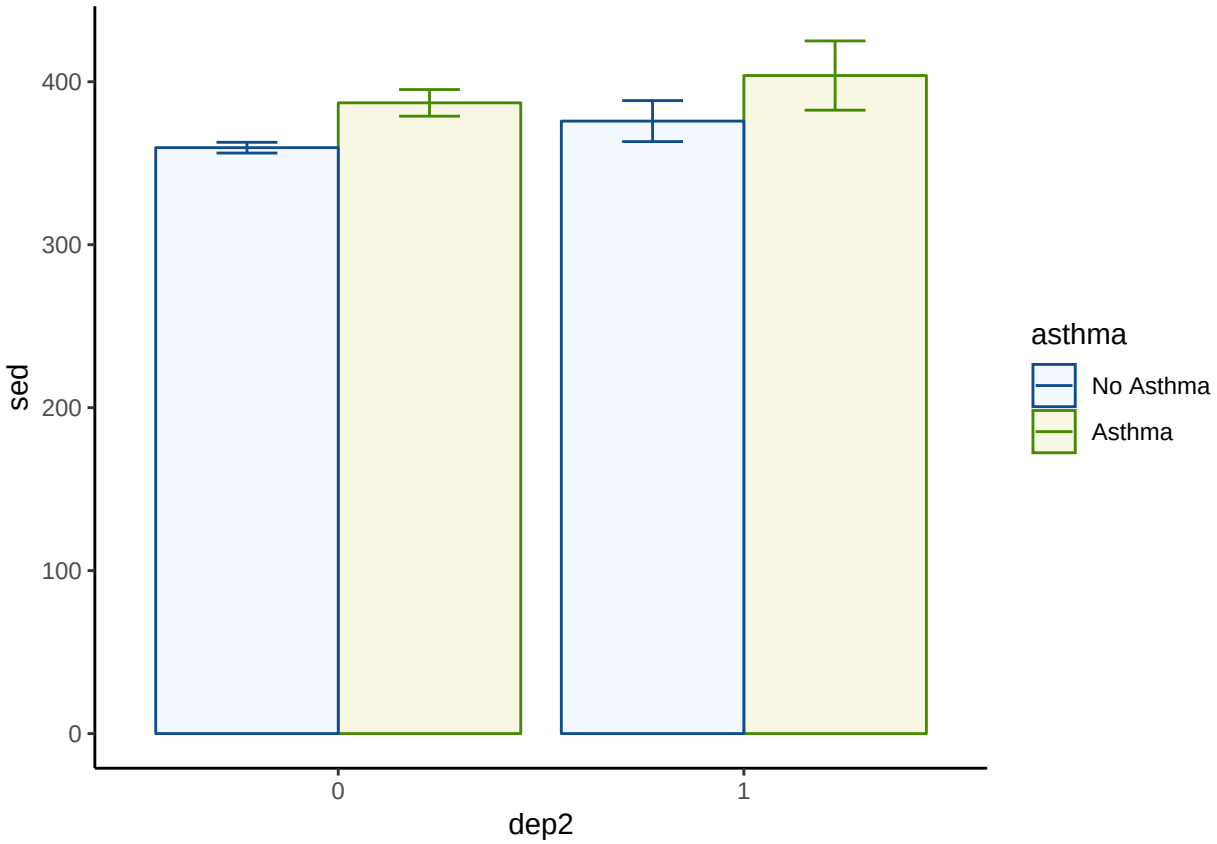
```
p = position_dodge(width = .9)
p1 = ggplot(summed_data, aes(x = dep2, y = sed, group = asthma)) +
  geom_bar(aes(fill = asthma, color = asthma),
           stat = "identity",
           position = p,
           alpha = .8) +
  geom_errorbar(aes(ymin = sed - s_se, ymax = sed + s_se,
```

```
                    color = asthma),
                position = p,
                width = .3) +
  scale_color_manual(values = c("dodgerblue4", "chartreuse4")) +   ## controls the color of the error b
  scale_fill_manual(values = c("aliceblue", "beige"))
```

```
p1 +
  theme_bw()
```

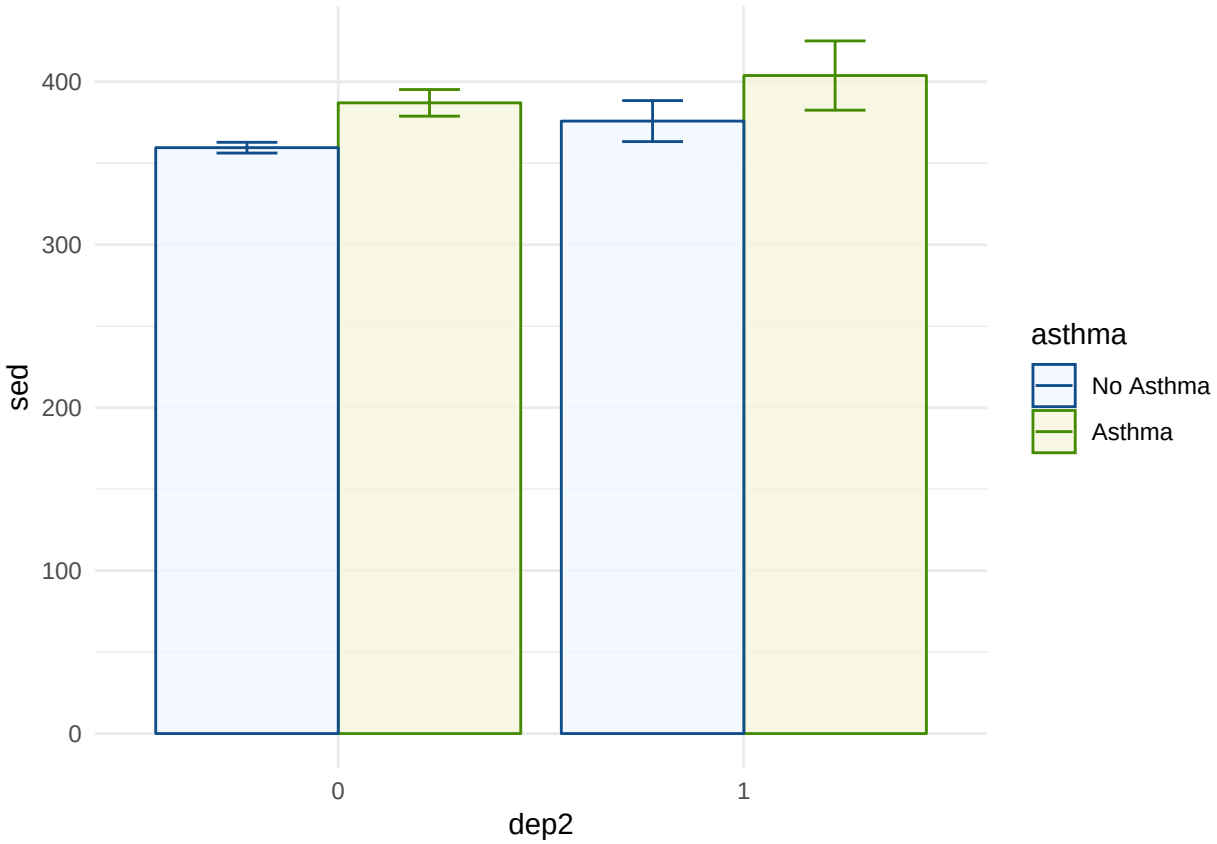



```
p1 +
  theme_classic()
```



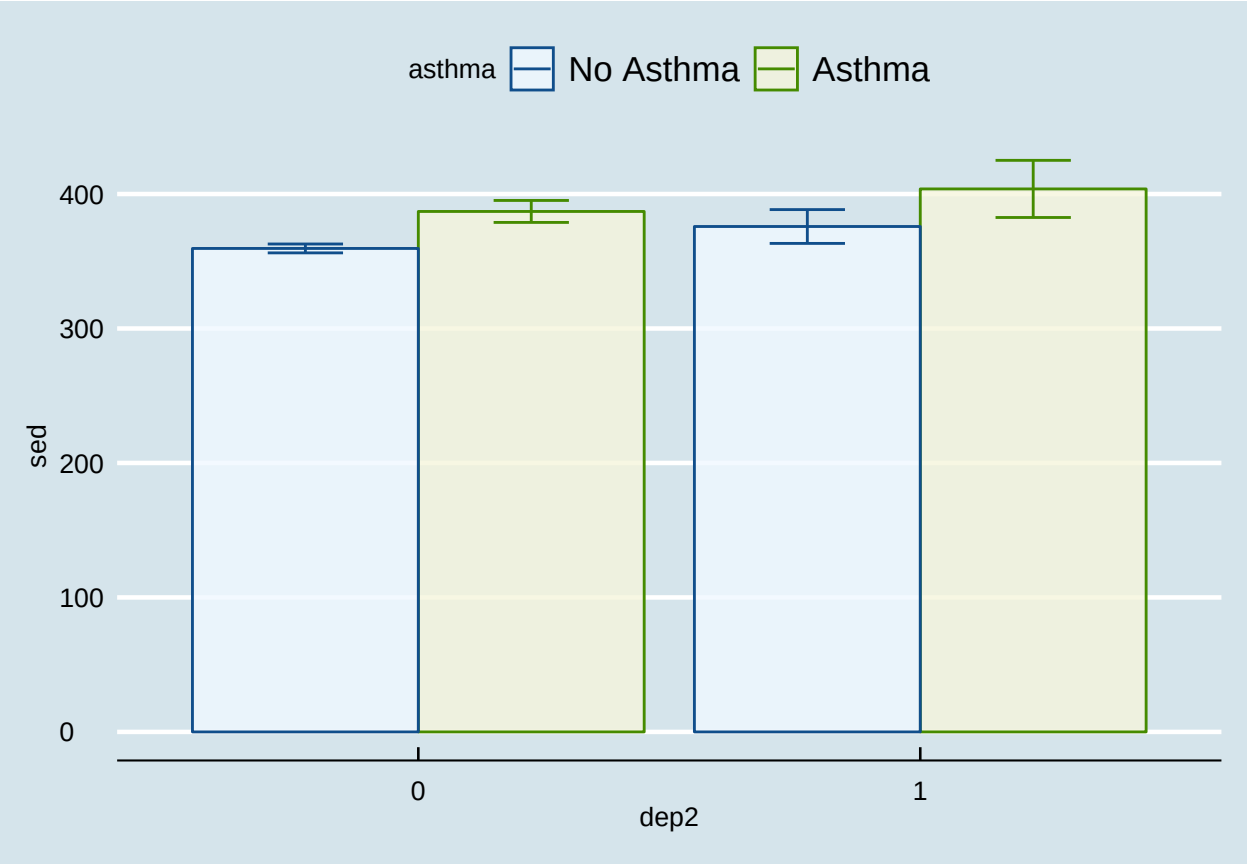

```
p1 +
  theme_minimal()
```

Theme Minimal



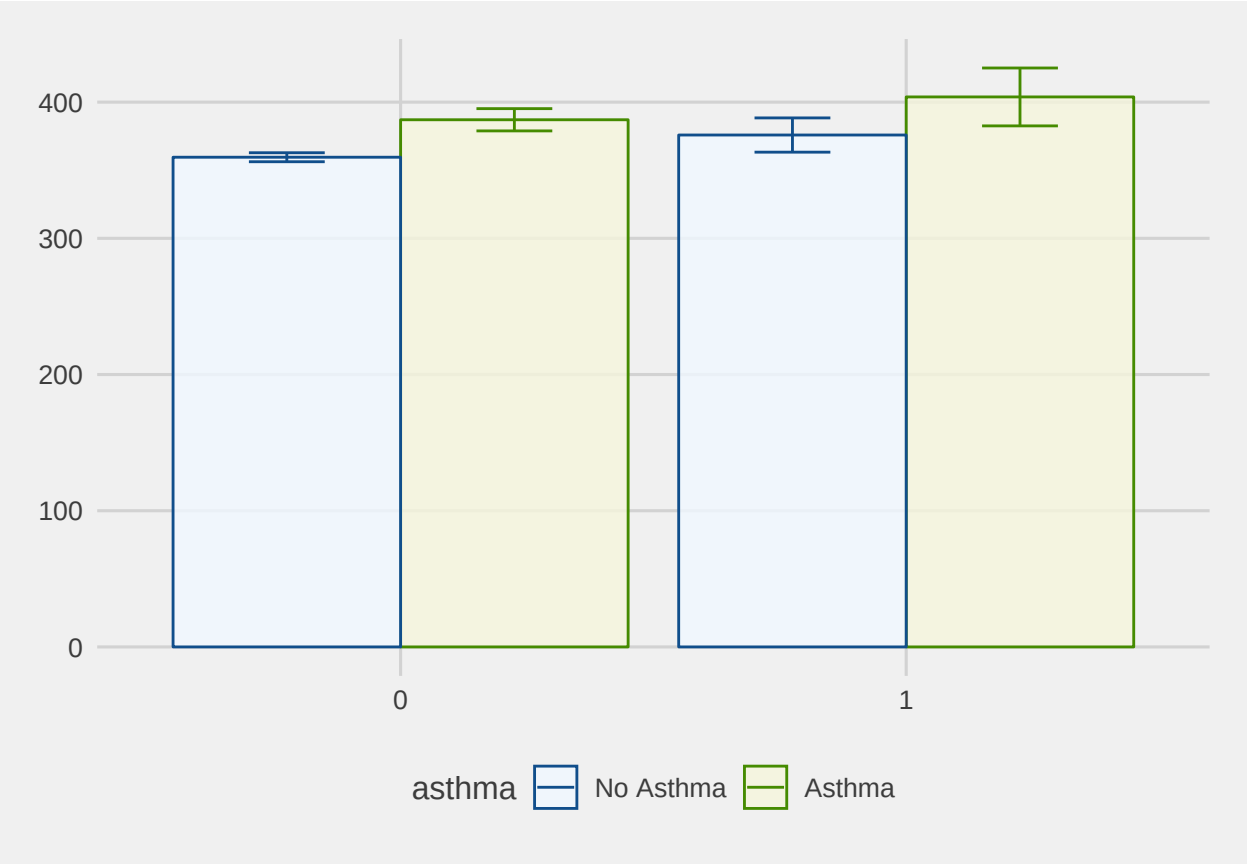

```
library(ggthemes)
p1 +
  theme_economist()
```

Theme Economist (from ggthemes)



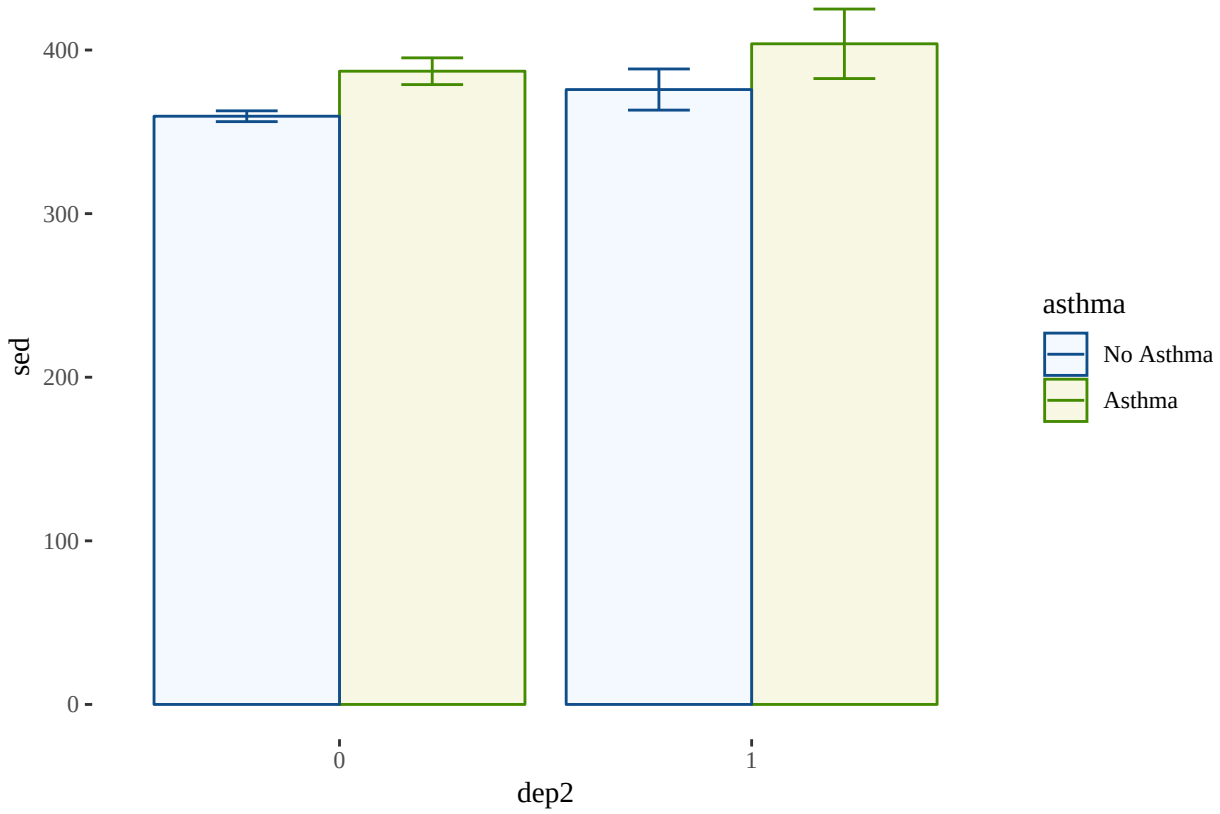

```
p1 +
  theme_fivethirtyeight()
```

Theme FiveThirtyEight (from `ggthemes`)



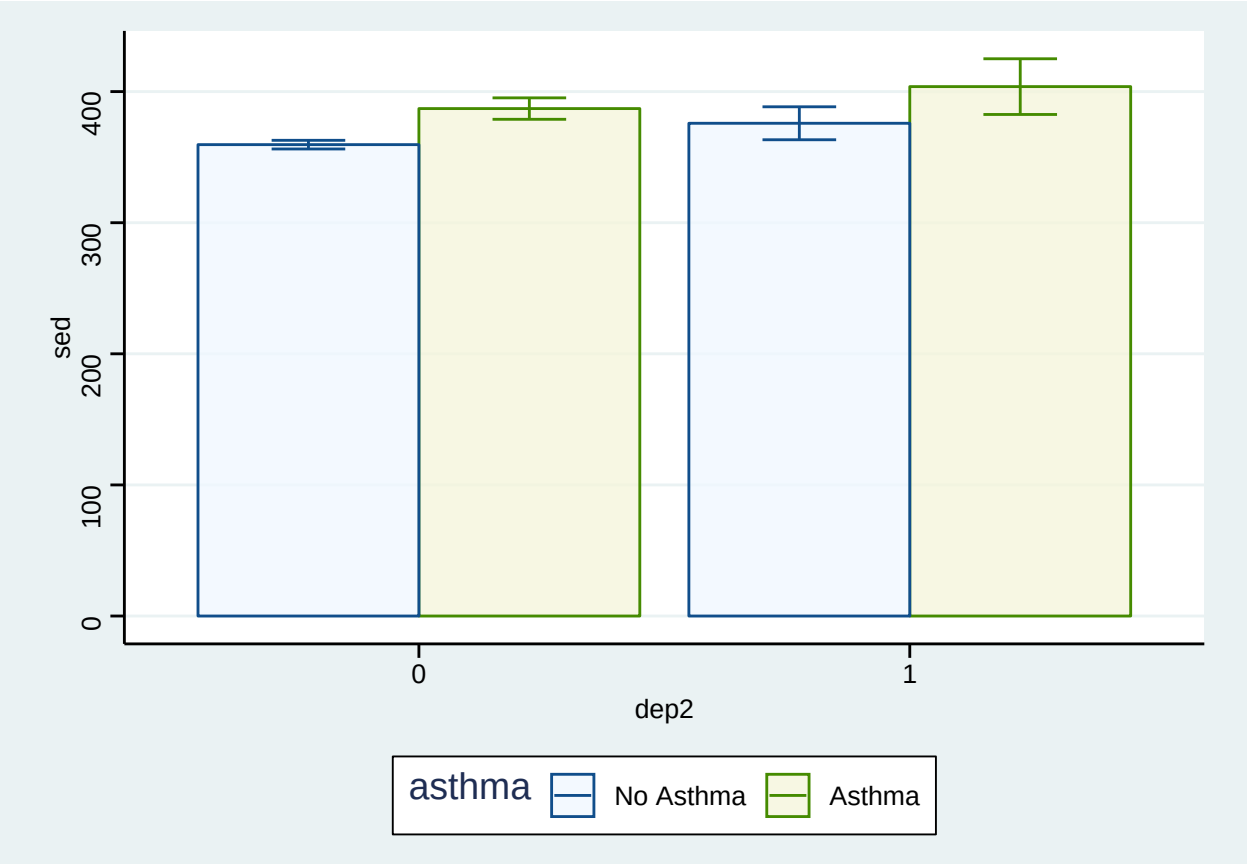

```
p1 +
  theme_tufte()
```

Theme Tufte (from ggthemes)




```
p1 +
  theme_stata()
```

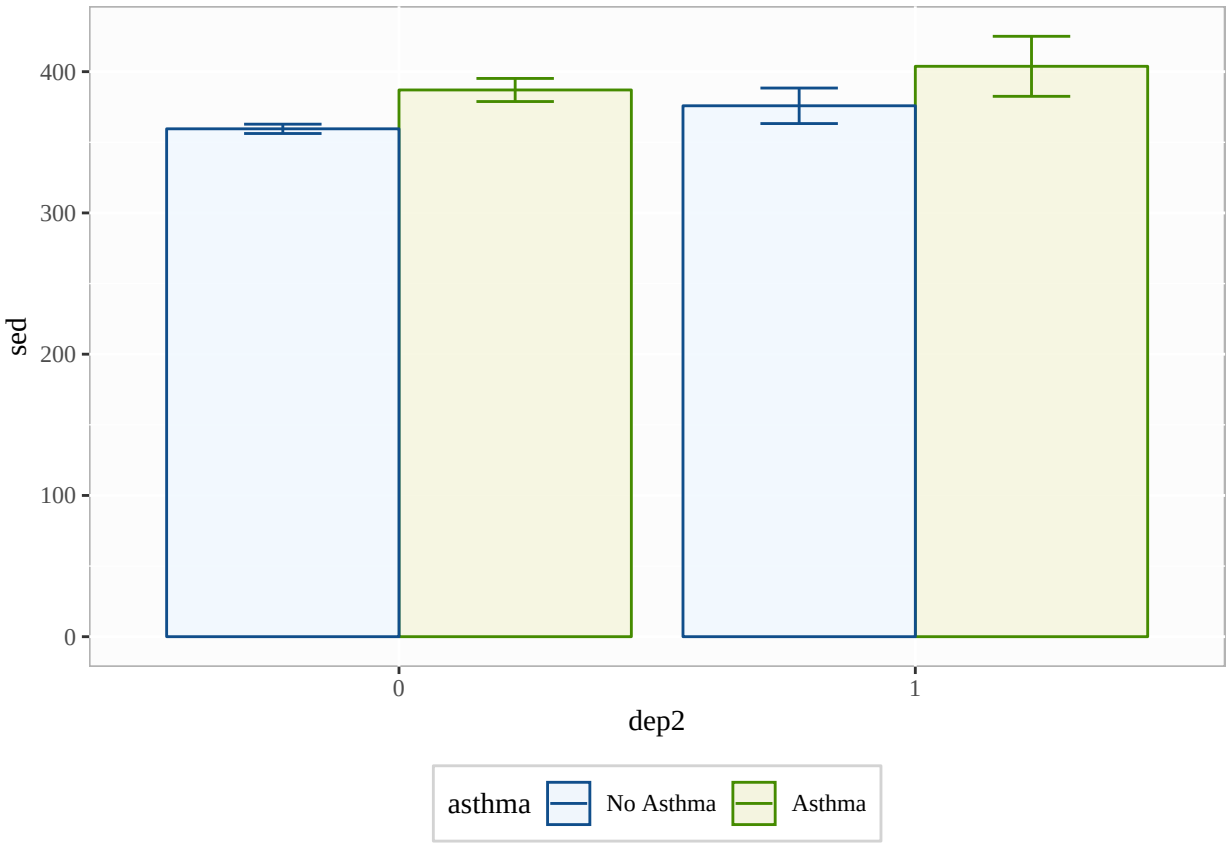
Theme Stata (from ggthemes)



Your Own Theme

There are many more but you get the idea. In addition to the built in themes, you can use the `theme()` function and make your own adjustments. There are *many* options so we will just introduce the idea.

```
p1 +
  theme(legend.position = "bottom",  ## puts legend at the bottom of figure
        legend.background = element_rect(color = "lightgrey"),  ## outlines legend
        panel.background = element_rect(fill = "grey99",    ## fills the plot with a very light grey
                                        color = "grey70"),  ## light border around plot
        text = element_text(family = "Times"))      ## all text in plot is now Times
```

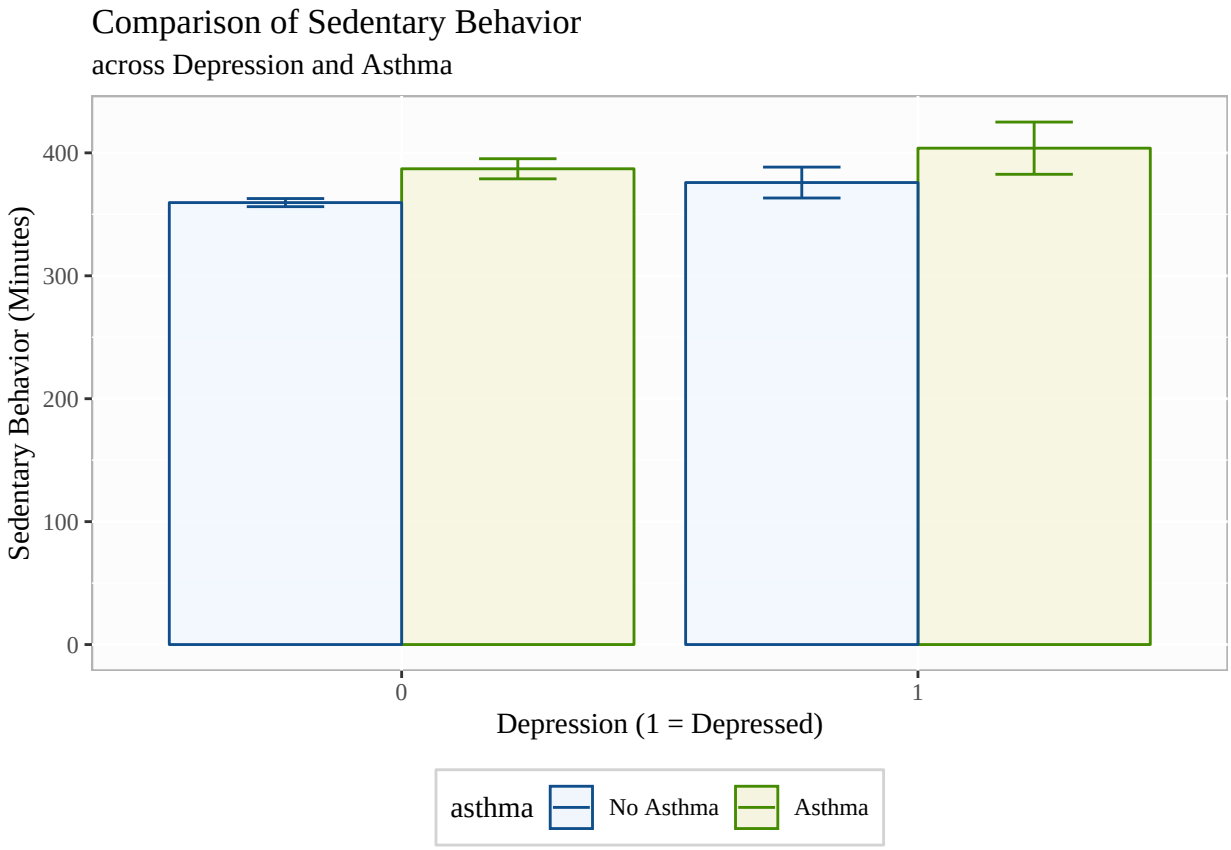


There are many more options but essentially if there is something you want to change, you probably can.

Labels and Titles

Using our last plot, we will also want to add good labels and/or titles.

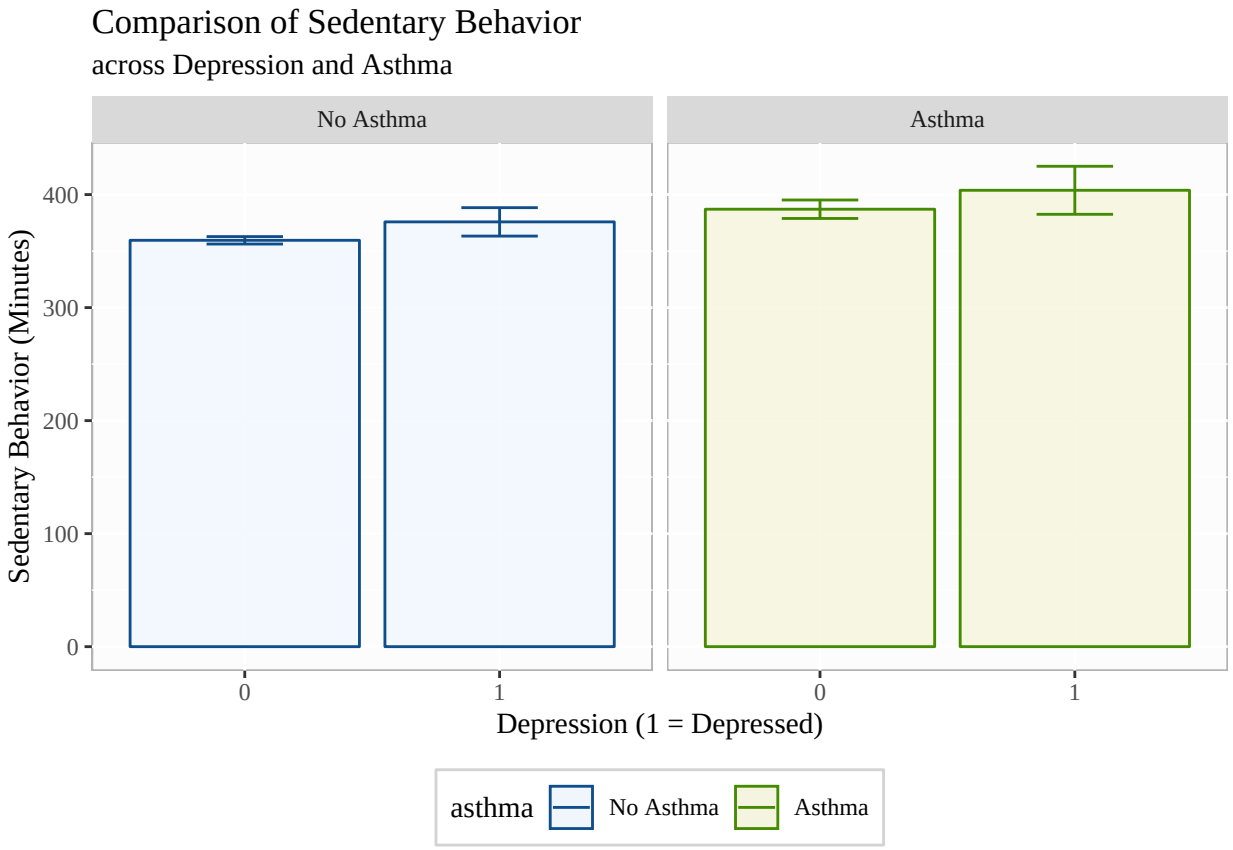
```
p1 +
  theme(legend.position = "bottom",
        legend.background = element_rect(color = "lightgrey"),
        panel.background = element_rect(fill = "grey99",
                                        color = "grey70"),
        text = element_text(family = "Times")) +
  labs(y = "Sedentary Behavior (Minutes)",
       x = "Depression (1 = Depressed)",
       title = "Comparison of Sedentary Behavior",
       subtitle = "across Depression and Asthma")
```



Facetting

Facetting is very useful when trying to compare more than three variables at a time or you cannot use color or shading. It is often useful and beautiful. Facetting splits the data based on some grouping variable (e.g., asthma) to highlight differences in the relationship.

```
p1 +
  theme(legend.position = "bottom",
        legend.background = element_rect(color = "lightgrey"),
        panel.background = element_rect(fill = "grey99",
                                        color = "grey70"),
        text = element_text(family = "Times")) +
  labs(y = "Sedentary Behavior (Minutes)",
       x = "Depression (1 = Depressed)",
       title = "Comparison of Sedentary Behavior",
       subtitle = "across Depression and Asthma") +
  facet_grid(~asthma)
```



You can facet by more than one variable and it will create separate panels for each combination of the facetting variables.

Apply It

This link contains a folder complete with an Rstudio project file, an RMarkdown file, and a few data files. Download it and unzip it to do the following steps.

Step 1

Open the `Chapter3.Rproj` file. This will open up RStudio for you.

Step 2

Once RStudio has started, in the panel on the lower-right, there is a `Files` tab. Click on that to see the project folder. You should see the data files and the `Chapter3.Rmd` file. Click on the `Chapter3.Rmd` file to open it. In this file, import the data, create a descriptive table with `table1()` and three different types of visualizations (e.g., boxplot, histogram, scatterplot).

Once that code is in the file, click the `knit` button. This will create an HTML file with the code and output knitted together into one nice document. This can be read into any browser and can be used to show your work in a clean document.

Conclusions

This was a quick demonstration of plotting with `ggplot2`. There is so much more you can do. However, in the end, exploring and communicating the data through plots is simply something you need to practice.

With time, you can *a priori* picture the types of plots that will highlight things in your data, the ways you can adjust it, and how you need to manipulate your data to make it plot ready. Be patient and have fun trying things. In my experience, almost anytime I think, "Can R do this?", it can, so try to do cool stuff and you'll probably find that you can.

Chapter 4: Basic Analyses

"The goal is to turn data into information, and information into insight." — Carly Fiorina

In this chapter we are going to demonstrate basic modeling in `R`. Lucky for us, `R` is built for these analyses. It is actually quite straight-forward to run these types of models and analyze the output. Not only that, but there are simple ways to compare models.

We will go through the **ANOVA** family of analyses, the **linear regression** models, and look at **diagnostics** of each.

ANOVA

ANOVA stands for **an**alysis **o**f **va**riance. It is a family of methods (e.g. ANCOVA, MANOVA) that all share the fact that they compare a continuous dependent variable by a grouping factor variable (and may have multiple outcomes or other covariates).

$$Y_i = \alpha_0 + \alpha_1 \text{Group}_i + e_i$$

Since the groups are compared using "effect coding," the α_0 is the grand mean and each of the group level means are compared to it.

To run an ANOVA model, you can simply use the `aov` function. In the example below, we are analyzing whether family size (although not fully continuous it is still useful for the example) differs by race.

```
df$race <- factor(df$ridreth1,
                  labels=c("MexicanAmerican", "OtherHispanic", "White", "Black", "Other"))
df$famsize <- as.numeric(df$dmdfmsiz)

fit <- aov(famsize ~ race, df)
anova(fit)
```

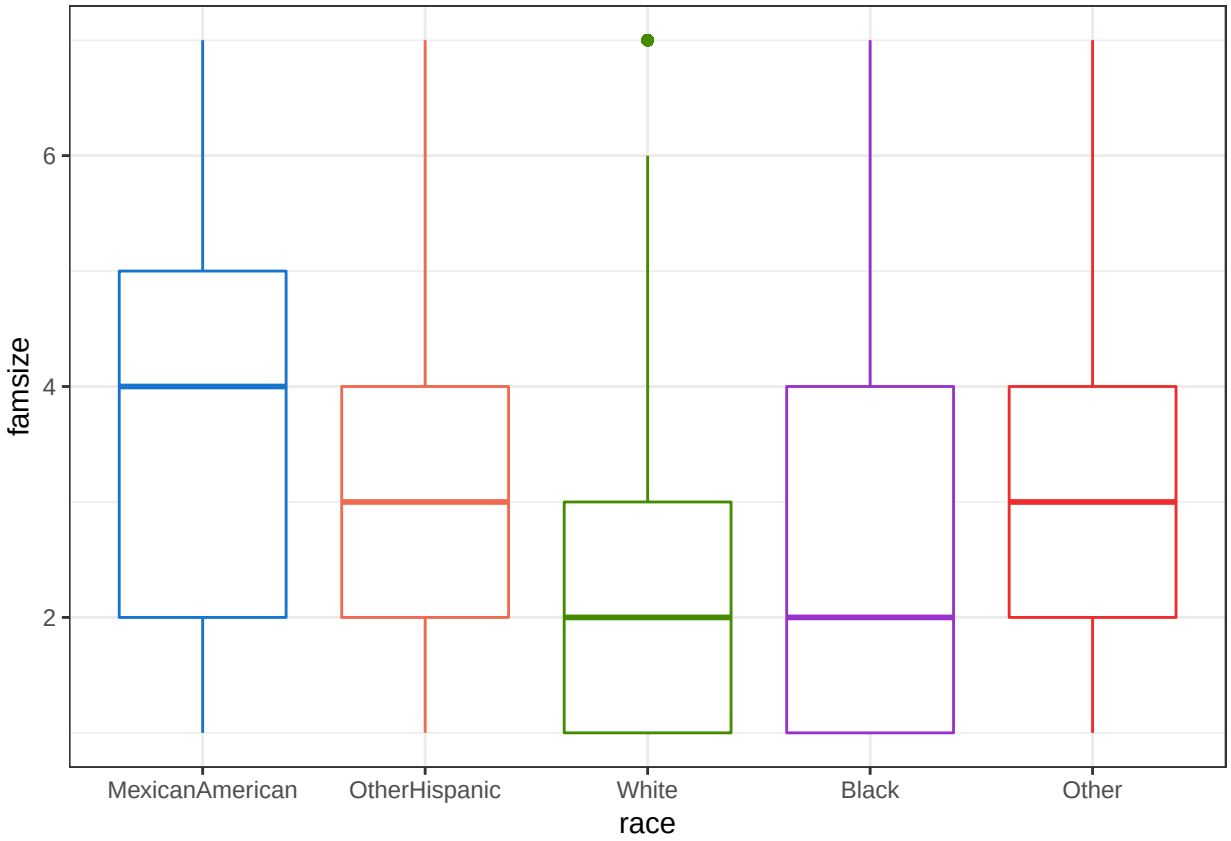
```
## Analysis of Variance Table
##
## Response: famsize
##              Df Sum Sq Mean Sq F value Pr(>F)
## race          4    541   135.3    51.4 <2e-16 ***
## Residuals 4627  12187     2.6
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

We make sure the variables are the right type, then we use the `aov` function. Inside of the function we have what is called a formula. It has the general structure: `leftside ~ rightside`. Generally, the left side is an outcome variable and the right side is the predictor (i.e. independent) variable. Here, we have `race` predicting `famsize`. We assign the model to the name `fit` which is a common way of denoting it is a model. Finally, we use the `anova` function to output a nice ANOVA table.

In the output we see the normal ANOVA table and we can see the p-value (`Pr(>F)`) is very, very small and thus is quite significant. We can look at how the groups relate using a box plot. We will be using some of the practice you got in Chapter 3 using `ggplot2` for this.

```
library(ggplot2)

ggplot(df, aes(x=race, y=famsize)) +
  geom_boxplot(aes(color=race)) +
  scale_color_manual(guide=FALSE,
                     values=c("dodgerblue3", "coral2", "chartreuse4",
                              "darkorchid", "firebrick2")) +
  theme_bw()
```



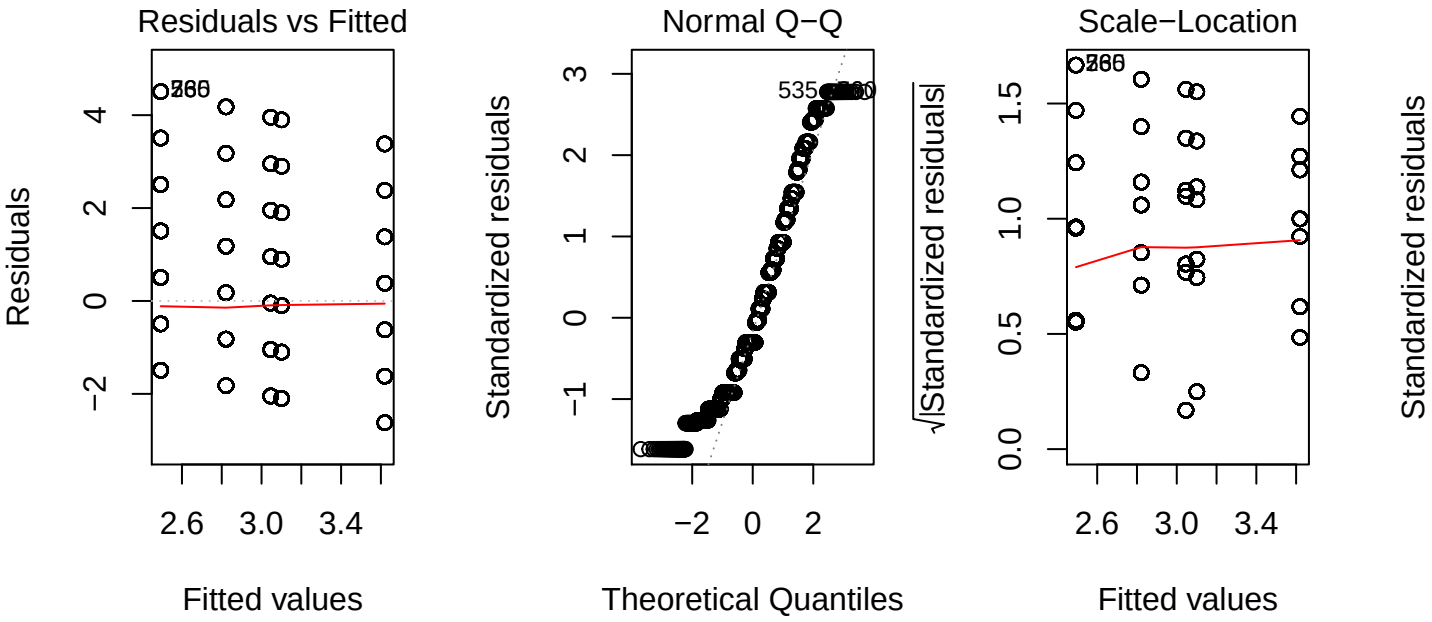
This immediately gives us an idea of where some differences may be occuring. It would appear that "White" and "MexicanAmerican" groups are different in family size.

Assumptions

We also would like to make sure the assumptions look like they are being met. In ANOVA, we want the residuals to be distributed normally, the variance of each group should be approximately the same, the groups are assumed to be randomly assigned, and the sample should be randomly selected as well.

In `R` we can get some simple graphical checks using `plot`. All we provide is our ANOVA object (here it is `fit`). The line before it `par(mfrow=c(1,2))` tells `R` to have two plots per row (the 1 means one row, 2 means two columns).

```
par(mfrow=c(1,2))
plot(fit)
```



Here, it looks like we have a problem with normality (see the Normal Q-Q plot). Those dots should approximately follow the dotted line, which is not the case. In the first plot (Residuals vs. Fitted) suggests we have approximate homoskedasticity.

Linear Modeling

Linear regression is nearly identical to ANOVA. In fact, a linear regression with a continuous outcome and categorical predictor is exactly the same (if we use effect coding). For example, if we run the same model but with the linear regression function `lm` we get the same ANOVA table.

```
fit2 <- lm(famsize ~ race, data=df)
anova(fit2)
```

```
## Analysis of Variance Table
##
## Response: famsize
##             Df Sum Sq Mean Sq F value Pr(>F)
## race         4    541   135.3    51.4 <2e-16 ***
## Residuals 4627  12187     2.6
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Surprise! It is the same as before. Here we can also use the `summary` function and we get the coefficients in the model as well (using dummy coding). The first level of the categorical variable is the reference group (the group that the others are compared to). We also get the intercept (in this case, the average value of the reference group).

```
summary(fit2)
```

```
##
## Call:
## lm(formula = famsize ~ race, data = df)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -2.619 -1.493 -0.493  0.954  4.507
```

```
##
## Coefficients:
##                   Estimate Std. Error t value Pr(>|t|)
## (Intercept)         3.6193     0.0777   46.56  < 2e-16 ***
## raceOtherHispanic  -0.5184     0.1081   -4.79  1.7e-06 ***
## raceWhite          -1.1260     0.0868  -12.98  < 2e-16 ***
## raceBlack          -0.7980     0.0906   -8.81  < 2e-16 ***
## raceOther          -0.5732     0.0980   -5.85  5.3e-09 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.62 on 4627 degrees of freedom
## Multiple R-squared:  0.0425, Adjusted R-squared:  0.0417
## F-statistic: 51.4 on 4 and 4627 DF,  p-value: <2e-16
```

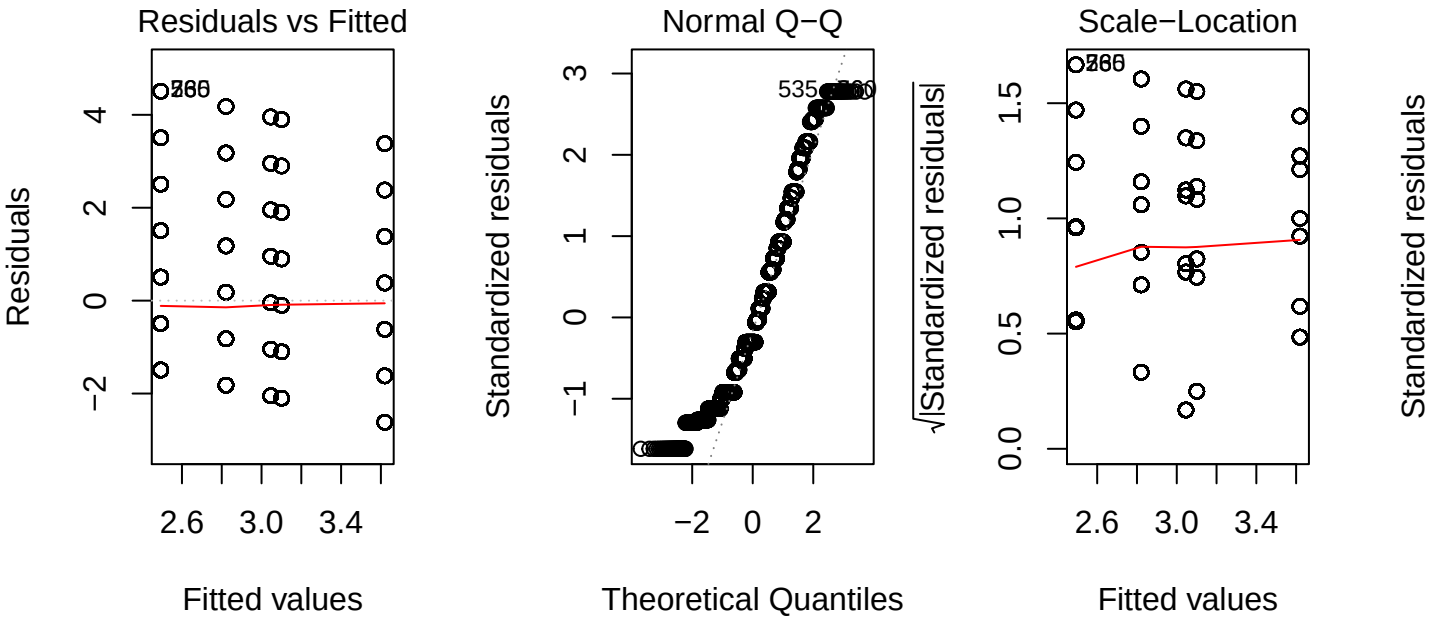
Assumptions

Linear regression has a few important assumptions, often called "Gauss-Markov Assumptions". These include:

1. The model is linear in parameters.
2. Homoskedasticity (i.e. the variance of the residual is roughly uniform across the values of the independents).
3. Normality of residuals.

Numbers 2 and 3 are fairly easy to assess using the `plot()` function on the model object as we did with the ANOVA model. The linear in parameters suggests that the relationship between the outcome and independents is linear.

```
par(mfrow=c(1,2))
plot(fit2)
```



Comparing Models

Often when running linear regression, we want to compare models and see if one fits significantly better than another. We also often want to present all the models in a table to let our readers compare the models. We will demonstrate both.

Compare Statistically

Using the `anova()` function, we can compare models statistically.

```
anova(fit, fit2)
```

```
## Analysis of Variance Table
##
## Model 1: famsize ~ race
## Model 2: famsize ~ race
##   Res.Df   RSS Df Sum of Sq F Pr(>F)
## 1   4627 12187
## 2   4627 12187  0         0
```

The `anova()` function works with all sorts of modeling schemes and can help in model selection. Not surprisingly, when we compared the ANOVA and the simple linear model, they are *exactly* the same in overall model terms (the only difference is in how the cateogrical variable is coded—either effect coding in ANOVA or dummy coding in regression). For a more interesting comparison, lets run a new model with an additional variable and then make a comparison.

```
fit3 = lm(famsize ~ race + marriage, data=df)
summary(fit3)
```

```
##
## Call:
## lm(formula = famsize ~ race + marriage, data = df)
##
## Residuals:
##    Min     1Q Median     3Q    Max
##  -2.97  -1.18  -0.41   1.03   5.29
##
## Coefficients:
##                   Estimate Std. Error t value Pr(>|t|)
## (Intercept)         3.9689     0.0779   50.95  < 2e-16 ***
## raceOtherHispanic  -0.4118     0.1037   -3.97  7.2e-05 ***
## raceWhite          -1.0268     0.0836  -12.28  < 2e-16 ***
## raceBlack          -0.5588     0.0879   -6.36  2.3e-10 ***
## raceOther          -0.5624     0.0946   -5.94  3.0e-09 ***
## marriage2          -1.2291     0.0867  -14.18  < 2e-16 ***
## marriage3          -1.2025     0.0778  -15.45  < 2e-16 ***
## marriage4          -0.4775     0.1253   -3.81  0.00014 ***
## marriage5          -0.8093     0.0597  -13.56  < 2e-16 ***
## marriage6          -0.4921     0.0887   -5.55  3.0e-08 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.55 on 4622 degrees of freedom
## Multiple R-squared:  0.126,  Adjusted R-squared:  0.124
## F-statistic: 73.7 on 9 and 4622 DF,  p-value: <2e-16
```

Notice that the variable is associated with the outcome according to the t-test seen in the summary. So we

would expect that `fit3` is better than `fit2` at explaining the outcome, which we see in the output below.

```
anova(fit2, fit3)
```

```
## Analysis of Variance Table
##
## Model 1: famsize ~ race
## Model 2: famsize ~ race + marriage
##   Res.Df   RSS Df Sum of Sq    F Pr(>F)
## 1   4627 12187
## 2   4622 11131  5      1057 87.8 <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Compare in a Table

We can also compare the models in a well-formatted table that makes many aspects easy to compare. Two main packages allow us to compare models:

1. `stargazer`
2. `texreg`

Both provide simple functions to compare multiple models. For example, `stargazer` provides:

```
library(stargazer)
stargazer(fit2, fit3,
          type = "text")
```

```
##
## ===============================================================
##                                 Dependent variable:
##                     ------------------------------------------
##                                        famsize
##                                (1)                  (2)
## ---------------------------------------------------------------
## raceOtherHispanic          -0.518***            -0.412***
##                             (0.108)              (0.104)
##
## raceWhite                  -1.130***            -1.030***
##                             (0.087)              (0.084)
##
## raceBlack                  -0.798***            -0.559***
##                             (0.091)              (0.088)
##
## raceOther                  -0.573***            -0.562***
##                             (0.098)              (0.095)
##
## marriage2                                       -1.230***
##                                                  (0.087)
##
## marriage3                                       -1.200***
##                                                  (0.078)
##
## marriage4                                       -0.477***
##                                                  (0.125)
##
## marriage5                                       -0.809***
```

```
##                                                    (0.060)
##
## marriage6                                         -0.492***
##                                                    (0.089)
##
## Constant                        3.620***           3.970***
##                                 (0.078)            (0.078)
##
## ------------------------------------------------------------------------
## Observations                     4,632              4,632
## R2                               0.043              0.126
## Adjusted R2                      0.042              0.124
## Residual Std. Error    1.620 (df = 4627)        1.550 (df = 4622)
## F Statistic         51.400*** (df = 4; 4627) 73.700*** (df = 9; 4622)
## ========================================================================
## Note:                                      *p<0.1; **p<0.05; ***p<0.01
```

When Assumptions Fail

There are many things we can try when our assumptions fail. In my opinion, the best and most interpretable way is to use a Generalized Linear Model (GLM) which is discussed in the next chapter. There are a few other things you can try which I'll show here. But, keep in mind that these things can cause other problems. For example, to fix normality we may accidentally cause heteroskedasticity. With that in mind, here are some common methods to help a model fit better.

Log-Linear, Log-Log, Linear-Log, Other

Sounds like a great tongue-twister? Well, it is but it's also three ways of specifying (i.e. deciding what is in) your model better.

Log-Linear is where we adjust the outcome variable by a natural log transformation. This is done easily in R:

```
df$log_outcome <- log(df$outcome)

lm(log_outcome ~ var1, data=df)
```

Log-Log is where we adjust both the outcome and the predictor variable with a log transformation. This is also easily done:

```
df$log_outcome <- log(df$outcome)
df$log_var1    <- log(df$var1)

lm(log_outcome ~ log_var1, data=df)
```

Linear-Log is where we adjsut just the predictor variable with a log transformation. And, you guessed it, this is easily done in R:

```
df$log_var1 <- log(df$var1)

lm(outcome ~ log_var1 + var2, data=df)
```

Other methods such as square rooting the outcome or using some power function (e.g. square, cube) are also quite common. There are functions that look for the best transformation to use. However, I will not cover it here since I think GLM's are better. So if you want to learn about other ways to help your linear model go to the next chapter.

Interactions

Many times hypotheses dealing with human beings include interactions between effects. Interactions are when the effect of one variable depends on another variable. For example, the effect of marital status on family size may depend on whether the individual is a minority. In fact, this is the hypothesis we'll test below.

Including interactions in ANOVA and regression type models are very simple in `R`. Since interpretations of interaction effects are often best through plots, we will also show simple methods to visualize the interactions as well.

Interactions in ANOVA

In general, we refer to ANOVA's with interactions as "2-way Factorial ANOVA's". We interact race and marriage status in this ANOVA. For simplicity, we created a binary race variable called minority using the `ifelse()` function. We explain this in more depth in Chapter 5.

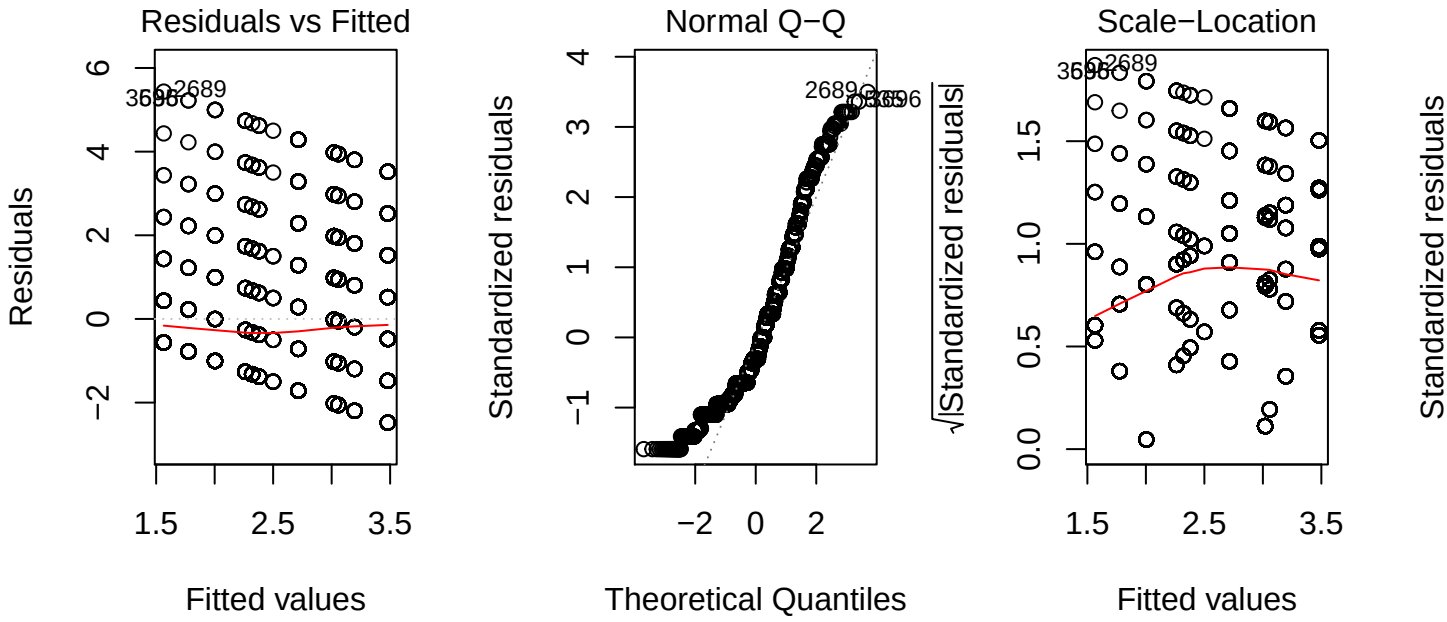
```
df$minority <- factor(ifelse(df$race == "White", 0, 1),
                      labels = c("White", "Minority"))
fit_anova <- aov(famsize ~ minority*marriage, df)
anova(fit_anova)
```

```
## Analysis of Variance Table
##
## Response: famsize
##                      Df Sum Sq Mean Sq F value Pr(>F)
## minority              1    335     335  137.96 <2e-16 ***
## marriage              5   1153     231   95.01 <2e-16 ***
## minority:marriage     5     25       5    2.05  0.068 .
## Residuals          4620  11216       2
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Notice two things: First, the interaction is significant (p = .003). This is important since we are going to try to interpret this interaction. Second, by including `minority*marriage` we get both the main effects and the interaction. This is very important for interpretation purposes so you can thank `R` for making it a bit more easy on you.

We can check the assumptions the same way as before:

```
par(mfrow=c(1,2))
plot(fit_anova)
```



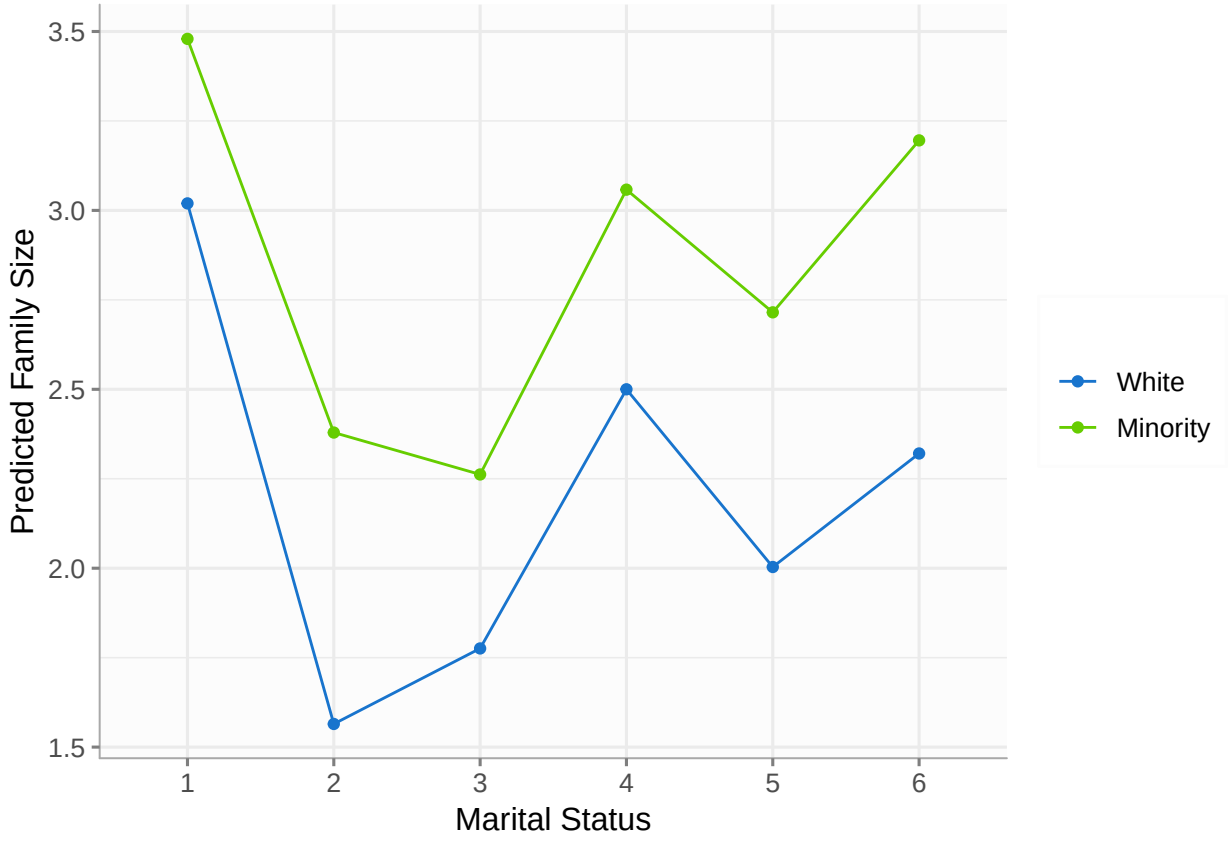
Again, the assumptions are not met for this model. But, if we ignore that for now, we can quickly find a way to interpret the interaction.

We first create a new data set that is composed of every possible combination of the variables in the model. This allows us to get unbiased estimates for the plotting.

```
newdata <- expand.grid(minority = levels(df$minority),
                       marriage = levels(df$marriage))
newdata$preds <- predict(fit_anova, newdata=newdata)
```

We now use `ggplot2` just as before.

```
ggplot(newdata, aes(x = marriage, y = preds, group = minority)) +
  geom_line(aes(color = minority)) +
  geom_point(aes(color = minority)) +
  labs(y = "Predicted Family Size",
       x = "Marital Status") +
  scale_color_manual(name = "",
                     values = c("dodgerblue3", "chartreuse3")) +
  theme_anteo_wh()           ## from anteo package
```



The plot tells use a handful of things. For example, we see minorities generally have more children across marital statuses. However, the difference is smaller for married and divorced individuals compared to widowed, separated, never married, and living with a partner. There's certainly more to gleen from the plot, but we won't waste your time.

Interactions in Linear Regression

Interactions in linear regression is nearly identical as in ANOVA, except we use dummy coding. It provides a bit more information. For example, we get the coefficients from the linear regression whereas the ANOVA does not provide this. We can run a regression model via:

```
fit_reg <- lm(famsize ~ minority*marriage, df)
summary(fit_reg)
```

```
##
## Call:
## lm(formula = famsize ~ minority * marriage, data = df)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -2.479 -1.196 -0.479  0.980  5.435
##
## Coefficients:
##                            Estimate Std. Error t value Pr(>|t|)
## (Intercept)                  3.0195     0.0513   58.85  < 2e-16 ***
## minorityMinority             0.4599     0.0672    6.84  8.9e-12 ***
## marriage2                   -1.4548     0.1301  -11.19  < 2e-16 ***
## marriage3                   -1.2437     0.1163  -10.70  < 2e-16 ***
```

```
## marriage4                    -0.5195     0.2891   -1.80    0.072 .
## marriage5                    -1.0161     0.1041   -9.76  < 2e-16 ***
## marriage6                    -0.6989     0.1455   -4.80  1.6e-06 ***
## minorityMinority:marriage2    0.3546     0.1741    2.04    0.042 *
## minorityMinority:marriage3    0.0263     0.1561    0.17    0.866
## minorityMinority:marriage4    0.0981     0.3210    0.31    0.760
## minorityMinority:marriage5    0.2518     0.1268    1.99    0.047 *
## minorityMinority:marriage6    0.4152     0.1833    2.26    0.024 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.56 on 4620 degrees of freedom
## Multiple R-squared:  0.119,  Adjusted R-squared:  0.117
## F-statistic: 56.7 on 11 and 4620 DF,  p-value: <2e-16
```

We used `summary()` to see the coefficients. If we used `anova()` it would have been the same as the one for the ANOVA.

We can use the exact same methods here as we did with the ANOVA, including checking assumptions, creating a new data set, and using `ggplot2` to check the interaction. We won't repeat it here so you can move on to Chapter 5.

Apply It

This link contains a folder complete with an Rstudio project file, an RMarkdown file, and a few data files. Download it and unzip it to do the following steps.

Step 1

Open the `Chapter4.Rproj` file. This will open up RStudio for you.

Step 2

Once RStudio has started, in the panel on the lower-right, there is a `Files` tab. Click on that to see the project folder. You should see the data files and the `Chapter4.Rmd` file. Click on the `Chapter4.Rmd` file to open it. In this file, import the data and run each type of statistical analysis presented in this chapter (there are others that are presented in Chapters 5, 6, and 7 as well that you do not need to do yet).

Once that code is in the file, click the `knit` button. This will create an HTML file with the code and output knitted together into one nice document. This can be read into any browser and can be used to show your work in a clean document.

Chapter 5: Generalized Linear Models

> "You must stick to your conviction, but be ready to abandon your assumptions." — Dennis Waitley

Generalized Linear Models (GLM's) are extensions of linear regression to areas where assumptions of normality and homoskedasticity do not hold. There are several versions of GLM's, each for different types and distributions of outcomes. We are going to go through several of the most common.

This chapter is to introduce the method very briefly and demonstrate how to perform one in `R`. We do not delve into the details of each method much, but rather focus on showing the quirks of the coding.

We discuss:

1. Logistic Regression
2. Poisson Regression
3. GLM with Gamma distribution
4. Negative binomial
5. Beta Regression

Logistic Regression

For binary outcomes (e.g., yes or no, correct or incorrect, sick or healthy), logistic regression is a fantastic tool that provides useful and interpretable information. Much like simple and multiple linear regression, logistic regression[18] uses dummy coding and provides coefficients that tell us the relationship between the outcome and the independent variables.

Since the outcome is binary, we use a statistical transformation to make things work well. This makes it so the outcome is in "log-odds." A simple exponentiation of the coefficients and we get very useful "odds ratios." These are very common in many fields using binary data.

Luckily, running a logistic regression is simple in `R`. We first create the binary outcome variable called `dep`. We use a new function called `mutate` to create a new variable (we could do this a number of ways but this is probably the cleanest way).

```
## First creating binary depression variable
df <- df %>%
  mutate(dep = dpq010 + dpq020 + dpq030 + dpq040 + dpq050 +
               dpq060 + dpq070 + dpq080 + dpq090) %>%
  mutate(dep2 = ifelse(dep >= 10, 1,
                ifelse(dep < 10, 0, NA)))
```

Note that we added the values from the ten variables that give us an overall depression score (`dep`). We then use `ifelse()` to create a binary version of depression called `dep2` with a cutoff of ≥ 16 meaning depressed. Because there are missing values denoted as "NA" in this variable, we use a "nested ifelse" to say:

[18] Technically, logistic regression is a linear regression model.

1. IF depression ≥ 10 then dep2 is 1,
2. IF dpression < 10, then dep2 is 0,
3. ELSE dep2 is NA.

Note that these nested `ifelse()` statements can be as long as you want. We further need to clean up the asthma and sedentary variables.

```
## Fix some placeholders
df <- df %>%
  mutate(asthma = washer(mcq010, 9),
         asthma = washer(asthma, 2, value = 0)) %>%
  mutate(sed = washer(pad680, 9999, 7777))
```

Now let's run the logistic regression:

```
l_fit <- glm(dep2 ~ asthma + sed + race + famsize,
             data = df,
             family = "binomial")
summary(l_fit)
```

```
##
## Call:
## glm(formula = dep2 ~ asthma + sed + race + famsize, family = "binomial",
##     data = df)
##
## Deviance Residuals:
##    Min      1Q  Median      3Q     Max
## -0.783  -0.448  -0.408  -0.364   2.547
##
## Coefficients:
##                    Estimate Std. Error z value Pr(>|z|)
## (Intercept)       -2.620355   0.238077  -11.01  < 2e-16 ***
## asthma             0.568845   0.127633    4.46  8.3e-06 ***
## sed                0.000564   0.000261    2.16   0.0307 *
## raceOtherHispanic  0.716257   0.232867    3.08   0.0021 **
## raceWhite          0.128706   0.211641    0.61   0.5431
## raceBlack          0.018921   0.220546    0.09   0.9316
## raceOther         -0.490141   0.257012   -1.91   0.0565 .
## famsize           -0.031831   0.037322   -0.85   0.3937
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 2706.3  on 4436  degrees of freedom
## Residual deviance: 2648.2  on 4429  degrees of freedom
##   (195 observations deleted due to missingness)
## AIC: 2664
##
## Number of Fisher Scoring iterations: 5
```

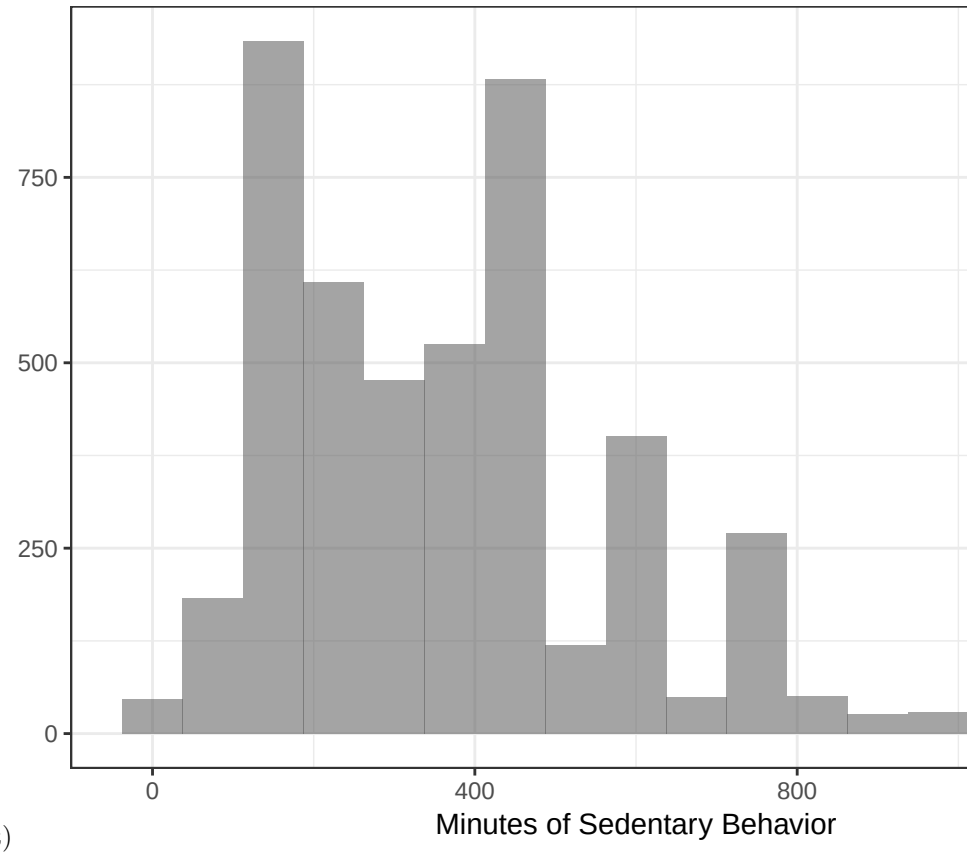
We used `glm()` (stands for generalized linear model). The key to making it logistic, since you can use `glm()` for a linear model using maximum likelihood instead of `lm()` with least squares, is `family = "binomial"`. This tells `R` to do a logistic regression.

Poisson Regression

As we did in logistic regression, we will use the `glm()` function. The difference here is we will be using an outcome that is a count variable. For example, the sedentary variable (`sed`) that we have in `df` is a count of the minutes of sedentary activity.

```
p_fit <- glm(sed ~ asthma + race + famsize,
             data = df,
             family = "poisson")
summary(p_fit)
```

```
##
## Call:
## glm(formula = sed ~ asthma + race + famsize, family = "poisson",
##     data = df)
##
## Deviance Residuals:
##    Min      1Q  Median      3Q     Max
## -27.36   -8.43   -1.48    5.82   34.51
##
## Coefficients:
##                    Estimate Std. Error z value Pr(>|z|)
## (Intercept)        5.649987   0.003555  1589.3   <2e-16 ***
## asthma             0.061496   0.002143    28.7   <2e-16 ***
## raceOtherHispanic  0.139344   0.004094    34.0   <2e-16 ***
## raceWhite          0.348462   0.003344   104.2   <2e-16 ***
## raceBlack          0.340035   0.003443    98.8   <2e-16 ***
## raceOther          0.355795   0.003627    98.1   <2e-16 ***
## famsize           -0.018867   0.000549   -34.4   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for poisson family taken to be 1)
##
##     Null deviance: 496351  on 4436  degrees of freedom
## Residual deviance: 475428  on 4430  degrees of freedom
##   (195 observations deleted due to missingness)
## AIC: 508999
##
## Number of Fisher Scoring iterations: 5
```



Sedentary may be over-dispersed (see plot)
and so other methods related to poisson may be necessary. For this book, we are not going to be delving into these in depth but we will introduce some below.

Gamma

Regression with a gamma distribution are often found when analyzing costs in dollars. It is very similar to poisson but does not require integers and can handle more dispersion. However, the outcome must have values > 0. Just for demonstration:

```
## Adjust sed
df$sed_gamma <- df$sed + .01
g_fit <- glm(sed_gamma ~ asthma + race + famsize,
             data = df,
             family = "Gamma")
summary(g_fit)
```

```
##
## Call:
## glm(formula = sed_gamma ~ asthma + race + famsize, family = "Gamma",
##     data = df)
##
## Deviance Residuals:
##    Min      1Q  Median      3Q     Max
## -4.359  -0.461  -0.084   0.293   1.687
##
## Coefficients:
##                   Estimate Std. Error t value Pr(>|t|)
## (Intercept)       3.57e-03   1.13e-04   31.51   <2e-16 ***
```

```
## asthma            -1.60e-04   5.86e-05   -2.74   0.0063 **
## raceOtherHispanic -4.87e-04   1.31e-04   -3.72   0.0002 ***
## raceWhite         -1.09e-03   1.08e-04  -10.11   <2e-16 ***
## raceBlack         -1.07e-03   1.10e-04   -9.70   <2e-16 ***
## raceOther         -1.11e-03   1.15e-04   -9.70   <2e-16 ***
## famsize            5.11e-05   1.55e-05    3.29   0.0010 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for Gamma family taken to be 0.293)
##
##     Null deviance: 1664.8  on 4436  degrees of freedom
## Residual deviance: 1604.2  on 4430  degrees of freedom
##   (195 observations deleted due to missingness)
## AIC: 59154
##
## Number of Fisher Scoring iterations: 5
```

Two-Part or Hurdle Models

We are going to use the `pscl` package to run a hurdle model. These models are built for situations where there is a count variable with many zeros ("zero-inflated"). The hurdle model makes slightly different assumptions regarding the zeros than the pure negative binomial that we present next. The hurdle consists of two models: one for whether the person had a zero or more (binomial) and if more than zero, how many (poisson).

To run a hurdle model, we are going to make a sedentary variable with many more zeros to illustrate and then we will run a hurdle model.

```
## Zero inflated sedentary (don't worry too much about the specifics)
df$sed_zero <- ifelse(sample(1:100,
                             size = length(df$sed),
                             replace=TRUE) %in% c(5,10,11,20:25), 0,
                      df$sed)
## Hurdle model
library(pscl)
h_fit = hurdle(sed_zero ~ asthma + race + famsize,
               data = df)
summary(h_fit)
```

```
##
## Call:
## hurdle(formula = sed_zero ~ asthma + race + famsize, data = df)
##
## Pearson residuals:
##    Min     1Q Median     3Q    Max
## -3.157 -1.501 -0.231  1.244  9.455
##
## Count model coefficients (truncated poisson with log link):
##                     Estimate Std. Error z value Pr(>|z|)
## (Intercept)         5.662968   0.003709  1526.7   <2e-16 ***
## asthma              0.059152   0.002255    26.2   <2e-16 ***
## raceOtherHispanic   0.126674   0.004274    29.6   <2e-16 ***
## raceWhite           0.332563   0.003487    95.4   <2e-16 ***
## raceBlack           0.324970   0.003589    90.5   <2e-16 ***
## raceOther           0.336748   0.003797    88.7   <2e-16 ***
```

```
## famsize           -0.018251   0.000578   -31.6   <2e-16 ***
## Zero hurdle model coefficients (binomial with logit link):
##                   Estimate Std. Error z value Pr(>|z|)
## (Intercept)         2.3312     0.2104   11.08   <2e-16 ***
## asthma             -0.0436     0.1423   -0.31     0.76
## raceOtherHispanic   0.0137     0.2414    0.06     0.95
## raceWhite          -0.0330     0.1958   -0.17     0.87
## raceBlack           0.0037     0.2039    0.02     0.99
## raceOther          -0.2030     0.2125   -0.96     0.34
## famsize            -0.0107     0.0354   -0.30     0.76
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Number of iterations in BFGS optimization: 12
## Log-likelihood: -2.32e+05 on 14 Df
```

Notice that the output has two parts: "Count model coefficients (truncated poisson with log link):" and "Zero hurdle model coefficients (binomial with logit link):". Together they tell us about the relationship between the predictors and a count variable with many zeros.

Negative Binomial

Similar to that above, negative binomial is for zero-inflated count variables. It makes slightly different assumptions than the hurdle and doesn't use a two-part approach. In order to run a negative binomial model we'll use the `MASS` package and the `glm.nb()` function.

```
library(MASS)
fit_nb <- glm.nb(sed_zero ~ asthma + race + famsize,
                 data = df)
summary(fit_nb)
```

Note that this model is not really appropriate because our data is somewhat contrived.

Beta Regression

For outcomes that are bound between a lower and upper bound, Beta Regression is a great method. For example, if we are looking at test scores that are bound between 0 and 100. It is a very flexible method and allows for some extra analysis regarding the variation.

For this, we are going to use the `betareg` package. But first, we are going to reach a little and create a ficticiously bound variable in the data set.

```
## Variable bound between 0 and 1
df$beta_var <- sample(seq(.05, .99, by = .01),
                      size = length(df$asthma),
                      replace = TRUE)
library(betareg)
fit_beta <- betareg(beta_var ~ asthma + race + famsize,
                    data = df)
summary(fit_beta)
```

```
##
## Call:
## betareg(formula = beta_var ~ asthma + race + famsize, data = df)
##
## Standardized weighted residuals 2:
```

```
##    Min     1Q Median     3Q    Max
## -2.044 -0.689 -0.052  0.626  2.898
##
## Coefficients (mean model with logit link):
##                    Estimate Std. Error z value Pr(>|z|)
## (Intercept)        0.139651   0.063114    2.21    0.027 *
## asthma            -0.010276   0.043610   -0.24    0.814
## raceOtherHispanic  0.003176   0.071845    0.04    0.965
## raceWhite          0.005926   0.058610    0.10    0.919
## raceBlack          0.000894   0.060847    0.01    0.988
## raceOther          0.037336   0.065222    0.57    0.567
## famsize           -0.013627   0.010798   -1.26    0.207
##
## Phi coefficients (precision model with identity link):
##       Estimate Std. Error z value Pr(>|z|)
## (phi)   2.4875     0.0453    54.9   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Type of estimator: ML (maximum likelihood)
## Log-likelihood: 87.3 on 8 Df
## Pseudo R-squared: 0.000533
## Number of iterations: 17 (BFGS) + 1 (Fisher scoring)
```

The output provides coefficients and the "Phi" coefficients. Both are important parts of using beta regression but we are not going to discuss it here.

There are many resources available to learn more about beta regression and each of these GLM's. As for now, we are going to move on to more complex modeling where there are clustering or repeated measures in the data.

Apply It

This link contains a folder complete with an Rstudio project file, an RMarkdown file, and a few data files. Download it and unzip it to do the following steps.

Step 1

Open the `Chapter4.Rproj` file. This will open up RStudio for you.

Step 2

Once RStudio has started, in the panel on the lower-right, there is a `Files` tab. Click on that to see the project folder. You should see the data files and the `Chapter4.Rmd` file. Click on the `Chapter4.Rmd` file to open it. In this file, import the data and run each type of statistical analysis presented in this chapter (there are others that are presented in Chapter 4—that you may have done already—and Chapters 6 and 7 that you do not need to do yet).

Once that code is in the file, click the `knit` button. This will create an HTML file with the code and output knitted together into one nice document. This can be read into any browser and can be used to show your work in a clean document.

Conclusions

One of the great things about `R` is that most modeling is very similar to the basic `lm()` function. In all of these GLM's the arguments are nearly all the same: a formula, the data, and family of model. As you'll see for Multilevel and Other Models chapters, this does not change much. Having a good start with basic models and GLM's gets you ready for nearly every other modeling type in `R`.

Chapter 6: Multilevel Modeling

"Simplicity does not precede complexity, but follows it." — Alan Perlis

Multilevel data are more complex and don't meet the assumptions of regular linear or generalized linear models. But with the right modeling schemes, the results can be very interpretable and actionable. Two powerful forms of multilevel modeling are:

1. Generalized Estimating Equations (GEE)
2. Mixed effects (ME; i.e., hierarchical linear modeling, multilevel modeling)

Several similarities and differences should be noted briefly. As for similarities, they both attempt to control for the lack of independence within clusters, although they do it in different ways. Also, they are both built on linear regression which makes them flexible and powerful at finding relationships in the data.

The differences are subtle but important. First, the interpretation is somewhat different between the two. GEE is a population-averaged (e.g., marginal) model whereas ME is subject specific. In other words, *GEE is the average effect* while *ME is the effect found in the average person.* In a linear model, these coefficients are the same but when we use different forms such as logistic or poisson, these can be quite different (although in my experience they generally tell a similar story). Second, ME models are much more complex than the GEE models and can struggle with convergence compared to the GEE. This also means that GEE's are generally fitted much more quickly. Still the choice of the modeling technique should be driven by your hypotheses and not totally dependent on speed of the computation.

First, if we needed to, we'd reshape our data so that it is ready for the analyses (see Chapter 8 for more on reshaping). For both modeling techniques we want our data in long form[19]. What this implies is that each row is an observation. What this actually means about the data depends on the data. For example, if you have repeated measures, then often data is stored in wide form—a row is an individual. To make this long, we want each time point within a person to be a row—a single individual can have multiple rows but each row is a unique observation.

Currently, our data is in long form since we are working within community clusters within this data. So, each row is an observation and each cluster has multiple rows. Note that although these analyses will be within community clusters instead of within subjects (i.e. repeated measures), the overall steps will be the exact same.

This chapter certainly does not cover all of multilevel modeling in `R`. Entire books are dedicated to that single subject. Rather, we are introducing the methods and the packages that can be used to start using these methods.

GEE

There are two packages, intimately related, that allow us to perform GEE modeling—`gee` and `geepack`. These have some great features and make running a fairly complex model pretty simple. However, as great

[19]We discuss what this means in much more depth and demonstrate reshaping of data in Chapter 8. It is an important tool to understand if you are working with data in various forms. Although many reshape their data by copying-and-pasting in a spreadsheet, what we present in Chapter 8 is much more efficient, cleaner, less error-prone, and replicatable.

as they are, there are some annoying shortcomings. We'll get to a few of them throughout this section.

GEE's, in general, want a few pieces of information from you. First, the outcome and predictors. This is just as in linear regression and GLM's. Second, we need to provide a correlation structure. This tells the model the approximate pattern of correlations between the time points or clusters. It also wants a variable that tells the cluster ID's. Finally, it also wants the family (i.e. the type of distribution).

Since this is not longitudinal, but rather clustered within communities, we'll assume for this analysis an unstructured correlation structure. It is the most flexible and we have enough power for it here.

For `geepack` to work, we need to filter out the missing values for the variables that will be in the model.

```
df2 <- df %>%
  filter(complete.cases(dep, famsize, sed, race, asthma))
```

Now, we'll build the model with both packages (just for demonstration). We predict depression with asthma, family size, minutes of sedentary behavior, and the subject's race.

```
library(gee)
fit_gee <- gee(dep ~ asthma + famsize + sed + race,
               data = df2,
               id = df2$sdmvstra,
               corstr = "unstructured")
```

```
##       (Intercept)      asthmaAsthma          famsize              sed
##           2.50002           1.35608          -0.04213           0.00136
## raceOtherHispanic         raceWhite         raceBlack         raceOther
##           1.18500           0.11395           0.10054          -0.55548
```

```
summary(fit_gee)$coef
```

```
##                   Estimate Naive S.E. Naive z Robust S.E. Robust z
## (Intercept)        2.49551   0.286782   8.702    0.269043    9.276
## asthmaAsthma       1.35304   0.186710   7.247    0.213798    6.329
## famsize           -0.03949   0.046195  -0.855    0.045747   -0.863
## sed                0.00136   0.000336   4.039    0.000355    3.823
## raceOtherHispanic  1.19248   0.307556   3.877    0.330961    3.603
## raceWhite          0.11619   0.253155   0.459    0.227969    0.510
## raceBlack          0.09680   0.262583   0.369    0.236050    0.410
## raceOther         -0.55505   0.280930  -1.976    0.240657   -2.306
```

```
library(geepack)
fit_geeglm <- geeglm(dep ~ asthma + famsize + sed + race,
                     data = df2,
                     id = df2$sdmvstra,
                     corstr = "unstructured")
summary(fit_geeglm)
```

```
##
## Call:
## geeglm(formula = dep ~ asthma + famsize + sed + race, data = df2,
##     id = df2$sdmvstra, corstr = "unstructured")
##
##  Coefficients:
##                    Estimate   Std.err  Wald Pr(>|W|)
## (Intercept)        2.557936  0.270072 89.71  < 2e-16 ***
## asthmaAsthma       1.349289  0.215620 39.16  3.9e-10 ***
## famsize           -0.044672  0.045709  0.96  0.32841
```

```
## sed                0.001301  0.000355 13.45  0.00024 ***
## raceOtherHispanic  1.175037  0.331898 12.53  0.00040 ***
## raceWhite          0.080638  0.229566  0.12  0.72539
## raceBlack          0.064203  0.236326  0.07  0.78587
## raceOther         -0.590205  0.241338  5.98  0.01446 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Estimated Scale Parameters:
##             Estimate Std.err
## (Intercept)     19.5   0.784
##
## Correlation: Structure = unstructured  Link = identity
##
## Estimated Correlation Parameters:
##           Estimate Std.err
## alpha.1:2   0.1248  0.0165
## alpha.1:3   0.4207  0.1034
## alpha.1:4   2.8964  1.0668
## alpha.1:5  -1.8545  0.2028
## alpha.2:3   0.1224  0.0633
## alpha.2:4  -0.0894  0.2023
## alpha.2:5   0.2054  0.0372
## alpha.3:4  -0.4960  0.1123
## alpha.3:5   0.2504  0.0388
## alpha.4:5  -0.6694  0.0876
## Number of clusters:   4109   Maximum cluster size: 5
```

The `gee` package doesn't directly provide p-values but provides the z-scores, which can be used to find the p-values. The `geepack` provides the p-values in the way you'll see in the `lm()` and `glm()` functions.

These models are interpreted just as the regular GLM. It has adjusted for the correlations within the clusters and provides valid standard errors and p-values.

Mixed Effects

Mixed effects models require a bit more thinking about the effects. It is called "mixed effects" because we include both fixed and random effects into the model simultaneously. The random effects are those that we don't necessarily care about the specific values but want to control for it and/or estimate the variance. The fixed effects are those we are used to estimating in linear models and GLM's.

These are a bit more clear with an example. We will do the same overall model as we did with the GEE but we'll use ME. To do so, we'll use the `lme4` package. In the model below, we predict depression with asthma, family size, minutes of sedentary behavior, and the subject's race. We have a random intercept (which allows the intercept to vary across clusters).

```
library(lme4)
fit_me <- lmer(dep ~ asthma + famsize + sed + race + (1 | cluster),
               data = df2,
               REML = FALSE)
summary(fit_me)
```

```
## Linear mixed model fit by maximum likelihood  ['lmerMod']
## Formula: dep ~ asthma + famsize + sed + race + (1 | cluster)
##    Data: df2
##
```

```
##      AIC      BIC   logLik deviance df.resid
##    25780    25844   -12880    25760     4427
##
## Scaled residuals:
##    Min     1Q Median     3Q    Max
## -1.327 -0.635 -0.355  0.272  5.435
##
## Random effects:
##  Groups   Name        Variance Std.Dev.
##  cluster  (Intercept)  0.105   0.324
##  Residual             19.389   4.403
## Number of obs: 4437, groups:  cluster, 14
##
## Fixed effects:
##                     Estimate Std. Error t value
## (Intercept)         2.491678   0.302768    8.23
## asthmaAsthma        1.335445   0.186618    7.16
## famsize            -0.042857   0.046341   -0.92
## sed                 0.001425   0.000337    4.23
## raceOtherHispanic   1.289890   0.320595    4.02
## raceWhite           0.008348   0.259449    0.03
## raceBlack           0.171658   0.273382    0.63
## raceOther          -0.552746   0.285512   -1.94
##
## Correlation of Fixed Effects:
##             (Intr) asthmA famsiz sed    rcOthH racWht rcBlck
## asthmaAsthm -0.042
## famsize     -0.510 -0.004
## sed         -0.324 -0.044  0.051
## rcOthrHspnc -0.556 -0.032  0.051 -0.038
## raceWhite   -0.680 -0.038  0.135 -0.148  0.639
## raceBlack   -0.643 -0.057  0.094 -0.131  0.624  0.775
## raceOther   -0.580  0.000  0.048 -0.135  0.589  0.725  0.693
```

You'll see that there are no p-values provided here. This is because p-values are not well-defined in the ME framework. A good way to test it can be through the `anova()` function, comparing models. Let's compare a model with and without `asthma` to see if the model is significantly better with it in.

```
fit_me1 <- lmer(dep ~ famsize + sed + race + (1 | cluster),
                data = df2,
                REML = FALSE)

anova(fit_me, fit_me1)
```

```
## Data: df2
## Models:
## fit_me1: dep ~ famsize + sed + race + (1 | cluster)
## fit_me: dep ~ asthma + famsize + sed + race + (1 | cluster)
##         Df   AIC   BIC logLik deviance Chisq Chi Df Pr(>Chisq)
## fit_me1  9 25829 25886 -12905    25811
## fit_me  10 25780 25844 -12880    25760  50.9      1    9.9e-13 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

This comparison strongly suggests that `asthma` is a significant predictor ($\chi^2 = 50.5$, $p < .001$). We can do this with both fixed and random effects, as below:

```
fit_me2 <- lmer(dep ~ famsize + sed + race + (1 | cluster),
                data = df2,
                REML = TRUE)
fit_me3 <- lmer(dep ~ famsize + sed + race + (1 + asthma | cluster),
                data = df2,
                REML = TRUE)
anova(fit_me2, fit_me3, refit = FALSE)
```

```
## Data: df2
## Models:
## fit_me2: dep ~ famsize + sed + race + (1 | cluster)
## fit_me3: dep ~ famsize + sed + race + (1 + asthma | cluster)
##         Df   AIC   BIC logLik deviance Chisq Chi Df Pr(>Chisq)
## fit_me2  9 25855 25912 -12918    25837
## fit_me3 11 25821 25892 -12900    25799  37.3      2      8e-09 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Here, including random slopes for asthma appears to be significant ($\chi^2 = 36.9$, p $< .001$).

Linear mixed effects models converge pretty well. You'll see that the conclusions and estimates are very similar to that of the GEE. For generalized versions of ME, the convergence can be harder and more picky. As we'll see below, it complains about large eigenvalues and tells us to rescale some of the variables.

```
library(lme4)
fit_gme <- glmer(dep2 ~ asthma + famsize + sed + race + (1 | cluster),
                 data = df2,
                 family = "binomial")
```

```
## Warning in checkConv(attr(opt, "derivs"), opt$par, ctrl =
## control$checkConv, : Model failed to converge with max|grad| = 0.00853519
## (tol = 0.001, component 1)
```

```
## Warning in checkConv(attr(opt, "derivs"), opt$par, ctrl = control$checkConv, : Model is nearly unide
##  - Rescale variables?;Model is nearly unidentifiable: large eigenvalue ratio
##  - Rescale variables?
```

After a quick check, we can see that `sed` is huge compared to the other variables. If we simply rescale it, using the `I()` function within the model formula, we can rescale it by 1,000. Here, that is all it needed to converge.

```
library(lme4)
fit_gme <- glmer(dep2 ~ asthma + famsize + I(sed/1000) + race + (1 | cluster),
                 data = df2,
                 family = "binomial")
summary(fit_gme)
```

```
## Generalized linear mixed model fit by maximum likelihood (Laplace
##   Approximation) [glmerMod]
##  Family: binomial  ( logit )
## Formula: dep2 ~ asthma + famsize + I(sed/1000) + race + (1 | cluster)
##    Data: df2
##
##      AIC      BIC   logLik deviance df.resid
##     2665     2722    -1323     2647     4428
##
## Scaled residuals:
```

```
##    Min     1Q Median     3Q    Max
## -0.635 -0.329 -0.295 -0.258  5.032
##
## Random effects:
##  Groups  Name        Variance Std.Dev.
##  cluster (Intercept) 0.0232   0.152
## Number of obs: 4437, groups:  cluster, 14
##
## Fixed effects:
##                   Estimate Std. Error z value Pr(>|z|)
## (Intercept)        -2.6316     0.2435  -10.81  < 2e-16 ***
## asthmaAsthma        0.5619     0.1281    4.39  1.1e-05 ***
## famsize            -0.0336     0.0374   -0.90   0.3696
## I(sed/1000)         0.5835     0.2618    2.23   0.0258 *
## raceOtherHispanic   0.7564     0.2421    3.12   0.0018 **
## raceWhite           0.0955     0.2159    0.44   0.6581
## raceBlack           0.0531     0.2277    0.23   0.8155
## raceOther          -0.4950     0.2590   -1.91   0.0560 .
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Correlation of Fixed Effects:
##             (Intr) asthmA famsiz I(/100 rcOthH racWht rcBlck
## asthmaAsthm -0.057
## famsize     -0.491 -0.012
## I(sed/1000) -0.324 -0.042  0.031
## rcOthrHspnc -0.653 -0.031  0.044 -0.029
## raceWhite   -0.715 -0.037  0.132 -0.148  0.709
## raceBlack   -0.684 -0.064  0.088 -0.124  0.715  0.781
## raceOther   -0.571 -0.003  0.046 -0.122  0.606  0.688  0.653
```

Apply It

This link contains a folder complete with an Rstudio project file, an RMarkdown file, and a few data files. Download it and unzip it to do the following steps.

Step 1

Open the `Chapter4.Rproj` file. This will open up RStudio for you.

Step 2

Once RStudio has started, in the panel on the lower-right, there is a `Files` tab. Click on that to see the project folder. You should see the data files and the `Chapter4.Rmd` file. Click on the `Chapter4.Rmd` file to open it. In this file, import the data and run each type of statistical analysis presented in this chapter (there are others that are presented in Chapters 4 and 5 that you may have done already and methods from Chapter 7 that you have not done yet).

Once that code is in the file, click the `knit` button. This will create an HTML file with the code and output knitted together into one nice document. This can be read into any browser and can be used to show your work in a clean document.

Conclusions

This has been a really brief introduction into a thriving, large field of statistical analyses. These are the general methods for using R to analyze multilevel data. Our next chapter will discuss more modeling techniques in R, including mediation, mixture, and structural equation modeling.

Chapter 7: Other Modeling Techniques

"Simplicity is the ultimate sophistication." — Leonardo da Vinci

In this chapter we cover, however briefly, modeling techniques that are especially useful to make complex relationships easier to interpret. We will focus on mediation and moderation modeling, methods relating to structural equation modeling (SEM), and methods applicable to our field from machine learning. Although these machine learning may appear very different than mediation and SEM, they each have advantages that can help in different situations. For example, SEM is useful when we know there is a high degree of measurement error or our data has multiple indicators for each construct. On the other hand, regularized regression and random forests–two popular forms of machine learning–are great to explore patterns and relationships there are hundreds or thousands of variables that may predict an outcome.

Mediation modeling, although often used within SEM, can help us understand pathways of effect from one variable to another. It is especially useful with moderating variables (i.e., variables that interact with another).

So we'll start with discussing mediation, then we'll move on to SEM, followed by machine learning techniques.

Mediation Modeling

Mediation modeling can be done via several packages. For now, we recommend using either `lavaan` (stands for "latent variable analysis")[20] or `MarginalMediation` (written by the author of this book). Although both are technically still "beta" versions, they both perform very well especially for more simple models. It makes mediation modeling straightforward.

Below, we model the following mediation model:

$$depression = \beta_0 + \beta_1 asthma + \epsilon_1$$

$$time_{Sedentary} = \lambda_0 + \lambda_1 asthma + \lambda_2 depression + \epsilon_2$$

In essence, we believe that asthma increases depression which in turn increases the amount of time spent being sedentary. To run this with `MarginalMediation`, we will use two distinct regression models (see Chapter 4) and combine them with `mma()`. The object `pathbc` is the model with sedentary behavior as the outcome and `patha` is the path leading to the mediator.

```
library(MarginalMediation)

df$sed_hr = df$sed/60  ## in hours instead of minutes
```

[20]The `lavaan` package has some great vignettes at http://lavaan.ugent.be/ to help with the other types of models it can handle.

```r
pathbc <- glm(sed_hr ~ dep + asthma, data = df)
patha  <- glm(dep ~ asthma, data = df)

mma(pathbc, patha,
    ind_effects = c("asthmaAsthma-dep"))
```

```
##
## calculating a paths... b and c paths... Done.


##
##    Marginal Mediation Analysis
##
## A marginal mediation model with:
##    1 mediators
##    1 indirect effects
##    1 direct effects
##    500 bootstrapped samples
##    95% confidence interval
##    n = 4632
##
## Formulas:
##      sed_hr ~ dep + asthma
##      dep ~ asthma
##
## Regression Models:
##
##      sed_hr ~
##                           Est     SE Est/SE P-Value
##         (Intercept)  5.8857 0.0626  93.99 0.00000
##         dep          0.0437 0.0111   3.93 0.00009
##         asthmaAsthma 0.4123 0.1391   2.96 0.00305
##
##      dep ~
##                          Est     SE Est/SE P-Value
##         (Intercept)  2.95 0.0706  41.76       0
##         asthmaAsthma 1.46 0.1825   7.99       0
##
## Unstandardized Mediated Effects:
##
##    Indirect Effects:
##
##      sed_hr ~
##                                Indirect  Lower Upper
##         asthmaAsthma => dep   0.0637 0.0304 0.109
##
##    Direct Effects:
##
##      sed_hr ~
##                         Direct Lower Upper
##         asthmaAsthma  0.412 0.132 0.691
##
##
## Standardized Mediated Effects:
```

```
##
##    Indirect Effects:
##
##      sed_hr ~
##                              Indirect   Lower  Upper
##         asthmaAsthma => dep    0.019 0.00906 0.0324
##
##    Direct Effects:
##
##      sed_hr ~
##                          Direct  Lower Upper
##         asthmaAsthma  0.123 0.0394 0.206
```

This gives us the estimates of the individual regression models and the estimates of the indirect and direct effects.

To do the exact same model with `lavaan`, we can do the following:

```
library(lavaan)

## Our model
model1 <- '
  dep ~ asthma
  sed_hr ~ dep + asthma
'
## sem function to run the model
fit <- sem(model1, data = df)
summary(fit)
```

```
## lavaan 0.6-3 ended normally after 30 iterations
##
##   Optimization method                           NLMINB
##   Number of free parameters                          5
##
##                                                   Used       Total
##   Number of observations                          4614        4632
##
##   Estimator                                         ML
##   Model Fit Test Statistic                       0.000
##   Degrees of freedom                                 0
##
## Parameter Estimates:
##
##   Information                                 Expected
##   Information saturated (h1) model          Structured
##   Standard Errors                             Standard
##
## Regressions:
##                    Estimate  Std.Err  z-value  P(>|z|)
##   dep ~
##     asthma            1.478    0.183    8.084    0.000
##   sed_hr ~
##     dep               0.044    0.011    3.929    0.000
##     asthma            0.412    0.139    2.965    0.003
##
## Variances:
```

```
##                    Estimate  Std.Err  z-value  P(>|z|)
##    .dep              19.597    0.408   48.031    0.000
##    .sed_hr           11.171    0.233   48.031    0.000
```

From the output we see asthma does predict depression and depression does predict time being sedentary. There is also a direct effect of asthma on sedentary behavior even after controlling for depression. We can further specify the model to have it give us the indirect effect and direct effects tested.

```
## Our model
model2 <- '
  dep ~ a*asthma
  sed_hr ~ b*dep + c*asthma

  indirect := a*b
  total := c + a*b
'
## sem function to run the model
fit2 <- sem(model2, data = df)
summary(fit2)
```

```
## lavaan 0.6-3 ended normally after 30 iterations
##
##   Optimization method                           NLMINB
##   Number of free parameters                          5
##
##                                                   Used       Total
##   Number of observations                          4614        4632
##
##   Estimator                                         ML
##   Model Fit Test Statistic                       0.000
##   Degrees of freedom                                 0
##
## Parameter Estimates:
##
##   Information                                 Expected
##   Information saturated (h1) model          Structured
##   Standard Errors                             Standard
##
## Regressions:
##                    Estimate  Std.Err  z-value  P(>|z|)
##   dep ~
##     asthma     (a)    1.478    0.183    8.084    0.000
##   sed_hr ~
##     dep        (b)    0.044    0.011    3.929    0.000
##     asthma     (c)    0.412    0.139    2.965    0.003
##
## Variances:
##                    Estimate  Std.Err  z-value  P(>|z|)
##    .dep              19.597    0.408   48.031    0.000
##    .sed_hr           11.171    0.233   48.031    0.000
##
## Defined Parameters:
##                    Estimate  Std.Err  z-value  P(>|z|)
##     indirect          0.065    0.018    3.534    0.000
##     total             0.477    0.138    3.448    0.001
```

We defined a few things in the model. First, we gave the coefficients labels of `a`, `b`, and `c`. Doing so allows us to define the `indirect` and `total` effects. Here we see the indirect effect, although small, is significant at $p < .001$. The total effect is larger (not surprising) and is also significant.

Also note that we can make the regression equations have other covariates as well if we needed to (i.e. control for age or gender) just as we do in regular regression.

```
## Our model
model2.1 <- '
  dep ~ asthma + ridageyr
  sed_hr ~ dep + asthma + ridageyr
'
## sem function to run the model
fit2.1 <- sem(model2.1, data = df)
summary(fit2.1)
```

```
## lavaan 0.6-3 ended normally after 33 iterations
##
##   Optimization method                           NLMINB
##   Number of free parameters                          7
##
##                                                   Used       Total
##   Number of observations                          4614        4632
##
##   Estimator                                         ML
##   Model Fit Test Statistic                       0.000
##   Degrees of freedom                                 0
##   Minimum Function Value               0.0000000000000
##
## Parameter Estimates:
##
##   Information                                 Expected
##   Information saturated (h1) model          Structured
##   Standard Errors                             Standard
##
## Regressions:
##                    Estimate  Std.Err  z-value  P(>|z|)
##   dep ~
##     asthma            1.462    0.183    7.980    0.000
##     ridageyr         -0.005    0.004   -1.330    0.183
##   sed_hr ~
##     dep               0.044    0.011    3.927    0.000
##     asthma            0.412    0.139    2.956    0.003
##     ridageyr         -0.000    0.003   -0.063    0.950
##
## Variances:
##                    Estimate  Std.Err  z-value  P(>|z|)
##    .dep              19.590    0.408   48.031    0.000
##    .sed_hr           11.171    0.233   48.031    0.000
```

Although we don't show it here, we can also do moderation ("interactions") as part of the mediation model (especially using the `MarginalMediation` package).

Structural Equation Modeling

Instead of summing our depression variable, we can use SEM to run the mediation model from above but use the latent variable of depression instead.

```
## Our model
model3 <- '
  dep1 =~ dpq010 + dpq020 + dpq030 + dpq040 + dpq050 + dpq060 + dpq070 + dpq080 + dpq090
  dep1 ~ a*asthma
  sed_hr ~ b*dep1 + c*asthma

  indirect := a*b
  total := c + a*b
'
## sem function to run the model
fit3 <- sem(model3, data = df)
summary(fit3)
```

```
## lavaan 0.6-3 ended normally after 47 iterations
##
##   Optimization method                           NLMINB
##   Number of free parameters                         22
##
##                                                   Used       Total
##   Number of observations                          4614        4632
##
##   Estimator                                         ML
##   Model Fit Test Statistic                    1065.848
##   Degrees of freedom                                43
##   P-value (Chi-square)                           0.000
##
## Parameter Estimates:
##
##   Information                                 Expected
##   Information saturated (h1) model          Structured
##   Standard Errors                             Standard
##
## Latent Variables:
##                    Estimate  Std.Err  z-value  P(>|z|)
##   dep1 =~
##     dpq010            1.000
##     dpq020            1.096    0.024   45.136    0.000
##     dpq030            1.133    0.031   36.908    0.000
##     dpq040            1.149    0.030   38.066    0.000
##     dpq050            0.933    0.025   36.773    0.000
##     dpq060            0.929    0.022   42.107    0.000
##     dpq070            0.871    0.022   39.760    0.000
##     dpq080            0.686    0.019   36.325    0.000
##     dpq090            0.308    0.011   28.544    0.000
##
## Regressions:
##                    Estimate  Std.Err  z-value  P(>|z|)
##   dep1 ~
##     asthma     (a)    0.173    0.023    7.656    0.000
##   sed_hr ~
```

```
##     dep1       (b)    0.342    0.105    3.275    0.001
##     asthma     (c)    0.418    0.139    2.998    0.003
##
## Variances:
##                    Estimate  Std.Err  z-value  P(>|z|)
##    .dpq010            0.306    0.007   42.008    0.000
##    .dpq020            0.212    0.006   37.549    0.000
##    .dpq030            0.559    0.013   43.807    0.000
##    .dpq040            0.514    0.012   43.302    0.000
##    .dpq050            0.384    0.009   43.862    0.000
##    .dpq060            0.221    0.005   40.808    0.000
##    .dpq070            0.249    0.006   42.420    0.000
##    .dpq080            0.217    0.005   44.038    0.000
##    .dpq090            0.090    0.002   46.106    0.000
##    .sed_hr           11.179    0.233   48.012    0.000
##    .dep1              0.256    0.010   24.657    0.000
##
## Defined Parameters:
##                    Estimate  Std.Err  z-value  P(>|z|)
##     indirect          0.059    0.020    3.019    0.003
##     total             0.477    0.138    3.448    0.001
```

We defined `dep1` as a latent variable using `=~`. Although the model does not fit the data well–"`P-value (Chi-square) = 0.000`"–it is informative for demonstration. We would likely need to find out how the measurement model (`dep1 =~ dpq010 + dpq020 + dpq030 +`) actually fits before throwing it into a mediation model. We can do that via:

```
model4 <- '
  dep1 =~ dpq010 + dpq020 + dpq030 + dpq040 + dpq050 + dpq060 + dpq070 + dpq080 + dpq090
'
fit4 <- cfa(model4, data=df)
summary(fit4)
```

```
## lavaan 0.6-3 ended normally after 29 iterations
##
##   Optimization method                           NLMINB
##   Number of free parameters                         18
##
##   Number of observations                          4632
##
##   Estimator                                         ML
##   Model Fit Test Statistic                     985.831
##   Degrees of freedom                                27
##   P-value (Chi-square)                           0.000
##
## Parameter Estimates:
##
##   Information                                 Expected
##   Information saturated (h1) model          Structured
##   Standard Errors                             Standard
##
## Latent Variables:
##                    Estimate  Std.Err  z-value  P(>|z|)
##   dep1 =~
##     dpq010            1.000
```

```
##     dpq020            1.097    0.024   45.383    0.000
##     dpq030            1.128    0.031   36.962    0.000
##     dpq040            1.145    0.030   38.136    0.000
##     dpq050            0.927    0.025   36.630    0.000
##     dpq060            0.930    0.022   42.294    0.000
##     dpq070            0.870    0.022   39.941    0.000
##     dpq080            0.681    0.019   36.350    0.000
##     dpq090            0.307    0.011   28.609    0.000
##
## Variances:
##                   Estimate  Std.Err  z-value  P(>|z|)
##    .dpq010            0.306    0.007   42.051    0.000
##    .dpq020            0.211    0.006   37.470    0.000
##    .dpq030            0.560    0.013   43.909    0.000
##    .dpq040            0.515    0.012   43.400    0.000
##    .dpq050            0.390    0.009   44.041    0.000
##    .dpq060            0.221    0.005   40.835    0.000
##    .dpq070            0.249    0.006   42.461    0.000
##    .dpq080            0.216    0.005   44.149    0.000
##    .dpq090            0.090    0.002   46.195    0.000
##     dep1              0.261    0.011   24.765    0.000
```

As we can see, there is a lack of fit in the measurement model. It is possible that these depression questions could be measuring more than one factor. We could explore this using exploratory factor analysis. We don't demonstrate that here, but know that it is possible to do in `R` with a few other packages.

Machine Learning Techniques

We are briefly going to introduce some machine learning techniques that may be of interest to researchers. We will quickly introduce and demonstrate:

1. Ridge, Lasso and Elastic Net
2. Random Forests

In order to use these methods, we can use the fantastic `caret` package. It allows us to do nearly any type of machine learning technique. It is a type of package that takes many other packages and gives us a simple syntax across all the methods.

Ridge, Lasso and Elastic Net

Lasso and elastic net can do variable selection in addition to estimation. Ridge is great at handling correlated predictors. Each of them are better than conventional methods at prediction and each of them can handle large numbers of predictors. To learn more see "Introduction to Statistical Learning" by Daniela Witten, Gareth James, Robert Tibshirani, and Trevor Hastie. A free PDF is available on their website.

To use the package, it wants the data in a very specific form. First, we need to remove any missingness. We use `na.omit()` to do this. We take all the predictors (without the outcome) and put it in a data matrix object. We only include a few for the demonstration but you can include *many* predictors. We name ours `X`. `Y` is our outcome.

```
df2 <- df %>%
  dplyr::select(riagendr, ridageyr, ridreth3, race, famsize, dep, asthma, sed_hr) %>%
  na.omit
```

Then we use the `train()` function to fit the different models. This function, by default, uses cross-

validation[21], which we don't discuss here, but it an important topic to become familiar with. Below we fit a model that is either a ridge, a lasso, or an elastic net model depending on the `alpha` level. This is done using the `method = "glmnet"` argument. We specify the model by the formula `sed_hr ~ .` which means we want `sed_hr` to be the outcome and all the rest of the variables to be predictors.

```
library(caret)

## Use 10-fold cross validation
fitControl <- trainControl(method = "cv",
                           number = 10)

## Run the model
fit <- train(sed_hr ~ .,
             method = "glmnet",
             data = df2,
             trControl = fitControl)
fit
```

```
## glmnet
##
## 4437 samples
##    7 predictor
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 3993, 3993, 3995, 3992, 3992, 3994, ...
## Resampling results across tuning parameters:
##
##   alpha  lambda    RMSE  Rsquared  MAE
##   0.10   0.000829  3.28  0.0430    2.69
##   0.10   0.008293  3.28  0.0427    2.69
##   0.10   0.082933  3.28  0.0422    2.69
##   0.55   0.000829  3.28  0.0429    2.69
##   0.55   0.008293  3.28  0.0422    2.69
##   0.55   0.082933  3.29  0.0414    2.70
##   1.00   0.000829  3.28  0.0429    2.69
##   1.00   0.008293  3.28  0.0422    2.69
##   1.00   0.082933  3.29  0.0403    2.71
##
## RMSE was used to select the optimal model using the smallest value.
## The final values used for the model were alpha = 0.1 and lambda = 0.000829.
```

With this model, we can assess the most important predictors of sedentary behavior. We can do that with:

```
varImp(fit)
```

```
## glmnet variable importance
##
##                  Overall
## raceOther         100.00
## raceBlack          83.52
## raceWhite          80.22
```

[21] Cross-validation is a common way to reduce over-fitting and make sure your model is generalizable. Generally, you split your data into training and testing sets. It is very common in machine learning and is beginning to be practiced in academic fields as well. We recommend using it as often as you can, especially with these methods but also to make sure your other models are accurate on new data as well.

```
## raceOtherHispanic   28.12
## asthmaAsthma        10.87
## ridreth3             5.47
## riagendr             4.81
## famsize              4.54
## dep                  1.37
## ridageyr             0.00
```

This shows us that, of these variables, `race` was most important, followed by `asthma`. Importantly, though, this model did not predict the outcome very well so these are likely not very important predictors overall.

Random Forests

Random forests is another machine learning method that can do fantastic prediction. It is built in a very different way than the methods we have discussed up to this point. It is not built on a linear modeling scheme; rather, it is built on classification and regression trees (CART). Again, "Introduction to Statistical Learning" is a great resource to learn more.

Conveniently, we can use the `randomForest` package. (We can also use the `caret` package here.) We specify the model by the formula `sed_hr ~ .` just like before, which means we want `sed_hr` to be the outcome and all the rest of the variables to be predictors.

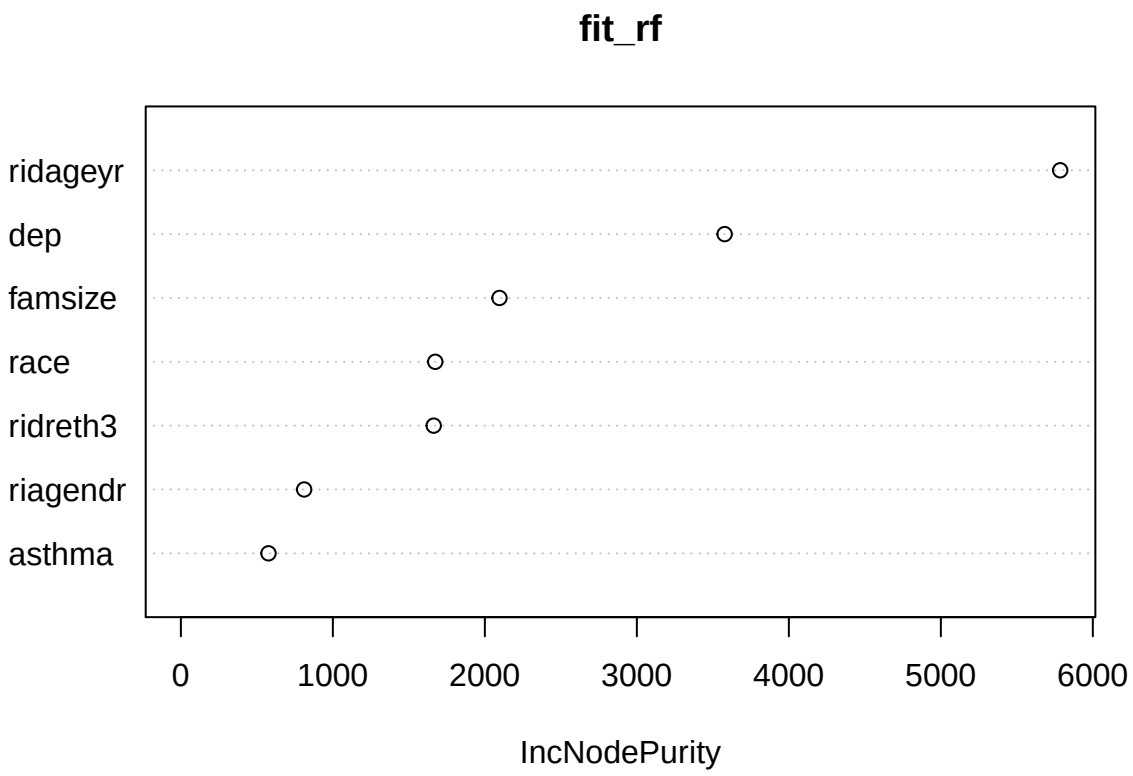
```
library(randomForest)

fit_rf <- randomForest(sed_hr ~ ., data = df2)
fit_rf
```

```
##
## Call:
##  randomForest(formula = sed_hr ~ ., data = df2)
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 2
##
##           Mean of squared residuals: 10.8
##                     % Var explained: 3.92
```

We can find out which variables were important in the model via:

```
par(mfrow=c(1,1))  ## back to one plot per page
varImpPlot(fit_rf)
```



We can see that age (`ridageyr`) is the most important variable, depression (`dep`) follows, with the family size (`famsize`) the third most important in the random forests model.

Apply It

This link contains a folder complete with an Rstudio project file, an RMarkdown file, and a few data files. Download it and unzip it to do the following steps.

Step 1

Open the `Chapter4.Rproj` file. This will open up RStudio for you.

Step 2

Once RStudio has started, in the panel on the lower-right, there is a `Files` tab. Click on that to see the project folder. You should see the data files and the `Chapter4.Rmd` file. Click on the `Chapter4.Rmd` file to open it. In this file, import the data and run each type of statistical analysis presented in this chapter (there are others that are presented in Chapters 4, 5, and 6 that you may have done already).

Once that code is in the file, click the `knit` button. This will create an HTML file with the code and output knitted together into one nice document. This can be read into any browser and can be used to show your work in a clean document.

Conclusions

Although we only discussed these methods briefly, that does not mean they are less important. On the contrary, they are essential upper level statistical methods. This brief introduction hopefully helped you know what `R` is capable of across a wide range of methods.

The next chapter begins our "advanced" topics, starting with "Advanced Data Manipulation".

Chapter 8: Advanced Data Manipulation

"Every new thing creates two new questions and two new opportunities." — Jeff Bezos

There's so much more we can do with data in `R` than what we've presented. Two main topics we need to clarify here are:

1. How do you reshape your data from wide to long form or vice versa in more complex data structures?
2. How do can we automate tasks that we need done many times?

We will introduce both ideas to you in this chapter. To discuss the first, show the use of `long()` and `wide()` from the `furniture` package. For the second, we need to talk about loops. Looping, for our purposes, refers to the ability to repeat something across many variables or data sets. There's many ways of doing this but some are better than others. For looping, we'll talk about:

1. vectorized functions,
2. `for` loops, and
3. the `apply` family of functions.

Reshaping Your Data

We introduced you to wide form and long form of your data in Chapter 2. In reality, data can take on nearly infinite forms but for most data in health, behavioral, and social science, these two forms are sufficient to know.

In some situations, your data may have multiple variables with multiple time points (known as time-variant variables) and other variables that are not (known as time-invariant variables) as shown:

```
##    ID Var_Time1 Var_Time2 Var2_Time1 Var2_Time2   Var3
## 1   1   -1.1602    0.1222     0.4078     0.1336 -0.622
## 2   2   -1.2247    0.8888    -0.8842     1.2115 -1.255
## 3   3   -0.0127    0.0884     0.7948    -0.0148  0.914
## 4   4    0.2099    0.5142    -0.3081     1.1523 -0.384
## 5   5   -1.2165    0.4719     0.3516    -0.6163 -0.412
## 6   6   -0.6803    0.8907    -0.4517     0.2043 -0.177
## 7   7   -1.0327    0.4296    -0.6858    -0.2949 -0.161
## 8   8   -1.4285    0.7404    -0.8492    -0.3095 -0.534
## 9   9   -0.0274    0.1439     0.0958     0.8501  2.072
## 10 10   -0.2369    0.1958     0.6369     1.1083  0.291
```

Notice that this data frame is in wide format (each ID is one row and there are multiple times or measurements per person for two of the variables). To change this to wide format, we'll use `long()`. The first argument is the data.frame, followed by two variable names (names that we go into the new long form), and then the numbers of the columns that are the measures (e.g., `Var_Time1` and `Var_Time2`).

```
long_form <- furniture::long(d1,
                             c("Var_Time1", "Var_Time2"),
                             c("Var2_Time1", "Var2_Time2"),
                             v.names = c("Var", "Var2"))
```

```
## id = ID
```

```
long_form
```

```
##       ID   Var3 time     Var    Var2
## 1.1    1 -0.622    1 -1.1602  0.4078
## 2.1    2 -1.255    1 -1.2247 -0.8842
## 3.1    3  0.914    1 -0.0127  0.7948
## 4.1    4 -0.384    1  0.2099 -0.3081
## 5.1    5 -0.412    1 -1.2165  0.3516
## 6.1    6 -0.177    1 -0.6803 -0.4517
## 7.1    7 -0.161    1 -1.0327 -0.6858
## 8.1    8 -0.534    1 -1.4285 -0.8492
## 9.1    9  2.072    1 -0.0274  0.0958
## 10.1 10  0.291    1 -0.2369  0.6369
## 1.2    1 -0.622    2  0.1222  0.1336
## 2.2    2 -1.255    2  0.8888  1.2115
## 3.2    3  0.914    2  0.0884 -0.0148
## 4.2    4 -0.384    2  0.5142  1.1523
## 5.2    5 -0.412    2  0.4719 -0.6163
## 6.2    6 -0.177    2  0.8907  0.2043
## 7.2    7 -0.161    2  0.4296 -0.2949
## 8.2    8 -0.534    2  0.7404 -0.3095
## 9.2    9  2.072    2  0.1439  0.8501
## 10.2 10  0.291    2  0.1958  1.1083
```

As you can see, it took the variable names and put that in our first variable that we called "measures". The actual values of the variables are now in the variable we called "values". Finally, notice that each ID now has two rows (one for each measure).

To go in the opposite direction (long to wide) we can use the `wide()` function. All we do is provide the long formed data frame, variables that are time-varying (`Var1` and `Var2`) and the variable showing the time points (`time`).

```
wide_form <- furniture::wide(long_form,
                             v.names = c("Var", "Var2"),
                             timevar = "time")
```

```
## id = ID
```

```
wide_form
```

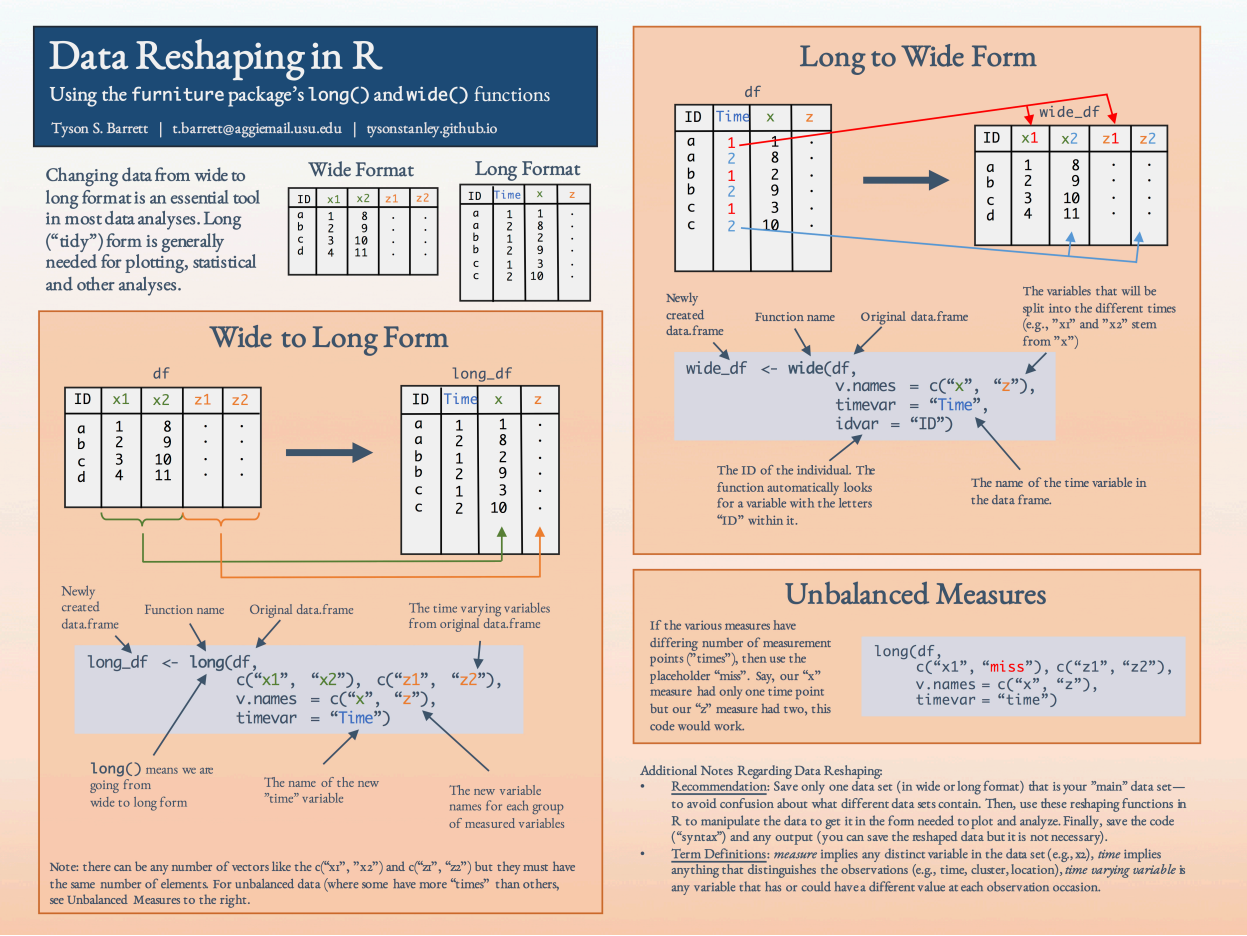
```
##       ID   Var3   Var.1  Var2.1  Var.2  Var2.2
## 1.1    1 -0.622 -1.1602  0.4078 0.1222  0.1336
## 2.1    2 -1.255 -1.2247 -0.8842 0.8888  1.2115
## 3.1    3  0.914 -0.0127  0.7948 0.0884 -0.0148
## 4.1    4 -0.384  0.2099 -0.3081 0.5142  1.1523
## 5.1    5 -0.412 -1.2165  0.3516 0.4719 -0.6163
## 6.1    6 -0.177 -0.6803 -0.4517 0.8907  0.2043
## 7.1    7 -0.161 -1.0327 -0.6858 0.4296 -0.2949
## 8.1    8 -0.534 -1.4285 -0.8492 0.7404 -0.3095
## 9.1    9  2.072 -0.0274  0.0958 0.1439  0.8501
```

```
## 10.1 10  0.291 -0.2369  0.6369 0.1958  1.1083
```

And we are back to the wide form.

These steps can be followed for situations where there are many measures per person, many people per cluster, etc. In most cases, this is the way multilevel data analysis occurs (as we discussed in Chapter 6) and is a nice way to get our data ready for plotting.

The following figure shows the features of both `long()` and `wide()`.



Repeating Actions (Looping)

To fully go into looping, understanding how to write your own functions is needed.

Your Own Functions

Let's create a function that estimates the mean (although it is completely unnecessary since there is already a perfectly good `mean()` function).

```
mean2 <- function(x){
  n <- length(x)
  m <- (1/n) * sum(x)
  return(m)
}
```

We create a function using the `function()` function.[22] Within the `function()` we put an `x`. This is the argument that the function will ask for. Here, it is a numeric vector that we want to take the mean of. We then provide the meat of the function between the `{}`. Here, we did a simple mean calculation using the `length(x)` which gives us the number of observations, and `sum()` which sums the numbers in `x`.

Let's give it a try:

```
v1 <- c(1,3,2,4,2,1,2,1,1,1)   ## vector to try
mean2(v1)                      ## our function
```

```
## [1] 1.8
```

```
mean(v1)                       ## the base R function
```

```
## [1] 1.8
```

Looks good! These functions that you create can do whatever you need them to (within the bounds that `R` can do). I recommend by starting outside of a function that then put it into a function. For example, we would start with:

```
n <- length(v1)
m <- (1/n) * sum(v1)
m
```

```
## [1] 1.8
```

and once things look good, we would put it into a function like we had before with `mean2`. It is an easy way to develop a good function and test it while developing it.

By creating your own function, you can simplify your workflow and can use them in loops, the `apply` functions and the `purrr` package.

For practice, we will write one more function. Let's make a function that takes a vector and gives us the N, the mean, and the standard deviation.

```
important_statistics <- function(x, na.rm=FALSE){
  N  <- length(x)
  M  <- mean(x, na.rm=na.rm)
  SD <- sd(x, na.rm=na.rm)

  final <- c(N, M, SD)
  return(final)
}
```

One of the first things you should note is that we included a second argument in the function seen as `na.rm=FALSE` (you can have as many arguments as you want within reason). This argument has a default that we provide as `FALSE` as it is in most functions that use the `na.rm` argument. We take what is provided in the `na.rm` and give that to both the `mean()` and `sd()` functions. Finally, you should notice that we took several pieces of information and combined them into the `final` object and returned that.

Let's try it out with the vector we created earlier.

```
important_statistics(v1)
```

```
## [1] 10.00  1.80  1.03
```

Looks good but we may want to change a few aesthetics. In the following code, we adjust it so we have each one labeled.

[22] That seemed like excessive use of the word function… It is important though. So, get used to it!

```
important_statistics2 <- function(x, na.rm=FALSE){
  N  <- length(x)
  M  <- mean(x, na.rm=na.rm)
  SD <- sd(x, na.rm=na.rm)

  final <- data.frame(N, "Mean"=M, "SD"=SD)
  return(final)
}
important_statistics2(v1)
```

```
##    N Mean   SD
## 1 10  1.8 1.03
```

We will come back to this function and use it in some loops and see what else we can do with it.

Vectorized

By construction, `R` is the fastest when we use the vectorized form of doing things. For example, when we want to add two variables together, we can use the `+` operator. Like most functions in `R`, it is vectorized and so it is fast. Below we create a new vector using the `rnorm()` function that produces normally distributed random variables. First argument in the function is the length of the vector, followed by the mean and SD.

```
v2 <- rnorm(10, mean=5, sd=2)
add1 <- v1 + v2
round(add1, 3)
```

```
##  [1]  7.21 11.76  6.38  8.73  7.02  4.92  7.60  4.24  6.27  5.93
```

We will compare the speed of this to other ways of adding two variables together and see it is the simplest and quickest.

For Loops

For loops have a bad reputation in the `R` world. This is because, in general, they are slow. It is among the slowest of ways to iterate (i.e., repeat) functions. We start here to show you, in essence, what the `apply` family of functions are doing, often, in a faster way.

At times, it is easiest to develop a for loop and then take it and use it within the `apply` or `purrr` functions. It can help you think through the pieces that need to be done in order to get your desired result.

For demonstration, we are using the `for` loop to add two variables together. The code between the `()`'s tells `R` information about how many loops it should do. Here, we are looping through `1:10` since there are ten observations in each vector. We could also specify this as `1:length(v1)`. When using `for` loops, we need to keep in mind that we need to initialize a variable in order to use it within the loop. That's precisely what we do with the `add2`, making it a numberic vector with 10 observations.

```
add2 <- vector("numeric", 10)   ## Initialize
for (i in 1:10){
  add2[i] <- v1[i] + v2[i]
}
round(add2, 3)
```

```
##  [1]  7.21 11.76  6.38  8.73  7.02  4.92  7.60  4.24  6.27  5.93
```

Same results! But, we'll see later that the speed is much than the vectorized function.

The apply family

The apply family of functions that we'll introduce are:

1. apply()
2. lapply()
3. sapply()
4. tapply()

Each essentially do a loop over the data you provide using a function (either one you created or another). The different versions are extremely similar with some minor differences. For apply() you tell it if you want to iterative over the columns or rows; lapply() assumes you want to iterate over the columns and outputs a list (hence the l); sapply() is similar to lapply() but outputs vectors and data frames. tapply() has the most differences because it can iterative over columns by a grouping variable. We'll show apply(), lapply() and tapply() below.

For example, we can add two variables together here. We provide it the data.frame that has the variables we want to add together.

```
df <- data.frame(v1, v2)
add3 <- apply(df, 1, sum)
round(add3, 3)
```

```
##  [1]  7.21 11.76  6.38  8.73  7.02  4.92  7.60  4.24  6.27  5.93
```

The function apply() has three main arguments: a) the data.frame or list of data, b) 1 meaning to apply the function for each row or 2 to the columns, and c) the function to use.

We can also use one of our own functions such as important_statistics2() within the apply family.

```
lapply(df, important_statistics2)
```

```
## $v1
##    N Mean   SD
## 1 10  1.8 1.03
##
## $v2
##    N Mean  SD
## 1 10  5.2 1.5
```

This gives us a list of two elements, one for each variable, with the statistics that our function provides. With a little adjustment, we can make this into a data.frame using the do.call() function with "rbind".

```
do.call("rbind", lapply(df, important_statistics2))
```

```
##     N Mean   SD
## v1 10  1.8 1.03
## v2 10  5.2 1.50
```

tapply() allows us to get information by a grouping factor. We are going to add a factor variable to the data frame we are using df and then get the mean of the variables by group.

```
group1 <- factor(sample(c(0,1), 10, replace=TRUE))
tapply(df$v1, group1, mean)
```

```
##   0   1
## 1.8 1.8
```

We now have the means by each group. This, however, is probably replaced by the 3 step summary that we learned earlier in dplyr using group_by() and summarize().

These functions are useful in many situations, especially where there are no vectorized functions. You can always get an idea of whether to use a `for` loop or an `apply` function by giving it a try on a small subset of data to see if one is better and/or faster.

Speed Comparison

We can test to see how fast functions are with the `microbenchmark` package. Since it wants functions, we will create a function that uses the `for` looop.

```
forloop <- function(var1, var2){
  add2 <- vector("numeric", length(var1))
  for (i in 1:10){
    add2[i] <- var1[i] + var2[i]
  }
  return(add2)
}
```

Below, we can see that the vectorized version is nearly 50 times faster than the `for` loop and 300 times faster than the `apply`. Although the `for` loop was faster here, sometimes it can be slower than the `apply` functions–it just depends on the situation. But, the vectorized functions will almost always be *much* faster than anything else. It's important to note that the `+` is also a function that can be used as we do below, highlighting the fact that anything that does something to an object in `R` is a function.

```
library(microbenchmark)
microbenchmark(forloop(v1, v2),
               apply(df, 1, sum),
               `+`(v1, v2))
```

```
## Unit: nanoseconds
##               expr   min    lq  mean median    uq     max neval cld
##    forloop(v1, v2)  1735  2126 49543   2407  2654 4714953   100   a
##  apply(df, 1, sum) 67726 70314 75627  71432 75688  190306   100   a
##            v1 + v2   169   242   386    350   448    2799   100   a
```

Of course, as it says the units are in nanoseconds. Whether a function takes 200 or 200,000 nanoseconds probably won't change your life. However, if the function is being used repeatedly or on on large data sets, this can make a difference.

Using "Anonymous Functions" in Apply

Last thing to know here is that you don't need to create a named function everytime you want to use `apply`. We can use what is called "Anonymous" functions. Below, we use one to get at the N and mean of the data.

```
lapply(df, function(x) rbind(length(x), mean(x, na.rm=TRUE)))
```

```
## $v1
##      [,1]
## [1,] 10.0
## [2,]  1.8
##
## $v2
##      [,1]
## [1,] 10.0
## [2,]  5.2
```

So we don't name the function but we design it like we would a named function, just minus the `return()`. We take `x` (which is a column of `df`) and do `length()` and `mean()` and bind them by rows. The first argument in the anonymous function will be the column or variable of the data you provide.

Here's another example:

```
lapply(df, function(y) y * 2 / sd(y))
```

```
## $v1
##  [1] 1.94 5.81 3.87 7.75 3.87 1.94 3.87 1.94 1.94 1.94
##
## $v2
##  [1]  8.25 11.65  5.82  6.29  6.67  5.20  7.45  4.30  7.00  6.55
```

We take `y` (again, the column of `df`), times it by two and divide by the standard deviation of `y`. Note that this is gibberish and is not some special formula, but again, we can see how flexible it is.

The last two examples also show something important regarding the output:

1. The output will be at the level of the anonymous function. The first had two numbers per variable because the function produced two summary statistics for each variable. The second we multiplied `y` by 2 (so it is still at the individual observation level) and then divide by the SD. This keeps it at the observation level so we get ten values for every variable.
2. We can name the argument anything we want (as long as it is one word). We used `x` in the first and `y` in the second but as long as it is the same within the function, it doesn't matter what you use.

Finally, we may not want our variables to be in the list format. We may want to control more tightly what is outputted from the looping. For that, we can thank the `purrr` package (part of the `tidyverse`; note the three r's in purrr). This package provides many valuable functions that you can explore. Of particular mention here, though, are some of the `map*()` functions that work just like `lapply()`.

1. `map()` – outputs a list
2. `map_df()` – outputs a data frame
3. `map_if()` – outputs a list but only makes any changes to the variables that meet a condition (e.g., `is.numeric()`).

```
purrr::map(df, function(y) y * 2 / sd(y))
```

```
## $v1
##  [1] 1.94 5.81 3.87 7.75 3.87 1.94 3.87 1.94 1.94 1.94
##
## $v2
##  [1]  8.25 11.65  5.82  6.29  6.67  5.20  7.45  4.30  7.00  6.55
```

```
purrr::map_df(df, function(y) y * 2 / sd(y))
```

```
## # A tibble: 10 x 2
##       v1    v2
##    <dbl> <dbl>
##  1  1.94  8.25
##  2  5.81 11.6
##  3  3.87  5.82
##  4  7.75  6.29
##  5  3.87  6.67
##  6  1.94  5.20
##  7  3.87  7.45
##  8  1.94  4.30
##  9  1.94  7.00
## 10  1.94  6.55
```

```
purrr::map_if(df, is.numeric, function(y) y * 2 / sd(y))
```

```
## $v1
```

```
##  [1] 1.94 5.81 3.87 7.75 3.87 1.94 3.87 1.94 1.94 1.94
##
## $v2
##  [1]  8.25 11.65  5.82  6.29  6.67  5.20  7.45  4.30  7.00  6.55
```

Apply It

This link contains a folder complete with an Rstudio project file, an RMarkdown file, and a few data files. Download it and unzip it to do the following steps.

Step 1

Open the `Chapter8.Rproj` file. This will open up RStudio for you.

Step 2

Once RStudio has started, in the panel on the lower-right, there is a `Files` tab. Click on that to see the project folder. You should see the data files and the `Chapter8.Rmd` file. Click on the `Chapter8.Rmd` file to open it. In this file, import the data, reshape it to long form, create your own function to do something for you, and apply the function in a loop over some of the variables of the data set.

Once that code is in the file, click the `knit` button. This will create an HTML file with the code and output knitted together into one nice document. This can be read into any browser and can be used to show your work in a clean document.

Conclusions

These are useful tools to use in your own data manipulation beyond that what we discussed with `dplyr`. It takes time to get used to making your own functions so be patient with yourself as you learn how to get `R` to do exactly what you want in a condensed, replicable format.

With these new tricks up your sleeve, we can move on to more advanced plotting using `ggplot2`.

Chapter 9: Reproducible Workflow with RMarkdown

> "The commonality between science and art is in trying to see profoundly - to develop strategies of seeing and showing." — Edward Tufte

Recently, researchers across the health, behavioral, and social sciences have become increasingly concerned with the reproducibility of research. The concern ranges from asserting that "most claimed research findings are false" (Ioannidis 2005, pg. 696) to "we need to make substantial changes to how we conduct research," (Cumming 2014, abstract). Some have come to refer to the situation as a "reproducibility crisis" (Begley and Ioannidis 2015; Taylor and Tibshirani 2015; Munafò et al. 2017).

The term "reproducibility" (or "replicability") is defined in various ways (Goodman, Fanelli, and Ioannidis 2016) but we'll stick with the definition provided by Goodman, Fanelli, and Ioannidis (2016).

1. *Methods Reproducibility* "refers to the provision of enough detail about study procedures and data so the same procedures could … be exactly repeated" (pg. 2) with the same data,
2. *Results Reproducibility* "refers to obtaining the same results from the conduct of an independent study whose procedures are as closely matched to the original experiment as possible" (pg. 2-3) with independent data, and
3. *Inferential Reproducibility* "refers to the drawing of qualitatively similar conclusions from either an independent replication of a study or a reanalysis of the original study" (pg. 4).

Of these, we are most interested in *methods reproducibility* for the purposes of this chapter. That is, we will discuss how `R` can help you, as a researcher, improve in this aspect of reproducibility. Because of this, this chapter is unique. It is devoted to the `R` approach to a reproducible workflow that keeps the methods and data intimately tied to the communication of the study. Instead of talking about `R` code, we will be showing how we can combine code, output, and regular text into one reproducible document.

"What?!" say you. "How can something so magical be possible?"

Well, its true and we can thank two important packages for this gift: `rmarkdown` and `knitr`. These allow us to use a simple type-setting approach called Markdown in conjunction with `R` code to produce beautiful documents.

Notably, some of this you have encountered if you've done any of the application parts of the chapters so hopefully this fills in some gaps about your experience with it.

R Markdown

Markdown is a simple type-setting language where certain symbols change how text will end up looking. A great cheatsheet shows which symbols do what. Herein, we will discuss the most important ones.

To start, we will start at the top of a `.Rmd` (R Markdown) document. This top part is called the YAML (stands for Yet Another Markup Language). This part controls the overall features of the document, including

what type you produce (an HTML document that can be displayed on all major browsers, a Microsoft Word document, or a PDF if we have LaTeX installed on our computer), the font size and style, title, author, etc.

The YAML cares about spacing (unlike `R` code) and requires a specific format. As you get more familiar with R Markdown, this will become more intuitive. For now, the main pieces we'll be using looks like this:

```
---
title: "The Title"
author: "You, the Author"
output: html_document
---
```

The `output:` argument is where we tell R Markdown what type of file we want to have produced. The most flexible type, requiring no set-up, is the `html_document`. For this book, this is what we will focus on.

> At the end of the chapter I introduce the other formats, and show several packages that provide pre-formatted documents that can be submitted at various journals or can be used to create theses and dissertations.

After the bottom `---`, our R Markdown document actually starts. From here, we can use regular text, `R` code, and Markdown symbols to control what is shown and how.

We can control headers, bolding, and italics using Markdown symbols, as shown below.

```
# Header 1
## Header 2

This is regular text.
We can **bold** text using two stars.
We can *italicize* using one star.
```

We can also include `R` code directly in the text that will be evaluated within the document and produce output that will be printed out in the document. That means we can print tables and figures directly from `R` code without dragging and dropping figures or typing up complex tables. Furthermore, it means that if something changes in the analyses, the tables and figures will automatically update; no manual redoing of tables!

To include an `R` code "chunk", we use the following:

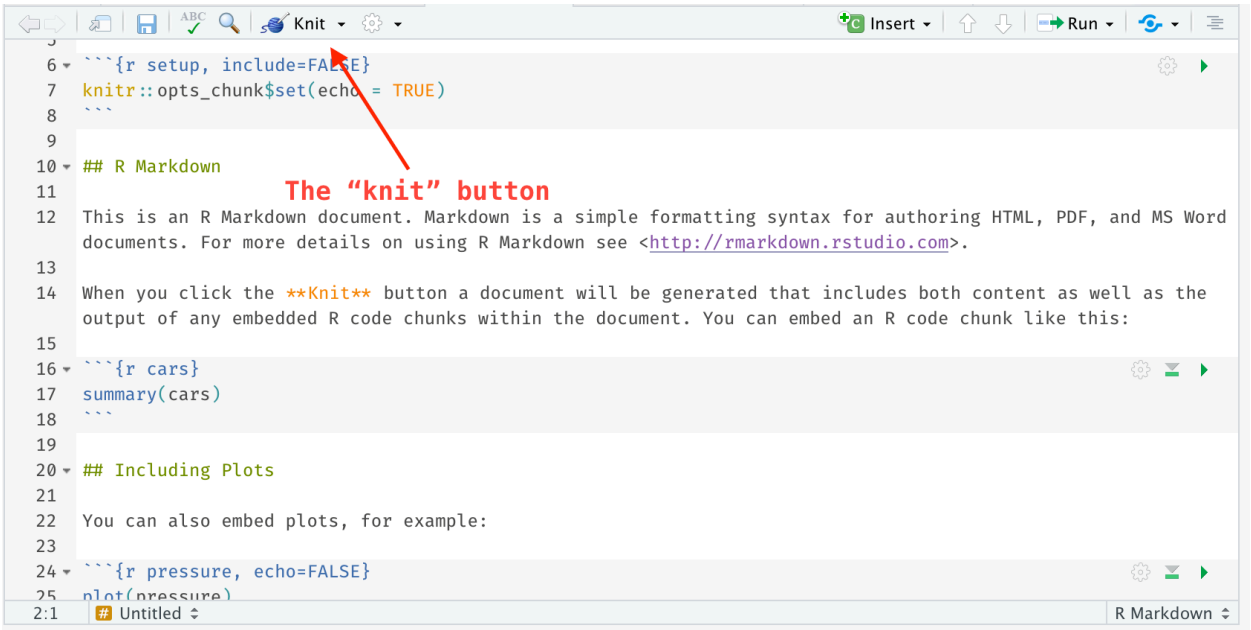
````
```{r}
## R Code goes here
```
````

These can be added automatically by clicking the "Insert" button (see top-right of the image below).

A chunk will be evaluated just like normal `R` code so everything we've learned in previous chapters applies. If there is output printed in the chunk (e.g., model output, a table, a `ggplot2` figure), this will show up in the document.

But, say you, all I see is the code and text I've written. Where is this magical document that is formatted and has output all beautifully intertwined? Great question. This is something that most of us are not used to. For those that have used other word processors (like Microsoft Word, Apple's Pages) you are using what is known as "WYSIWYG" (What You See Is What You Get). That is, you see how the document is going to look printed out while you type. This can have some advantages but it also can be very limiting.

R Markdown, on the other hand, must be "knit" to see what the formatted and evaluated document will look like. In RStudio, this is simple. There is a button at the top left of the scripting panel whenever an .Rmd (R Markdown) document is being displayed. By selecting this, our document will be created.



As we write, we can “knit” whenever we want to see what the document is looking like. Regardless of the type of document we choose, RStudio will find a way to show us the document. For example, if we choose `html_document`, a browser either within RStudio or a pop-up window will show the document.

With just this information, you can start writing reproducible documents. Congrats! Below I highlight a few additional features (among many, many others).

In-Line R Code

Beneficially, we can include in-line `R` code. Say we ran a model and we want to report the coefficient and p-value in the text. We can do this manually by just typing it in. But this comes with some problems if we update our data or analyses, as these values can change. If we aren't careful, we can miss that and mis-report the updated values. Luckily, inline `R` code is pretty straightforward.

Let's start with a quick model that we want to write about. We'll use a form of a model we ran in Chapter 4, using minority status and marital status as a predictor of family size.

```
df$minority <- factor(ifelse(df$race == "White", 0, 1),
                      labels = c("White", "Minority"))
fit_reg <- lm(famsize ~ minority + marriage, df)
summary(fit_reg)
```

```
##
## Call:
## lm(formula = famsize ~ minority + marriage, data = df)
##
## Residuals:
##    Min     1Q Median     3Q    Max
##  -2.34  -1.10  -0.34   0.88   4.36
##
## Coefficients:
##                   Estimate Std. Error t value Pr(>|t|)
## (Intercept)         2.8792     0.0381   75.48  < 2e-16 ***
## minorityMinority    0.4604     0.0421   10.93  < 2e-16 ***
## marriage2          -1.2353     0.0763  -16.19  < 2e-16 ***
## marriage3          -1.2195     0.0686  -17.79  < 2e-16 ***
```

```
## marriage4         -0.6341     0.1134   -5.59  2.4e-08 ***
## marriage5         -0.8840     0.0527  -16.76  < 2e-16 ***
## marriage6         -0.5277     0.0792   -6.66  3.0e-11 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.36 on 4447 degrees of freedom
##   (178 observations deleted due to missingness)
## Multiple R-squared:  0.139,  Adjusted R-squared:  0.138
## F-statistic:  120 on 6 and 4447 DF,  p-value: <2e-16
```

To make it easier to work with, we are going to use the `broom` package, which helps clean up the information that we'll want to report.

```
tidy_fit <- broom::tidy(fit_reg)
```

Let's say we want to report the coefficient and the p-value on the minority variable. We can grab the coefficient by using: `` `r tidy_fit$estimate[2]` ``. This will evaluate the expression `tidy_fit$estimate[2]` when we knit and it will print out the value. One problem that can happen is that this coefficient is seven decimal places; probably a bit too far to report in the text. So let's clean it up by doing: `` `r tidy_fit$estimate[2] %>% round(3)` ``, which rounds it to three decimal places. We may also want to format the p-value as well but let's say if it is below .001, let's report "p < .001". We can do this by first running the code in a code chunk:

```
formatted_p <- ifelse(tidy_fit$p.value < .001, ## Condition
                      "< .001",  ## if condition is true
                      paste("=", tidy_fit$p.value[2] %>% round(3))) ## if condition is false
```

and then referring to it inline as so: `` `r formatted_p[2]` ``.

Below, is an example of how we could report this.

> Controlling for marital status, minority status was associated with a family size `` `r tidy_fit$estimate[2] %>% round(3)` `` individuals larger than non-minority families (p `` `r formatted_p[2]` ``).

which prints as:

> Controlling for marital status, minority status was associated with a family size 0.46 individuals larger than non-minority families (p < .001).

Then, if we decide we want to change the covariates, or filter out extreme cases, these will automatically update to the most recent model.

Pre-formatted Documents

One of the main ideas is to be able to write without worrying about formatting. This is accomplished by using built-in formats provided by a number of great packages.

TABLE of PACKAGES HERE

Other Tricks

HTML code, figures, other knitr stuff

Important, Important

The document's code must be fully self-contained. That means, anything you want it to run has to be in the document, regardless of what you've already run outside of knitting. For example, if we are testing our

code and running it throughout, when we go to knit. It will re-run everything in the document and forget everything else you've done that is not in the document.

Some consequences of this:

1. If you don't include the code to read in the data set that you use in the document, the document won't knit. Why? Because it doesn't know where to get the data since you haven't told *the document* where the data set is at.
2. Errors anywhere in the code within the document will make it so it cannot knit. So if you have some experimental code that isn't working yet, this could cause problems. Luckily, in the code chunk you can tell the document to ignore the code using `eval = FALSE`:

```
```{r, eval = FALSE}
## R Code goes here
```
```

These are actually good things for you. This forces you to be *fully* reproducible in your work. R Markdown won't let you skip steps in the data analysis. Although working through bugs can sometimes be annoying, this will ultimately bless your research. For if you can't even reproduce your own research, then how can we expect any other researcher to reproduce it?

But this feature goes beyond that. In fact, it makes it so another research could reproduce your work by just downloading and running your R Markdown document. This removes all guesswork for others regarding your data analysis and reporting.

Going one step further would be to post your R Markdown document in a publically accessible repository, with (if possible) the data used in the R Markdown document. Although maybe intimidating showing others your code, this is actually an important step in making your research as reproducible as possible.

Apply It

This link contains a folder complete with an Rstudio project file, an RMarkdown file, and a few data files. Download it and unzip it to do the following steps.

Step 1

Open the `Chapter9.Rproj` file. This will open up RStudio for you.

Step 2

Once RStudio has started, in the panel on the lower-right, there is a `Files` tab. Click on that to see the project folder. You should see the data files and the `Chapter9.Rmd` file. Click on the `Chapter9.Rmd` file to open it. In this file, import the data and run each type of statistical analysis presented in this chapter.

Once that code is in the file, click the `knit` button. This will create an HTML file with the code and output knitted together into one nice document. This can be read into any browser and can be used to show your work in a clean document.

Chapter 10: Where to Go from Here and Common Pitfalls

> "The journey of a thousand miles begins with one step." — Lao Tzu

There are many resources that can aid in developing your R skills from here. We have introduced the basics of R, helping you take a few steps on your journey of understanding R. We have focused on the ones that are most important for researchers in the health, behavioral, and social sciences.

Since this has been a primer, we hope that you will continue your learning of R via the various sources available at little to no cost. Just like this book, many R books are available online as well as in print. This allows you to explore and learn online at your own pace without having to buy a bunch of books or other resources.

Below, we list a few R books that we have found to be useful. Most are available free in some form.

1. R for Data Science by Hadley Wickham and Garrett Grolemund
2. Efficient R Programming by Colin gillespie and Robin Lovelace
3. The R Cookbook
4. An Introduction to Statistical Learning
5. R Markdown: The Definitive Guide

There are *many, many* books that talk about R in various forms so by no means is this a complete list.

Common Pitfalls

To end, we wanted to highlight some pitfalls that can plague any beginner to R. We list a few that we've encountered, although others surely exist.

1. Document your work.
2. Avoid overriding objects unless it is on purpose. Changing objects can be hard to keep track of in bigger projects.
3. Ask questions. R is very flexible; this can make it overwhelming to learn since there are many ways to perform the same task. However, there are people who have figured out easy ways to do complex stuff and most are willing to answer an email.
4. Plan out the steps of your data manipulation and analyses. A few minutes of planning can help you not get lost in the technology and lose sight of the goal.
5. Understand the statistics before throwing data in a model. This can lead to major problems in science. At the very least, understand the assumptions of the modeling type and when it can and should be used.
6. Do exploratory data analysis (EDA) to understand your data. R is made for this–so use it. Otherwise, your model may be completely wrong and have many violated assumptions.
7. Be transparent in your writing. If you use the R scripts correctly, you can provide your code as part of any publication. This will greatly increase replicability of our important research findings.

Quiz

As a final note, we thought we would give you a quiz to test your memory of the topics we've covered. Don't worry; no pressure to get them all. We've included some tougher ones. Regardless of how well you do, we hope you'll continue improving in your `R` programming skills.

Question 1

What kind of vector is this?

```
x <- c(10.1, 2.1, 4.6, 2.3, 8.9)
```

Question 2

What does this line of code do?

```
df[c(1,5), c("B", "C")]
```

Question 3

In the `tidyverse` there are four join functions. What are they?

Question 4

What functions are used in the "three step summary" as described in Chapter 2?

Question 5

What does the following code do?

```
ggplot(df, aes(x=C, y=D)) +
  geom_boxplot(aes(color = C)) +
  theme_bw() +
  scale_color_manual(values = c("dodgerblue4", "coral2"))
```

Question 6

Name three functions you can use to summarize your data in an informative way.

Question 7

What type of model does `aov()` perform?

Question 8

What are the differences between `aov()` and `lm()`?

Question 9

What assumptions of normality and heteroskedasticity fail, what function can be used to fit logistic and poisson regressions?

Question 10

If you were trying to perform logistic regression, what arguments are necessary?

Question 11

In multilevel modeling, which functions can be used to fit a Generalized Estimating Equations model?

Question 12

When comparing mixed effects models, what does `anova()` do?

Question 13

Can `R` do structural equation modeling? If so, what package(s) are useful?

Question 14

What types of models can `glmnet()` perform? How can you do a cross-validated "glmnet" model?

Question 15

Is the following data in wide or long form? How do you know?

```
##    ID  measures  values
## 1   1 Var_Time1 -1.4943
## 2   2 Var_Time1  0.6251
## 3   3 Var_Time1 -0.7726
## 4   4 Var_Time1 -1.2863
## 5   5 Var_Time1 -0.1473
## 6   6 Var_Time1 -0.9322
## 7   7 Var_Time1 -2.1992
## 8   8 Var_Time1  0.7929
## 9   9 Var_Time1  0.2382
## 10 10 Var_Time1 -0.2740
## 11  1 Var_Time2  0.0347
## 12  2 Var_Time2  0.7321
## 13  3 Var_Time2  0.6670
## 14  4 Var_Time2  0.0182
## 15  5 Var_Time2  0.7547
## 16  6 Var_Time2  0.2919
## 17  7 Var_Time2  0.6848
## 18  8 Var_Time2  0.5682
## 19  9 Var_Time2  0.6049
## 20 10 Var_Time2  0.7544
```

To make your data long form but it is currently in wide form, what function(s) can you use?

Question 16

What form of looping is the fastest? What does `apply()` do? Can you do for loops in `R`?

Question 17

What is the following code doing?

```
sandwhich <- function(pb, jam){
  s <- pb + jam
  return(s)
}
```

Question 18

What is wrong with this chunk of code?

```
df <- df +
  mutate(newvar = ifelse(oldvar == 1, 1, 0))
```

Question 19

What is your favorite built in theme or how would you make your favorite custom theme?

Question 20

What kind of plot does the following make?

```
pos = position_dodge(width = .1)
ggplot(summed_data, aes(x = dep2, y = sed, group = asthma, color = asthma)) +
  geom_line(position = pos) +
  geom_errorbar(aes(ymin = sed - s_se, ymax = sed + s_se),
                width = .1,
                position = pos)
```

Goodbye and Good Luck

I hope this has been a useful primer to get you into R. If you still feel rusty, feel free to go through the book again or look at other online resources. R is very flexible and can ease the data and analysis burden of research. Implement good practices and your work will become easier to track, easier to document, and easier to communicate. Good luck on your journey using R in your research!

```
Step1 <- of_a_journey(you) %>%
  has(begun)
You <- now_have_seen(aspects, of, R) %>%
  that_can(increase) %>%
  productivity(your)
GoodLuck <- journey(on, your)
```

Begley, C. Glenn, and John P A Ioannidis. 2015. "Reproducibility in science: Improving the standard for basic and preclinical research." *Circulation Research* 116 (1): 116–26. https://doi.org/10.1161/CIRCRESAHA.114.303819.

Cumming. 2014. "The new statistics: Why and how." *Psychological Science* 25 (1): 7–29. https://doi.org/10.1177/0956797613504966.

Goodman, Steven N, Daniele Fanelli, and John P A Ioannidis. 2016. "What does research reproducibility mean?" *Science Translational Medicine* 8 (341): 1–6. https://doi.org/10.1126/scitranslmed.aaf5027.

Ioannidis, John P A. 2005. "Why most published research findings are false." *PLoS Medicine* 2 (8): 0696–701. https://doi.org/10.1371/journal.pmed.0020124.

Munafò, Marcus R, Brian A Nosek, Dorothy V M Bishop, Katherine S Button, Christopher D Chambers, Nathalie Percie du Sert, Uri Simonsohn, et al. 2017. "A manifesto for reproducible science." *Nature Human Behaviour* 1 (January): 1–9. https://doi.org/10.1038/s41562-016-0021.

Taylor, Jonathan, and Robert J Tibshirani. 2015. "Statistical learning and selective inference." *Proceedings of the National Academy of Sciences of the United States of America* 112 (25). https://doi.org/10.1073/pnas.1507583112.