TYSON S. BARRETT

# R FOR THE HEALTH, BEHAVIORAL, AND SOCIAL SCIENCES

# *Contents*

```
## furniture 1.5.4: learn more at tysonstanley.github.io
```

# *Preface*

"Somewhere, something incredible is waiting to be known."

— Carl Sagan

`R` is made for scientists pursuing discoveries; discoveries that cannot be solved with human intuition alone. `R` is capable of data analysis in nearly any form. It is an open-source project meaning it can be extended and fixed by countless individuals around the world. It is almost always on the cutting edge of new techniques and methods. Yet, many are hesitant to learn to use it.

This is probably due to the high learning curve. Although challenging, with a little guidance, the journey in learning `R` programming can be made much simpler. It is for this exact reason this book was written.

Although there are many books and websites devoted to R, I've noticed that a simple introduction—without excessive information on things that are unlikely to be used—was lacking, particularly for the health, behavioral, and social sciences. This book began as I, working as a data science and statistical consultant, was trying to help my clientele with quantitative research across these fields. Three main reasons kept me repeatedly recommending my clients to use `R`: the replicability of the `R` centered workflow, the powerful and beautiful plotting capabilities, and the simple—yet extensive—data management/analysis tools.

This book is for beginners in `R`; especially those in health, behavioral and social sciences. I picked those fields specifically because much of their research overlaps in data types, methodologies employed, and in hypotheses tested. Both exploratory and confirmatory methods will be highlighted given their importance in these fields, among many others.

I'll introduce you to many ways in which `R` can be used in your work. You'll find that it can help in all facets of your data analysis and communication, while improving your replicability. In the long run, taking some time to learn a new tool will save you time, energy, and probably most importantly, frustration. When a researcher is

frustrated, it becomes so easy to overlook important features.

We will quickly, and succinctly, introduce the newest, easiest, and most understandable ways of working with your data. To do this, we will have three main parts: 1) working with and simple analyses of your data, 2) modeling your data, and 3) more advanced techniques to help your workflow.

### *Part I*

1. Chapter 1: The basics of the language
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 (ANOVA, Linear Regression)
5. Chapter 5: Generalized Linear Models
6. Chapter 6: Mulilevel Modeling
7. Chapter 7: Other Modeling Techniques

### *Part III*

8. Chapter 8: Advanced data manipulation
9. Chapter 9: Advanced plotting
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.

### *Download R and RStudio*

To begin, you will need to download the `R` software www.r-project.org and then the RStudio software www.rstudio.com. `R` is the brains and `RStudio`[1] is an "IDE" (something that helps us work with `R` much more easily).

Once both are installed (helps on installing the software can be found on www.rstudio.com, `www.r-bloggers.com`, and `www.statmethods.net`) you are good to go. The remainder of the book will be about actually using it.

Enjoy![2]

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

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

# *Chapter 1: The Basics of the Language*

> "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.

The best way to begin to learn it is by jumping right into it. This chapter will provide the background and foundation to start using `R` right away. This background revolves around data and functions—two 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.[3] The main object types that you'll work with are presented in the following table.

| Object | Description |
|---|---|
| Vector | A single column of data ('a variable') |
| Matrix | An n x p table of numeric data |
| Data Frame | An n x p table of data (numeric and/or categorical) |
| Function | Takes input and produces output—the workhorse of R |
| Operator | A special type of function (e.g. '<-' |

Each of these objects will be introduced in this chapter, highlighting

[3] 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

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)
```

The `c()` is a **function**[4] 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

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

anything) is saving those values so we can work with them[5]. 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`).

[5] 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.

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
##  [5] white    black    white    hispanic
##  [9] 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.

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

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(A = c(1, 2, 1, 4, 3), B = c(1.4,
    2.1, 4.6, 2, 8.2), C = c(0, 0, 1, 1, 1))
df
```

```
##   A   B C
## 1 1 1.4 0
## 2 2 2.1 0
## 3 1 4.6 1
## 4 4 2.0 1
## 5 3 8.2 1
```

We can do quite a bit with the `data.frame` that we called `df`[6]. Once again, we could have called this data frame anything, although I recommend short names. If "A" and "C" are factors we can tell `R` by:

```
df$A <- factor(df$A, labels = c("level1", "level2",
    "level3", "level4"))
df$C <- factor(df$C, labels = c("Male", "Female"))
```

In the above code, the `$` reaches into `df` to grab a variable (or column). The following code does the exact same thing:

```
df[["A"]] <- factor(df[["A"]], labels = c("level1",
    "level2", "level3", "level4"))
df[["C"]] <- factor(df[["C"]], labels = c("Male",
    "Female"))
```

and so is the following:

```
df[, "A"] <- factor(df[, "A"], labels = c("level1",
    "level2", "level3", "level4"))
df[, "C"] <- factor(df[, "C"], labels = c("Male",
    "Female"))
```

`df[["A"]]` grabs the `A` variable just like `df$A`. The last example shows that we can grab both columns and rows. In `df[, "C"]` we have a spot just a head of the comma. It works like this: `df[rows, columns]`. So we could specifically grab certain rows and certain columns.

```
df[1:3, "A"]
df[1:3, 1]
```

Both lines of the above code grabs rows 1 thorugh 3 and column "A".

Finally, we can combine the `c()` function to grab different rows and columns. To grab rows 1 and 5 and columns "B" and "C" you can do the following:

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

We may also want to get more information about the data frame before we do any subsetting. There are a few nice functions to get information that can help us know what we should do next with our data.

```
## Get the variable names
names(df)
```

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

```
## [1] "A" "B" "C"

## Know what type of variable it is
class(df$A)

## [1] "factor"

## Get quick summary statistics for each
## variable
summary(df)

##       A           B             C
##  level1:2   Min.   :1.40   Male  :2
##  level2:1   1st Qu.:2.00   Female:3
##  level3:1   Median :2.10
##  level4:1   Mean   :3.66
##             3rd Qu.:4.60
##             Max.   :8.20

## Get the first 10 columns of your data
head(df, n = 10)

##        A   B      C
## 1 level1 1.4   Male
## 2 level2 2.1   Male
## 3 level1 4.6 Female
## 4 level4 2.0 Female
## 5 level3 8.2 Female
```

We admit that the last one is a bit pointless since our data frame is only a few lines long. However, these functions can give you quick information about your data with hardly any effort on your part.

### *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).

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.

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.

There are some built in ways to import many delimited[7] data types (e.g. comma delimited–also called a CSV, tab delimited, space delimited). Other **packages**[8] have been developed to help with this as well.

[7] The delimiter is what separates the pieces of data.

[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.

### *R Data File*

The first, if it is an `R` data file in the form `.rda` or `.RData` simply use:

```
load("file.rda")
```

Note that you don't assign this to a name such as `df`. Instead, it loads whatever `R` objects were saved to it.

### *Delimited Files*

Most delimited files are saved as `.csv`, `.txt`, or `.dat`. As long as you know the delimiter, this process is easy.

```
df <- read.table("file.csv", sep = ",", header = TRUE)  ## for csv
df <- read.table("file.txt", sep = "\t", header = TRUE)  ## for tab delimited
df <- read.table("file.txt", sep = " ", header = TRUE)  ## for space delimited
```

The argument `sep` tells the function what kind of delimiter the data has and `header` tells `R` if the first row contains the variable names (you can change it to `FALSE` if the first row isn't).

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! Note that unless you are using the `load` function, you need to assign what is being read in to a name. In the examples, all were called `df`. In real life, you won't run a bunch of different `read` functions to the same name because only the last one run would be saved (the others would be written over). However, if you have multiple data files to import you can assign them to different names and later merge them. Merging, also called joining, is something we'll discuss in the next chapter.

### *Other Data Formats*

Data from other statistical software such as SAS, SPSS, or Stata are also easy to get into `R`. We will use two powerful packages:

1. `haven`
2. `foreign`

To install, simply run:

```
install.packages("packagename")
```

This only needs to be run once on a computer. Then, to use it in a single `R` session (i.e. from when you open `R` to when you close it) run:

```
library(packagename)
```

Using these packages, I will show you simple ways to bring your data in from other formats.

```
library(haven)
df <- read_dta("file.dta")  ## for Stata data
df <- read_spss("file.sav")  ## for SPSS data
df <- read_sas("file.sas7bdat")  ## for this type of SAS file

library(foreign)
df <- read.xport("file.xpt")  ## for export SAS files
```

If you have another type of data file to import, online helps found on sites like `www.stackoverflow.com` and `www.r-bloggers.com` often have the solution.

### *Saving Data*

Finally, there are many ways to save data. Most of the `read...` functions have a corresponding `write...` function.

```
write.table(df, file = "file.csv", sep = ",")  ## 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 sep 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 `write.table` simply run: `?write.table`. In the pane with the files, plots, packages, etc. a document will show up to give you more informaton.

### *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)
```

```
## Loading tidyverse: ggplot2
## Loading tidyverse: tibble
## Loading tidyverse: tidyr
## Loading tidyverse: readr
## Loading tidyverse: purrr
## Loading tidyverse: dplyr
```

```
## Conflicts with tidy packages ----------------
```

```
## filter(): dplyr, stats
## lag():    dplyr, stats
```

Note that when we loaded tidyverse it loaded 6 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:

[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`.

```
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)
```

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.

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. Grouping and Summarizing
4. Reshaping
5. Joining (merging)
6. Mutate and Transmute

> 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 -0.7066503 0.435040615
## 2   2 -1.1104510 0.880072281
## 3   3  0.7185098 0.894100884
## 4   4  1.1762907 0.906834961
## 5   5  0.6243977 0.619488233
## 6   6  0.6635914 0.805102223
## 7   7 -1.0705731 0.167266015
## 8   8  0.1805212 0.255621101
## 9   9 -1.4138766 0.629702678
## 10 10  1.5089726 0.004225435
```

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 0.39329329
## 2  1    2 0.48539464
## 3  1    3 0.80631108
## 4  1    4 0.07201179
## 5  2    1 0.95779710
## 6  2    2 0.22860134
## 7  3    1 0.80315401
## 8  3    2 0.95237160
## 9  3    3 0.54581338
```

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. Later chapters will delve into more complex data designs.

### *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. First, we will set our working directory with `setwd`. This tells `R` where to look for files, including your data files. My specific file location is below so you will need to adjust it to wherever you saved the NHANES data.

```
setwd("~/Dropbox/GitHub/blog_rstats/assets/Data/")

library(foreign)
dem_df <- read.xport("NHANES_demographics_11.xpt")
med_df <- read.xport("NHANES_MedHeath_11.xpt")
men_df <- read.xport("NHANES_MentHealth_11.xpt")
act_df <- read.xport("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.

[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`.

### *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 %>% summary
```

So the above code takes the data frame `dem_df` and puts it into the `summary` function. This does the same thing as `summary(df)`. In this simple case, it doesn't really make the code more readable, but in more complex situations it can really help. In the following sections you will see how and where this type of coding is helpful.

### *Select and Filter*

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)
```

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.

### *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

```
## Our Grouping Variable as a factor
dem_df$citizen <- factor(dem_df$dmdcitzn)

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

```
## # A tibble: 4 x 2
##   citizen     N
##    <fctr> <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")

# library(furniture)

dem_df$citizen <- washer(dem_df$citizen, 7)  ## Changes all 7's to NA's
dem_df$citizen <- washer(dem_df$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
```

```
## # A tibble: 3 x 2
##   citizen     N
##     <chr> <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))
```

```
## # A tibble: 3 x 3
##   citizen     N      Age
##     <chr> <int>    <dbl>
## 1       0  1040 37.31635
## 2       1  8685 30.66252
## 3    <NA>    31 40.35484
```

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`.

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.

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

```
# 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
```

```
df_long
```

```
##       ID time        Var
## 1.1    1    1  1.9006760
```

```
## 2.1   2    1  0.3212252
## 3.1   3    1 -0.4589058
## 4.1   4    1  0.1616575
## 5.1   5    1  0.1433238
## 6.1   6    1 -1.8277651
## 7.1   7    1 -1.3120190
## 8.1   8    1  1.3742150
## 9.1   9    1 -0.2087390
## 10.1 10    1 -0.6695582
## 1.2   1    2  0.7610891
## 2.2   2    2  0.1863900
## 3.2   3    2  0.6110682
## 4.2   4    2  0.8411384
## 5.2   5    2  0.5183731
## 6.2   6    2  0.9518318
## 7.2   7    2  0.7925063
## 8.2   8    2  0.6586377
## 9.2   9    2  0.2193556
## 10.2 10    2  0.3941358
```

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 0.7233727 0.8485018 0.8746767 0.6959951
## 5  2 0.2664332 0.8477673        NA        NA
## 7  3 0.5059690 0.8180783 0.1137797        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]

[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.

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
```

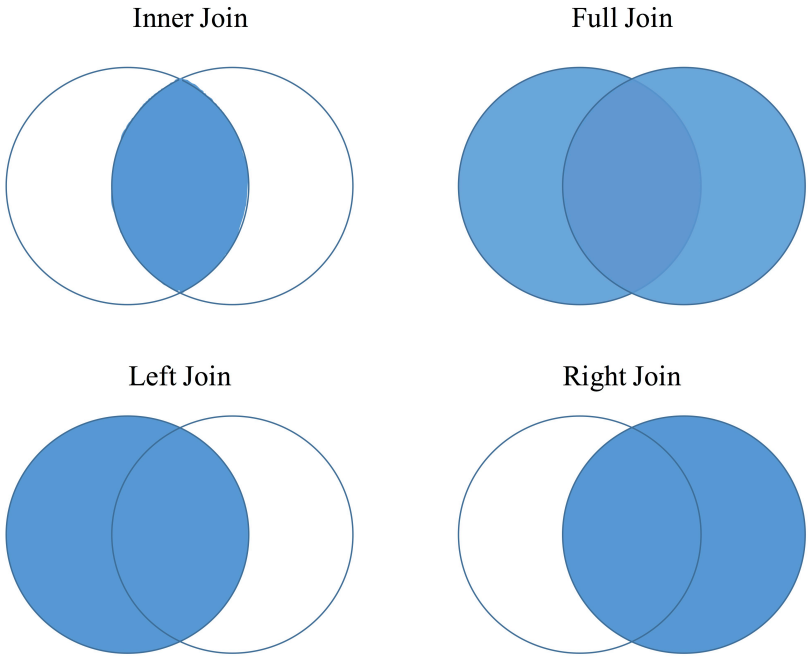



Figure 2: Joining

*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 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.

### *Mutate and Transmute*

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()`.

```
df_fake <- data.frame(ID = c(1:10), Var1 = rnorm(10),
    Var2 = runif(10))

df_fake <- df_fake %>% mutate(Var3 = Var1 + Var2,
    Var4 = Var1 * Var2 + Var3)
df_fake
```

```
##    ID       Var1      Var2        Var3
## 1   1 -0.3010499 0.2154341 -0.08561586
## 2   2  0.3403523 0.5372362  0.87758851
## 3   3  0.4780951 0.1902519  0.66834705
## 4   4  2.3005139 0.6370695  2.93758343
## 5   5  0.1639023 0.2557470  0.41964926
## 6   6  0.4594802 0.8987226  1.35820278
## 7   7  0.5159534 0.5897712  1.10572461
## 8   8  0.9499082 0.1733838  1.12329204
## 9   9  1.8018803 0.4291112  2.23099148
## 10 10 -0.1891407 0.8396435  0.65050273
##          Var4
## 1  -0.1504723
## 2   1.0604381
## 3   0.7593056
## 4   4.4031707
## 5   0.4615668
## 6   1.7711480
## 7   1.4100191
## 8   1.2879908
## 9   3.0041985
## 10  0.4916920
```

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

To highlight its benefits on real data, let's go back to our NHANES data (now called `df`). First, we are going to 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 `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))))
```

### *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.

```
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 1.819597 2.612388
## 2        2 1.815528 3.584081
```

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

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.

# *Chapter 3: Understanding and Describing Your Data*

> "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 first create a ficticious data set and we'll show the basic build of `table1`.

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

```
df <- data.frame(A = c(1, 2, 1, 4, 3, NA), B = c(1.4,
    2.1, 4.6, 2, NA, 3.4), C = c(0, 0, 1, 1, 1,
    1), D = rnorm(6))

# library(furniture)
table1(df, A, B, C, D)

##
## |==============================|
```

```
##                Mean/Count (SD/%)
##  Observations 6
##  A
##                2.2 (1.3)
##  B
##                2.7 (1.3)
##  C
##                0.7 (0.5)
##  D
##                -0.2 (1.0)
## |==============================|
```

This quickly gives you means and standard deviations (or counts and percentages if there were categorical variables). We could have also used the pipe operator here if we wanted via:

```
df %>% table1(A, B, C, D)
```

```
##
## |==============================|
##                Mean/Count (SD/%)
##  Observations 6
##  A
##                2.2 (1.3)
##  B
##                2.7 (1.3)
##  C
##                0.7 (0.5)
##  D
##                -0.2 (1.0)
## |==============================|
```

It turns out, for we want "A" and "C" to be factors.

```
df$A <- factor(df$A, labels = c("cat1", "cat2",
    "cat3", "cat4"))
df$C <- factor(df$C, labels = c("male", "female"))

table1(df, A, B, C, D)
```

```
##
## |==============================|
##                Mean/Count (SD/%)
##  Observations 6
##  A
##     cat1       2 (40%)
```

```
##     cat2      1 (20%)
##     cat3      1 (20%)
##     cat4      1 (20%)
##  B
##               2.7 (1.3)
##  C
##     male      2 (33.3%)
##     female    4 (66.7%)
##  D
##               -0.2 (1.0)
## |==============================|
```

So now we see the counts and percentages for the factor variables. But now we can take a step further and look for relationships. The code below shows the means/standard devaitions or counts/percentages by a grouping variable–in this case, `C`.

```
table1(df, A, B, D, splitby = ~C)
```

```
##
## |==================================|
##                     C
##               male       female
##  Observations 2          4
##  A
##     cat1      1 (50%)    1 (33.3%)
##     cat2      1 (50%)    0 (0%)
##     cat3      0 (0%)     1 (33.3%)
##     cat4      0 (0%)     1 (33.3%)
##  B
##               1.8 (0.5)  3.3 (1.3)
##  D
##               -0.0 (0.7) -0.2 (1.2)
## |==================================|
```

We can also test for differences by group as well (although this is not particularly good with a sample size of 5). It produces a warning since the $\chi^2$ approximation is not accurate with cells this small.

```
table1(df, A, B, D, splitby = ~C, test = TRUE)
```

```
##
## |========================================|
##                         C
##               male       female     P-Value
##  Observations 2          4
```

```
##  A                                    0.405
##     cat1      1 (50%)    1 (33.3%)
##     cat2      1 (50%)    0 (0%)
##     cat3      0 (0%)     1 (33.3%)
##     cat4      0 (0%)     1 (33.3%)
##  B                                    0.162
##               1.8 (0.5)  3.3 (1.3)
##  D                                    0.776
##               -0.0 (0.7) -0.2 (1.2)
## |=========================================|
```

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]

[15] `tysonstanley.github.io`

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

```
summary(df)
```

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

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

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

### *Visualizations*

Understanding your data, in my experience, generally requires visualizations. 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.

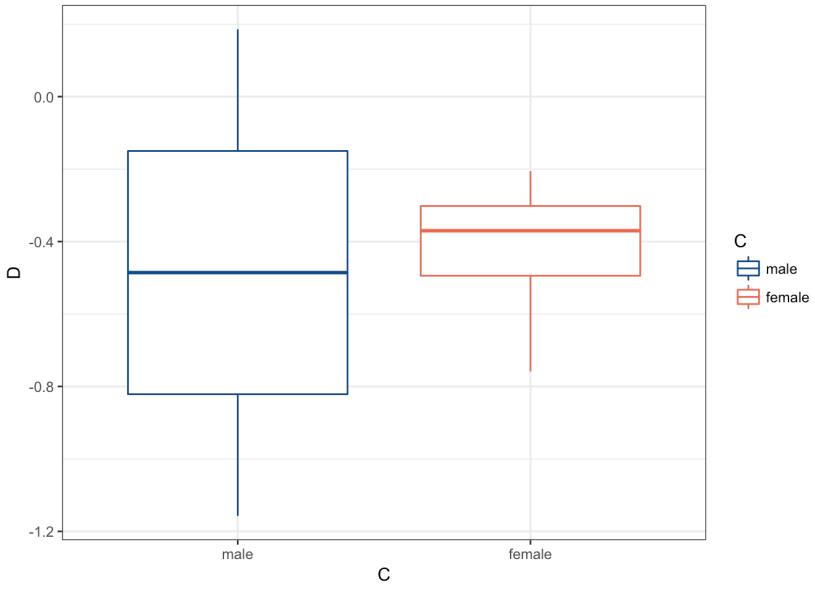
### *Using `ggplot2`*

This package has a straight-forward syntax. It is built by adding layers to the plot.

```
library(tidyverse)
```

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$A)  ## Makes a simple histogram
```




```
qplot(df$D, df$B)  ## Makes a scatterplot
```

```
## Warning: Removed 1 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.

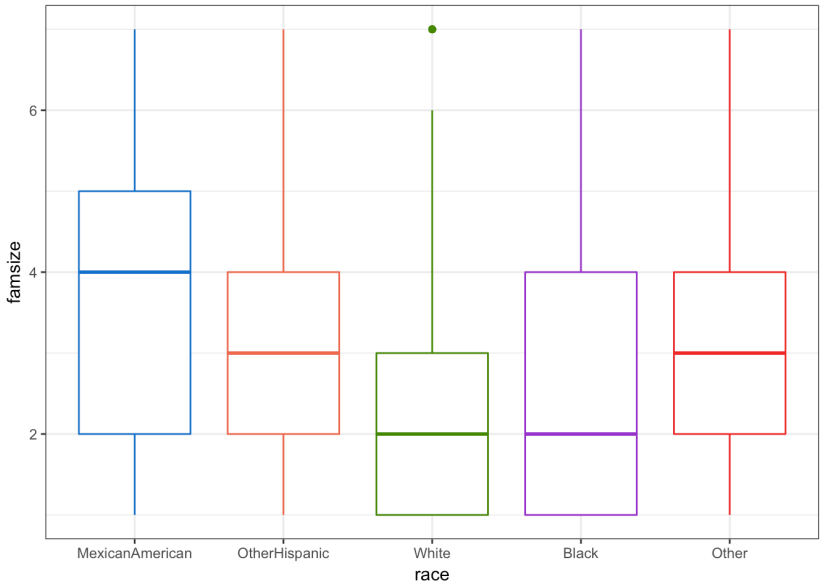
The key pieces of ggplot:

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.

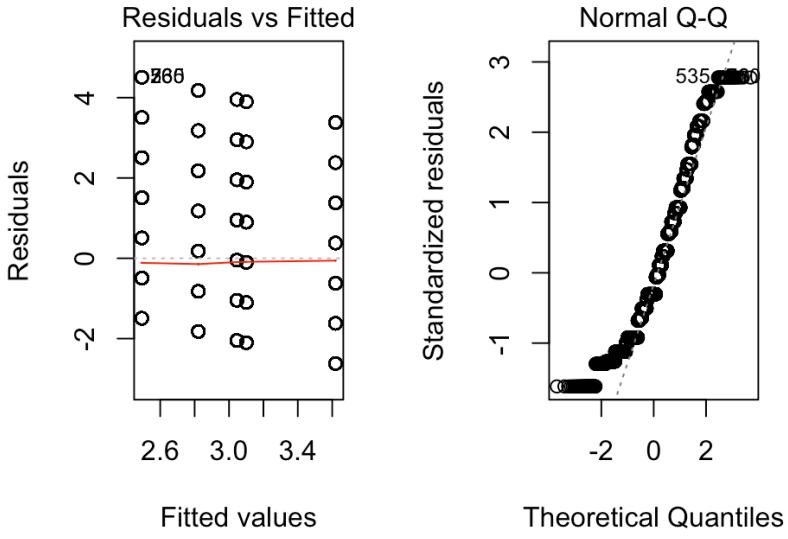
These same pieces will be found throughout `ggplot` plotting. In later chapters we will introduce much more in relation to these plots.

```
ggplot(df, aes(x = C, y = D)) + geom_boxplot(aes(color = C))
```



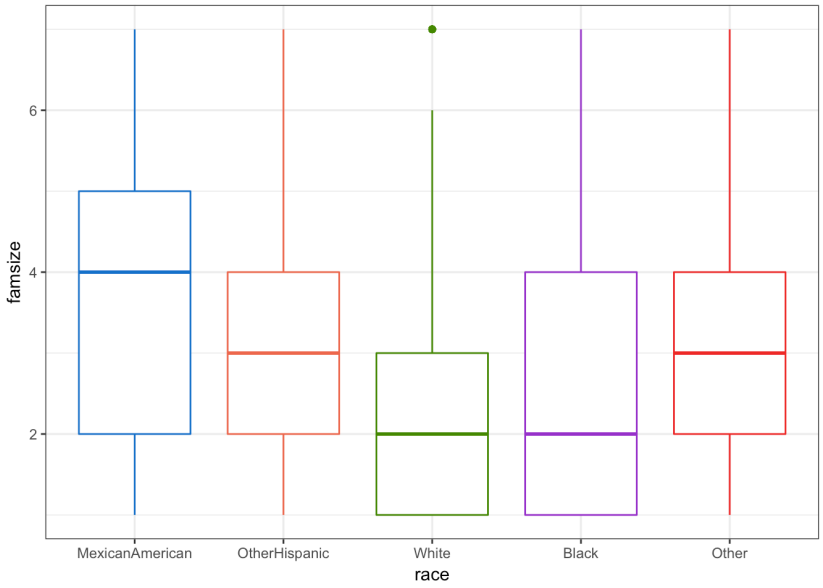

Here's a few more examples:

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



```
ggplot(df, aes(x = B, y = D)) + geom_point(aes(color = C))
```

```
## Warning: Removed 1 rows containing missing values
## (geom_point).
```

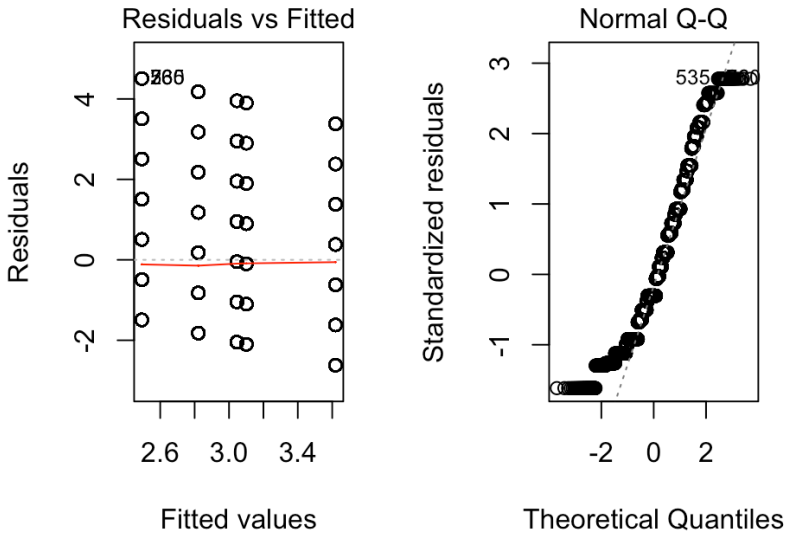



*Note that the warning that says it removed a row is because we had a missing value in "C".*

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.[16]

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

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




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.

# *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 straightforward 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 $\alpha_0$ 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
```

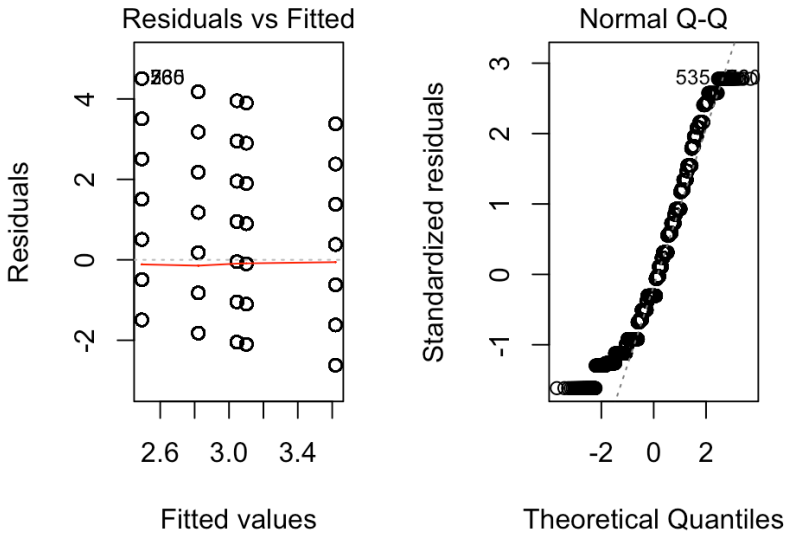
```
## race         4   541.2 135.300  51.367
## Residuals 4627 12187.5   2.634
##              Pr(>F)
## race      < 2.2e-16 ***
## Residuals
## ---
## Signif. codes:
##   0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1  ' ' 1
```

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
## race         4   541.2 135.300  51.367
## Residuals 4627 12187.5   2.634
##              Pr(>F)
## race      < 2.2e-16 ***
## Residuals
## ---
## 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
## -2.6193 -1.4932 -0.4932  0.9539  4.5068
##
## Coefficients:
##                   Estimate Std. Error
## (Intercept)        3.61927    0.07773
## raceOtherHispanic -0.51841    0.10814
## raceWhite         -1.12604    0.08676
## raceBlack         -0.79795    0.09056
```

```
## raceOther         -0.57320    0.09803
##                   t value Pr(>|t|)
## (Intercept)        46.565  < 2e-16 ***
## raceOtherHispanic  -4.794 1.69e-06 ***
## raceWhite         -12.978  < 2e-16 ***
## raceBlack          -8.812  < 2e-16 ***
## raceOther          -5.847 5.35e-09 ***
## ---
## Signif. codes:
##   0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1  ' ' 1
##
## Residual standard error: 1.623 on 4627 degrees of freedom
## Multiple R-squared:  0.04252,    Adjusted R-squared:  0.04169
## F-statistic: 51.37 on 4 and 4627 DF,  p-value: < 2.2e-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 12188
## 2   4627 12188  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
## -2.9689 -1.1774 -0.4101  1.0311  5.2870
##
## Coefficients:
##                    Estimate Std. Error
## (Intercept)         3.96893    0.07790
## raceOtherHispanic -0.41181    0.10368
## raceWhite         -1.02681    0.08364
## raceBlack         -0.55881    0.08791
## raceOther         -0.56242    0.09461
## marriage2         -1.22909    0.08666
## marriage3         -1.20245    0.07784
## marriage4         -0.47750    0.12529
## marriage5         -0.80932    0.05970
## marriage6         -0.49213    0.08866
##                    t value Pr(>|t|)
## (Intercept)         50.952  < 2e-16 ***
## raceOtherHispanic  -3.972 7.23e-05 ***
## raceWhite         -12.277  < 2e-16 ***
## raceBlack          -6.357 2.26e-10 ***
## raceOther          -5.945 2.97e-09 ***
```

```
## marriage2         -14.183  < 2e-16 ***
## marriage3         -15.447  < 2e-16 ***
## marriage4          -3.811  0.00014 ***
## marriage5         -13.556  < 2e-16 ***
## marriage6          -5.551 3.00e-08 ***
## ---
## Signif. codes:
##   0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1  ' ' 1
##
## Residual standard error: 1.552 on 4622 degrees of freedom
## Multiple R-squared:  0.1255, Adjusted R-squared:  0.1238
## F-statistic: 73.73 on 9 and 4622 DF,  p-value: < 2.2e-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 12188
## 2   4622 11131  5    1056.8 87.765 < 2.2e-16
##
## 1
## 2 ***
## ---
## 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)
```

```
##
## Please cite as:

##  Hlavac, Marek (2015). stargazer: Well-Formatted Regression and Summary Statistics Tables.

##  R package version 5.2. http://CRAN.R-project.org/package=stargazer

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

##
## ===============================================================
##                                 Dependent variable:
##                     ------------------------------------------
##                                        famsize
##                              (1)                    (2)
## ---------------------------------------------------------------
## raceOtherHispanic         -0.518***              -0.412***
##                            (0.108)                (0.104)
##
## raceWhite                 -1.126***              -1.027***
##                            (0.087)                (0.084)
##
## raceBlack                 -0.798***              -0.559***
##                            (0.091)                (0.088)
##
## raceOther                 -0.573***              -0.562***
##                            (0.098)                (0.095)
##
## marriage2                                        -1.229***
##                                                   (0.087)
##
## marriage3                                        -1.202***
##                                                   (0.078)
##
## marriage4                                        -0.477***
##                                                   (0.125)
##
## marriage5                                        -0.809***
##                                                   (0.060)
##
## marriage6                                        -0.492***
##                                                   (0.089)
##
## Constant                   3.619***               3.969***
##                            (0.078)                (0.078)
##
```

```
## ------------------------------------------------------------------------
## Observations                  4,632                     4,632
## R2                            0.043                     0.126
## Adjusted R2                   0.042                     0.124
## Residual Std. Error    1.623 (df = 4627)        1.552 (df = 4622)
## F Statistic         51.367*** (df = 4; 4627) 73.729*** (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
## minority              1   334.9  334.92
## marriage              5  1153.3  230.65
## minority:marriage     5    24.9    4.98
## Residuals          4620 11215.6    2.43
##                      F value  Pr(>F)
## minority           137.9637 < 2e-16 ***
## marriage            95.0112 < 2e-16 ***
## minority:marriage    2.0527 0.06831 .
## Residuals
## ---
## 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
## -2.4794 -1.1957 -0.4794  0.9805  5.4353
##
## Coefficients:
##                             Estimate
## (Intercept)                  3.01952
## minorityMinority             0.45989
## marriage2                   -1.45482
## marriage3                   -1.24374
## marriage4                   -0.51952
## marriage5                   -1.01614
## marriage6                   -0.69891
## minorityMinority:marriage2  0.35455
## minorityMinority:marriage3  0.02632
## minorityMinority:marriage4  0.09808
## minorityMinority:marriage5  0.25181
## minorityMinority:marriage6  0.41515
##                             Std. Error
## (Intercept)                    0.05131
## minorityMinority               0.06723
## marriage2                      0.13005
## marriage3                      0.11627
## marriage4                      0.28906
## marriage5                      0.10409
## marriage6                      0.14548
## minorityMinority:marriage2     0.17408
## minorityMinority:marriage3     0.15609
## minorityMinority:marriage4     0.32098
## minorityMinority:marriage5     0.12680
## minorityMinority:marriage6     0.18332
##                             t value Pr(>|t|)
## (Intercept)                  58.846  < 2e-16
## minorityMinority              6.841 8.89e-12
## marriage2                   -11.187  < 2e-16
## marriage3                   -10.697  < 2e-16
## marriage4                    -1.797   0.0724
## marriage5                    -9.762  < 2e-16
## marriage6                    -4.804 1.60e-06
## minorityMinority:marriage2   2.037   0.0417
## minorityMinority:marriage3   0.169   0.8661
## minorityMinority:marriage4   0.306   0.7599
```

```
## minorityMinority:marriage5   1.986   0.0471
## minorityMinority:marriage6   2.265   0.0236
##
## (Intercept)                 ***
## minorityMinority            ***
## marriage2                   ***
## marriage3                   ***
## marriage4                   .
## marriage5                   ***
## marriage6                   ***
## minorityMinority:marriage2 *
## minorityMinority:marriage3
## minorityMinority:marriage4
## minorityMinority:marriage5 *
## minorityMinority:marriage6 *
## ---
## Signif. codes:
##   0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1  ' ' 1
##
## Residual standard error: 1.558 on 4620 degrees of freedom
## Multiple R-squared:  0.1189, Adjusted R-squared:  0.1168
## F-statistic: 56.66 on 11 and 4620 DF,  p-value: < 2.2e-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.

# *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[17] uses dummy coding and provides coefficients that tell us the relationship between the outcome and the independent variables.

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

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 $\geq 16$ meaning depressed. Because there are missing values denoted as "NA" in this variable, we use a "nested ifelse" to say:

1. IF depression $\geq 10$ 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
## -0.7831  -0.4479  -0.4078  -0.3645   2.5471
##
## Coefficients:
##                      Estimate Std. Error
## (Intercept)       -2.6203555  0.2380770
## asthma             0.5688452  0.1276326
```

```
## sed                0.0005638  0.0002610
## raceOtherHispanic  0.7162568  0.2328673
## raceWhite          0.1287059  0.2116414
## raceBlack          0.0189205  0.2205461
## raceOther         -0.4901414  0.2570123
## famsize           -0.0318309  0.0373218
##                   z value Pr(>|z|)
## (Intercept)       -11.006  < 2e-16 ***
## asthma              4.457 8.32e-06 ***
## sed                 2.160   0.0307 *
## raceOtherHispanic   3.076   0.0021 **
## raceWhite           0.608   0.5431
## raceBlack           0.086   0.9316
## raceOther          -1.907   0.0565 .
## famsize            -0.853   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: 2664.2
##
## 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
## -27.362   -8.430   -1.477    5.823   34.507
##
## Coefficients:
##                     Estimate Std. Error
## (Intercept)        5.6499871  0.0035550
## asthma             0.0614965  0.0021434
## raceOtherHispanic  0.1393438  0.0040940
## raceWhite          0.3484622  0.0033438
## raceBlack          0.3400346  0.0034430
## raceOther          0.3557953  0.0036273
## famsize           -0.0188673  0.0005488
##                   z value Pr(>|z|)
## (Intercept)       1589.31   <2e-16 ***
## asthma              28.69   <2e-16 ***
## raceOtherHispanic   34.04   <2e-16 ***
## raceWhite          104.21   <2e-16 ***
## raceBlack           98.76   <2e-16 ***
## raceOther           98.09   <2e-16 ***
## famsize            -34.38   <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 + 0.01
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
## -4.3589  -0.4613  -0.0845   0.2926   1.6868
##
## Coefficients:
##                      Estimate Std. Error
## (Intercept)         3.567e-03  1.132e-04
## asthma             -1.604e-04  5.865e-05
## raceOtherHispanic -4.874e-04  1.309e-04
## raceWhite          -1.090e-03  1.078e-04
## raceBlack          -1.068e-03  1.102e-04
## raceOther          -1.110e-03  1.145e-04
## famsize             5.107e-05  1.552e-05
##                    t value Pr(>|t|)
## (Intercept)         31.515  < 2e-16 ***
## asthma              -2.735  0.00626 **
## raceOtherHispanic  -3.723  0.00020 ***
## raceWhite          -10.115  < 2e-16 ***
## raceBlack           -9.697  < 2e-16 ***
## raceOther           -9.695  < 2e-16 ***
## famsize              3.289  0.00101 **
## ---
## Signif. codes:
##   0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1  ' ' 1
##
## (Dispersion parameter for Gamma family taken to be 0.2932604)
##
```

```
##     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
## -3.3148 -1.5063 -0.2188  1.2829  9.6110
##
## Count model coefficients (truncated poisson with log link):
##                     Estimate Std. Error
## (Intercept)        5.6556743  0.0037237
## asthma             0.0670780  0.0022503
## raceOtherHispanic  0.1353773  0.0042856
```

```
## raceWhite          0.3392431  0.0034972
## raceBlack          0.3257467  0.0036009
## raceOther          0.3410073  0.0037937
## famsize           -0.0177763  0.0005775
##                   z value Pr(>|z|)
## (Intercept)       1518.85   <2e-16 ***
## asthma              29.81   <2e-16 ***
## raceOtherHispanic   31.59   <2e-16 ***
## raceWhite           97.00   <2e-16 ***
## raceBlack           90.46   <2e-16 ***
## raceOther           89.89   <2e-16 ***
## famsize            -30.78   <2e-16 ***
## Zero hurdle model coefficients (binomial with logit link):
##                   Estimate Std. Error
## (Intercept)        2.38261    0.21214
## asthma            -0.07456    0.14367
## raceOtherHispanic -0.09316    0.23669
## raceWhite         -0.04979    0.19581
## raceBlack          0.07070    0.20555
## raceOther          0.04474    0.22008
## famsize           -0.02533    0.03605
##                   z value Pr(>|z|)
## (Intercept)        11.232   <2e-16 ***
## asthma             -0.519    0.604
## raceOtherHispanic  -0.394    0.694
## raceWhite          -0.254    0.799
## raceBlack           0.344    0.731
## raceOther           0.203    0.839
## famsize            -0.703    0.482
## ---
## 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$beta_var <- sample(seq(0.05, 0.99, by = 0.01),
    size = length(df$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
## -2.0331 -0.6976 -0.0556  0.6349  2.8947
##
## Coefficients (mean model with logit link):
##                    Estimate Std. Error
## (Intercept)         0.02040    0.06351
## asthma             -0.05775    0.04389
## raceOtherHispanic   0.04868    0.07231
## raceWhite           0.02923    0.05899
## raceBlack           0.06942    0.06124
## raceOther           0.06649    0.06564
## famsize             0.01600    0.01087
```

```
##                   z value Pr(>|z|)
## (Intercept)         0.321    0.748
## asthma             -1.316    0.188
## raceOtherHispanic   0.673    0.501
## raceWhite           0.496    0.620
## raceBlack           1.134    0.257
## raceOther           1.013    0.311
## famsize             1.472    0.141
##
## Phi coefficients (precision model with identity link):
##       Estimate Std. Error z value Pr(>|z|)
## (phi)  2.43514    0.04426   55.01   <2e-16
##
## (phi) ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Type of estimator: ML (maximum likelihood)
## Log-likelihood: 73.39 on 8 Df
## Pseudo R-squared: 0.0014
## Number of iterations: 16 (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.

### *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[18]. 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.

[18] 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.

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 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
##       2.500022059       1.356081567
##           famsize               sed
```

```
##      -0.042132178       0.001362226
## raceOtherHispanic         raceWhite
##       1.184995689       0.113949209
##         raceBlack         raceOther
##       0.100536695      -0.555478773
```

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

```
##                       Estimate   Naive S.E.
## (Intercept)        2.495509790 0.2867816215
## asthmaAsthma       1.353039007 0.1867101195
## famsize           -0.039489294 0.0461945052
## sed                0.001358042 0.0003362291
## raceOtherHispanic  1.192481318 0.3075562837
## raceWhite          0.116185743 0.2531554533
## raceBlack          0.096800821 0.2625826864
## raceOther         -0.555053605 0.2809301544
##                      Naive z  Robust S.E.
## (Intercept)        8.7017773 0.2690426648
## asthmaAsthma       7.2467363 0.2137975620
## famsize           -0.8548483 0.0457474654
## sed                4.0390382 0.0003551901
## raceOtherHispanic  3.8772783 0.3309608614
## raceWhite          0.4589502 0.2279687738
## raceBlack          0.3686489 0.2360498473
## raceOther         -1.9757708 0.2406566044
##                     Robust z
## (Intercept)        9.2755169
## asthmaAsthma       6.3285989
## famsize           -0.8632018
## sed                3.8234244
## raceOtherHispanic  3.6030886
## raceWhite          0.5096564
## raceBlack          0.4100864
## raceOther         -2.3064133
```

```
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
## (Intercept)         2.5579361  0.2700717
## asthmaAsthma        1.3492892  0.2156202
## famsize            -0.0446716  0.0457087
## sed                 0.0013015  0.0003548
## raceOtherHispanic   1.1750373  0.3318983
## raceWhite           0.0806377  0.2295661
## raceBlack           0.0642028  0.2363255
## raceOther          -0.5902049  0.2413379
##                       Wald Pr(>|W|)
## (Intercept)       89.706  < 2e-16 ***
## asthmaAsthma      39.159 3.91e-10 ***
## famsize            0.955 0.328415
## sed               13.454 0.000244 ***
## raceOtherHispanic 12.534 0.000400 ***
## raceWhite          0.123 0.725392
## raceBlack          0.074 0.785875
## raceOther          5.981 0.014463 *
## ---
## Signif. codes:
##   0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1  ' ' 1
##
## Estimated Scale Parameters:
##             Estimate Std.err
## (Intercept)    19.49  0.7843
##
## Correlation: Structure = unstructured  Link = identity
##
## Estimated Correlation Parameters:
##           Estimate Std.err
## alpha.1:2  0.12480 0.01654
## alpha.1:3  0.42070 0.10339
## alpha.1:4  2.89640 1.06678
## alpha.1:5 -1.85447 0.20276
## alpha.2:3  0.12238 0.06330
## alpha.2:4 -0.08935 0.20229
## alpha.2:5  0.20541 0.03720
## alpha.3:4 -0.49597 0.11227
## alpha.3:5  0.25045 0.03879
## alpha.4:5 -0.66939 0.08761
## 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
## (Intercept)         2.491678   0.302768
## asthmaAsthma        1.335445   0.186618
## famsize            -0.042857   0.046341
## sed                 0.001425   0.000337
## raceOtherHispanic   1.289890   0.320595
## raceWhite           0.008348   0.259449
## raceBlack           0.171658   0.273382
## raceOther          -0.552746   0.285512
##                    t value
## (Intercept)           8.23
## asthmaAsthma          7.16
## famsize              -0.92
## sed                   4.23
## raceOtherHispanic     4.02
## raceWhite             0.03
## raceBlack             0.63
## raceOther            -1.94
##
## Correlation of Fixed Effects:
##              (Intr) asthmA famsiz sed
## 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
## raceBlack   -0.643 -0.057  0.094 -0.131
## raceOther   -0.580  0.000  0.048 -0.135
##              rcOthH racWht rcBlck
## asthmaAsthm
## famsize
## sed
## rcOthrHspnc
## raceWhite    0.639
## raceBlack    0.624  0.775
## raceOther    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
## fit_me1  9 25829 25886 -12905    25811
## fit_me  10 25780 25844 -12880    25760  50.9
##         Chi Df Pr(>Chisq)
## fit_me1
## fit_me       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
## fit_me2  9 25855 25912 -12918    25837
## fit_me3 11 25821 25892 -12900    25799  37.3
##         Chi Df Pr(>Chisq)
## fit_me2
## fit_me3      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| =
## 0.00854237 (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
## (Intercept)        -2.6316     0.2435
## asthmaAsthma        0.5619     0.1281
## famsize            -0.0336     0.0374
## I(sed/1000)         0.5835     0.2618
## raceOtherHispanic   0.7564     0.2421
## raceWhite           0.0955     0.2159
## raceBlack           0.0531     0.2277
## raceOther          -0.4950     0.2591
##                   z value Pr(>|z|)
## (Intercept)        -10.81  < 2e-16 ***
## asthmaAsthma         4.39  1.2e-05 ***
## famsize             -0.90   0.3696
## I(sed/1000)          2.23   0.0258 *
## raceOtherHispanic    3.12   0.0018 **
## raceWhite            0.44   0.6581
## raceBlack            0.23   0.8155
## raceOther           -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
## 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
## raceBlack   -0.684 -0.064  0.088 -0.124
## raceOther   -0.571 -0.003  0.046 -0.122
##             rcOthH racWht rcBlck
## asthmaAsthm
## famsize
## I(sed/1000)
## rcOthrHspnc
## raceWhite    0.709
## raceBlack    0.715  0.781
## raceOther    0.606  0.688  0.653
```

## *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.

## *Mediation Modeling*

Mediation modeling can be done via several packages. For now, we recommend using `lavaan` (stands for "latent variable analysis")[19]. Although it is technically still a "beta" version, it performs very well especially for more simple models. It makes mediation modeling straightforward.

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

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.

```r
library(lavaan)
df$sed_hr = df$sed/60  ## in hours instead of minutes

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

```
## lavaan (0.5-23.1097) converged normally after  30 iterations
##
##                                                   Used       Total
##   Number of observations                          4614        4632
##
##   Estimator                                         ML
##   Minimum Function Test Statistic                0.000
##   Degrees of freedom                                 0
##
## Parameter Estimates:
##
##   Information                                 Expected
##   Standard Errors                             Standard
##
## Regressions:
##                    Estimate  Std.Err
##   dep ~
##     asthma            1.478    0.183
##   sed_hr ~
##     dep               0.044    0.011
##     asthma            0.412    0.139
##   z-value  P(>|z|)
##
##     8.084    0.000
##
##     3.929    0.000
##     2.965    0.003
##
## Variances:
##                    Estimate  Std.Err
##    .dep              19.597    0.408
##    .sed_hr           11.171    0.233
```

```
##   z-value  P(>|z|)
##    48.031    0.000
##    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 (0.5-23.1097) converged normally after  30 iterations
##
##                                                   Used       Total
##   Number of observations                          4614        4632
##
##   Estimator                                         ML
##   Minimum Function Test Statistic                0.000
##   Degrees of freedom                                 0
##
## Parameter Estimates:
##
##   Information                                 Expected
##   Standard Errors                             Standard
##
## Regressions:
##                    Estimate  Std.Err
##   dep ~
##     asthma     (a)    1.478    0.183
##   sed_hr ~
##     dep        (b)    0.044    0.011
##     asthma     (c)    0.412    0.139
##   z-value  P(>|z|)
##
```

```
##     8.084    0.000
##
##     3.929    0.000
##     2.965    0.003
##
## Variances:
##                    Estimate  Std.Err
##    .dep              19.597    0.408
##    .sed_hr           11.171    0.233
##   z-value  P(>|z|)
##    48.031    0.000
##    48.031    0.000
##
## Defined Parameters:
##                    Estimate  Std.Err
##     indirect          0.065    0.018
##     total             0.477    0.138
##   z-value  P(>|z|)
##     3.534    0.000
##     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 (0.5-23.1097) converged normally after  33 iterations
##
##                                                   Used       Total
##   Number of observations                          4614        4632
##
##   Estimator                                         ML
```

```
##   Minimum Function Test Statistic                0.000
##   Degrees of freedom                                 0
##   Minimum Function Value               0.0000000000000
##
## Parameter Estimates:
##
##   Information                                 Expected
##   Standard Errors                             Standard
##
## Regressions:
##                    Estimate  Std.Err
##   dep ~
##     asthma            1.462    0.183
##     ridageyr         -0.005    0.004
##   sed_hr ~
##     dep               0.044    0.011
##     asthma            0.412    0.139
##     ridageyr         -0.000    0.003
##   z-value  P(>|z|)
##
##     7.980    0.000
##    -1.330    0.183
##
##     3.927    0.000
##     2.956    0.003
##    -0.063    0.950
##
## Variances:
##                    Estimate  Std.Err
##    .dep              19.590    0.408
##    .sed_hr           11.171    0.233
##   z-value  P(>|z|)
##    48.031    0.000
##    48.031    0.000
```

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

### *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 (0.5-23.1097) converged normally after  47 iterations
##
##                                                   Used       Total
##   Number of observations                          4614        4632
##
##   Estimator                                         ML
##   Minimum Function Test Statistic             1065.848
##   Degrees of freedom                                43
##   P-value (Chi-square)                           0.000
##
## Parameter Estimates:
##
##   Information                                 Expected
##   Standard Errors                             Standard
##
## Latent Variables:
##                    Estimate  Std.Err
##   dep1 =~
##     dpq010            1.000
##     dpq020            1.096    0.024
##     dpq030            1.133    0.031
##     dpq040            1.149    0.030
##     dpq050            0.933    0.025
##     dpq060            0.929    0.022
##     dpq070            0.871    0.022
##     dpq080            0.686    0.019
##     dpq090            0.308    0.011
##   z-value  P(>|z|)
##
##
##    45.136    0.000
```

```
##    36.908    0.000
##    38.066    0.000
##    36.773    0.000
##    42.107    0.000
##    39.760    0.000
##    36.325    0.000
##    28.544    0.000
##
## Regressions:
##                   Estimate  Std.Err
##   dep1 ~
##     asthma     (a)    0.173    0.023
##   sed_hr ~
##     dep1       (b)    0.342    0.105
##     asthma     (c)    0.418    0.139
##   z-value  P(>|z|)
##
##     7.656    0.000
##
##     3.275    0.001
##     2.998    0.003
##
## Variances:
##                   Estimate  Std.Err
##    .dpq010            0.306    0.007
##    .dpq020            0.212    0.006
##    .dpq030            0.559    0.013
##    .dpq040            0.514    0.012
##    .dpq050            0.384    0.009
##    .dpq060            0.221    0.005
##    .dpq070            0.249    0.006
##    .dpq080            0.217    0.005
##    .dpq090            0.090    0.002
##    .sed_hr           11.179    0.233
##    .dep1              0.256    0.010
##   z-value  P(>|z|)
##    42.008    0.000
##    37.549    0.000
##    43.807    0.000
##    43.302    0.000
##    43.862    0.000
##    40.808    0.000
##    42.420    0.000
##    44.038    0.000
```

```
##    46.106    0.000
##    48.012    0.000
##    24.657    0.000
##
## Defined Parameters:
##                   Estimate  Std.Err
##     indirect         0.059    0.020
##     total            0.477    0.138
##   z-value  P(>|z|)
##     3.019    0.003
##     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 (0.5-23.1097) converged normally after  29 iterations
##
##   Number of observations                          4632
##
##   Estimator                                         ML
##   Minimum Function Test Statistic              985.831
##   Degrees of freedom                                27
##   P-value (Chi-square)                           0.000
##
## Parameter Estimates:
##
##   Information                                 Expected
##   Standard Errors                             Standard
##
## Latent Variables:
##                   Estimate  Std.Err
##   dep1 =~
##     dpq010           1.000
##     dpq020           1.097    0.024
##     dpq030           1.128    0.031
```

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

#### *Ridge, Lasso and Elastic Net*

In order to do either ridge, lasso, or elastic net regression, we can use the fantastic `glmnet` package. Using the `cv.glmnet()` function we can run the ridge ($alpha = 0$), lasso ($alpha = 1$ which is default), and elastic net ($0 \leq alpha \leq 1$). It turns out that elastic net is the combination of the ridge and lasso methods and the closer `alpha` is to 1 the more it acts like lasso and the closer it is to 0 the more it acts like ridge.

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
X <- df2 %>% dplyr::select(-sed_hr) %>% data.matrix
Y <- df2$sed_hr
```

Then we use the `cv.glmnet()` function to fit the different models. The "cv" refers to cross-validation[20], which we don't discuss here, but it an important topic to become familiar with. Below we fit a ridge, a lasso, and an elastic net model. The elastic net model uses more of the lasso penalty because the `alpha` is closer to 1 than 0.

[20] 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.

```
library(glmnet)

fit_ridge <- cv.glmnet(X, Y, alpha = 0)
fit_lasso <- cv.glmnet(X, Y, alpha = 1)
fit_enet <- cv.glmnet(X, Y, alpha = 0.8)
```

The plots below show where appropriate `lambda` values are based on the mean squared error of the cross-validated prediction. The vertical dashed lines show a reasonable range of lambda values that can be used.

```
plot(fit_ridge)

plot(fit_lasso)

plot(fit_enet)
```

We can get the coefficients at a reasonable `lambda`. Specifically, we use the "1-SE rule" (near the right hand side vertical dashed lines in the above plots) by `s = "lambda.1se"`. You can directly tell it what `lambda` value you'd like but this is a simple rule of thumb.

```
coef(fit_ridge, s = "lambda.1se")
```

```
## 8 x 1 sparse Matrix of class "dgCMatrix"
##                       1
## (Intercept)  5.5019415
## riagendr     0.0527863
## ridageyr    -0.0007295
## ridreth3     0.0603953
## race         0.1046580
## famsize     -0.0494439
## dep          0.0130551
## asthma       0.1239795
```

```
coef(fit_lasso, s = "lambda.1se")
```

```
## 8 x 1 sparse Matrix of class "dgCMatrix"
##                    1
## (Intercept) 5.7065
## riagendr    .
## ridageyr    .
## ridreth3    .
## race        0.1355
## famsize     .
## dep         .
## asthma      .
```

```
coef(fit_enet, s = "lambda.1se")
```

```
## 8 x 1 sparse Matrix of class "dgCMatrix"
##                   1
## (Intercept) 5.5505
## riagendr    .
## ridageyr    .
## ridreth3    .
## race        0.1827
## famsize     .
## dep         .
## asthma      .
```

Although we briefly introduce these regression methods, they are indeed very important. We highly recommend learning more about them.

### *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 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(randomForest)
```

```
## randomForest 4.6-12
```

```
## Type rfNews() to see new features/changes/bug fixes.
```

```
##
## Attaching package: 'randomForest'
```

```
## The following object is masked from 'package:dplyr':
##
##     combine
```

```
## The following object is masked from 'package:ggplot2':
##
##     margin
```

```
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: 10.83
##                     % Var explained: 3.62
```

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.

### *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?
2. How do can we simplify tasks that we need done many times?

We will introduce both ideas to you in this chapter. To discuss the first, we will introduce two functions to help you reshape your data: `gather()` and `spread()`. 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, behavior, and social science, these two forms are sufficient to know. But the question is, how to change the form of your data?

In the `tidyverse` functions known as `gather` and `spread` can help with this in a simple way. To show you, we will use the fake data we started with in Chapter 2.

```
##    ID Var_Time1 Var_Time2
## 1   1   1.68145   0.89783
## 2   2  -0.42953   0.27267
## 3   3  -0.80791   0.64759
```

```
## 4   4   0.54402   0.30717
## 5   5   0.07319   0.81328
## 6   6  -1.17542   0.51395
## 7   7   1.30849   0.14696
## 8   8   1.32158   0.18437
## 9   9   0.24504   0.04632
## 10 10  -1.37345   0.67313
```

Notice that this data frame is in wide format (each ID is one row and there are multiple times or measurements per person). To change this to wide format, we'll use `gather()`. 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 (in this case, `Var_Time1` and `Var_Time2`).

```
library(tidyverse)
long_form <- gather(d1, "measures", "values",
    2:3)
long_form
```

```
##    ID  measures   values
## 1   1 Var_Time1  1.68145
## 2   2 Var_Time1 -0.42953
## 3   3 Var_Time1 -0.80791
## 4   4 Var_Time1  0.54402
## 5   5 Var_Time1  0.07319
## 6   6 Var_Time1 -1.17542
## 7   7 Var_Time1  1.30849
## 8   8 Var_Time1  1.32158
## 9   9 Var_Time1  0.24504
## 10 10 Var_Time1 -1.37345
## 11  1 Var_Time2  0.89783
## 12  2 Var_Time2  0.27267
## 13  3 Var_Time2  0.64759
## 14  4 Var_Time2  0.30717
## 15  5 Var_Time2  0.81328
## 16  6 Var_Time2  0.51395
## 17  7 Var_Time2  0.14696
## 18  8 Var_Time2  0.18437
## 19  9 Var_Time2  0.04632
## 20 10 Var_Time2  0.67313
```

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 `spread()` function. All we do is provide the long formed data frame, the measured variable (`measures`) and the variable with the values (`values`).

```
wide_form <- spread(long_form, measures, values)
wide_form
```

```
##    ID Var_Time1 Var_Time2
## 1   1   1.68145   0.89783
## 2   2  -0.42953   0.27267
## 3   3  -0.80791   0.64759
## 4   4   0.54402   0.30717
## 5   5   0.07319   0.81328
## 6   6  -1.17542   0.51395
## 7   7   1.30849   0.14696
## 8   8   1.32158   0.18437
## 9   9   0.24504   0.04632
## 10 10  -1.37345   0.67313
```

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 `reshape()` function*

It is also possible to do multiple measures at once when moving from wide to long or long to wide. The following figure shows the basics of the `reshape()` function that was made for an introductory R class (which can also be found here).

Note a few important features:

1. `reshape()` is used for moving from wide to long and long to wide. Here we just tell it the direction.
2. To indicate multiple columns, use a list of vectors (e.g., `list(c("x1", "x2"), c("z1", "z2"))`).
3. `reshape()` can be used before or after subsetting, allowing for only the necessary variables to be included in the reshaping process.

## *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.[21] 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`.

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

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] 10.000  1.800  1.033
```

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

```
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 1.033
```

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] 5.657 9.115 6.393 9.834 7.919 5.704
##  [7] 6.792 7.333 4.605 6.688
```

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] 5.657 9.115 6.393 9.834 7.919 5.704
##  [7] 6.792 7.333 4.605 6.688
```

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] 5.657 9.115 6.393 9.834 7.919 5.704
##  [7] 6.792 7.333 4.605 6.688
```

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 1.033
##
## $v2
##    N  Mean     SD
## 1 10 5.204 0.8932
```

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 1.800 1.0328
## v2 10 5.204 0.8932
```

`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
## 1.250 2.167
```

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
##    forloop(v1, v2)  9395 10973.5 15020.7
##  apply(df, 1, sum) 74609 77858.5 92794.9
##            v1 + v2   169   233.5   574.9
##   median      uq    max neval cld
##  11921.5 13244.0  42785   100  b
##  83572.0 98409.0 302127   100   c
##    437.5   536.5  15025   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,] 10.000
## [2,]  5.204
```

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] 1.936 5.809 3.873 7.746 3.873 1.936
##  [7] 3.873 1.936 1.936 1.936
```

```
##
## $v2
##  [1] 10.429 13.692  9.836 13.065 13.254
##  [6] 10.532 10.731 14.180  8.073 12.736
```

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.

### *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: Advanced Plotting*

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

Once again, we will turn to our friend `ggplot2` to plot; but now, we are going to take it to another level. We will use many of the options that this powerful package provides and discuss briefly some important aspects of a good plot.

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:

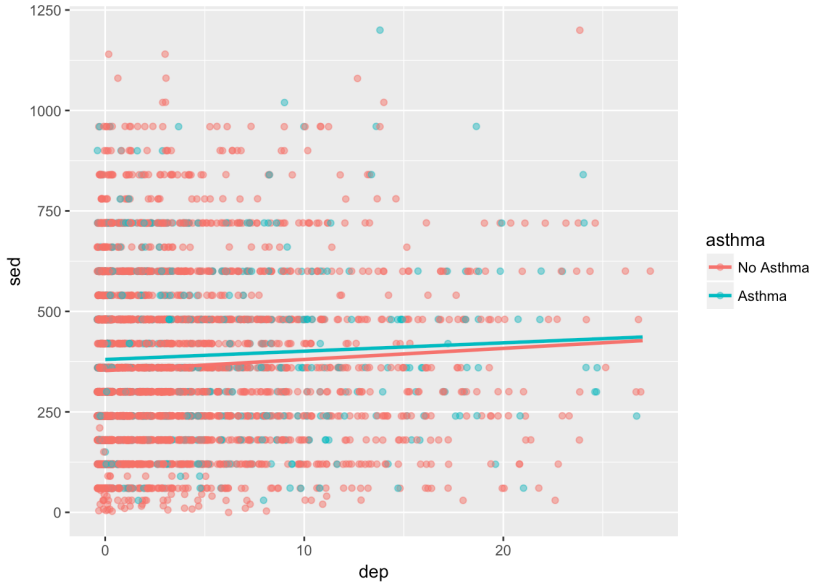
```
ggplot(df, aes(x = dep, y = sed, group = asthma,
    color = asthma))
```

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())
```

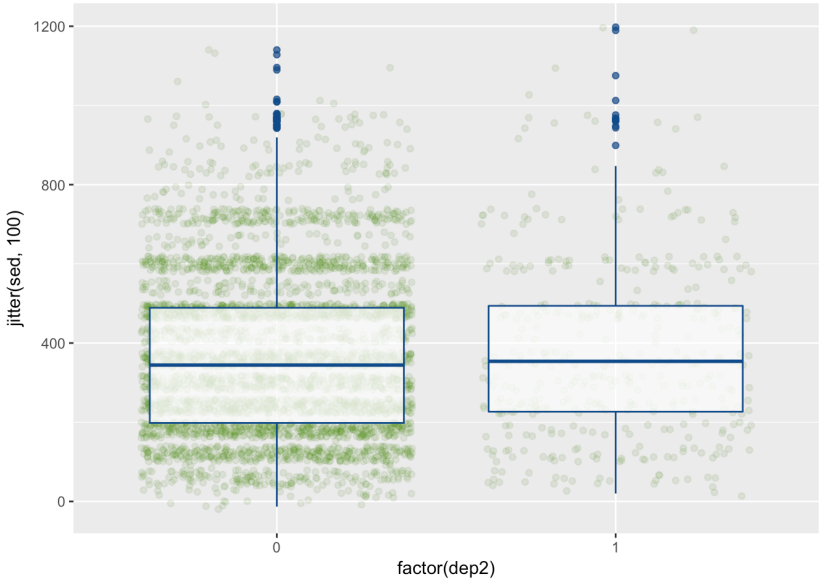
As you hopefully recognize a bit, we are summarizing the time spent being sedentary by both asthma and the dichotomous depression variables. If it doesn't make sense at first, read it line by line to see what I did. This will be useful for several types of plots.

## *Types of Plots*

### *Scatterplots*

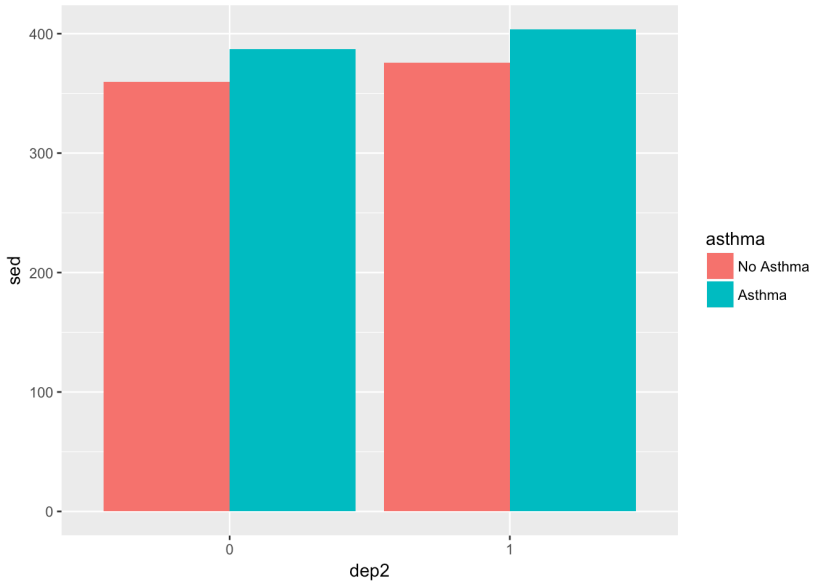
We'll start with a scatterplot–one of the most simple yet informative plots.

```
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 = 0.5) +
    geom_smooth(aes(color = asthma), method = "lm",
        se = FALSE)
```

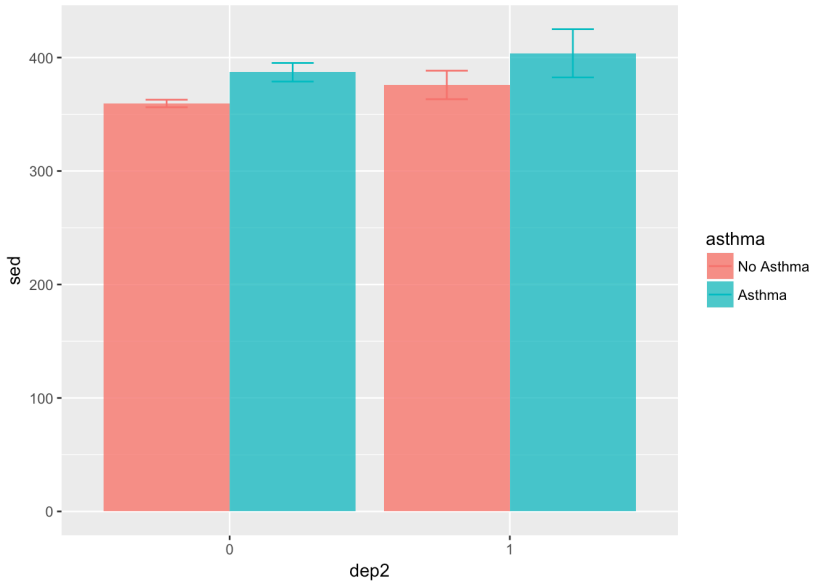



It's getting better but we could still use some more features. We'll come back to this in the next sections.

### *Boxplots*

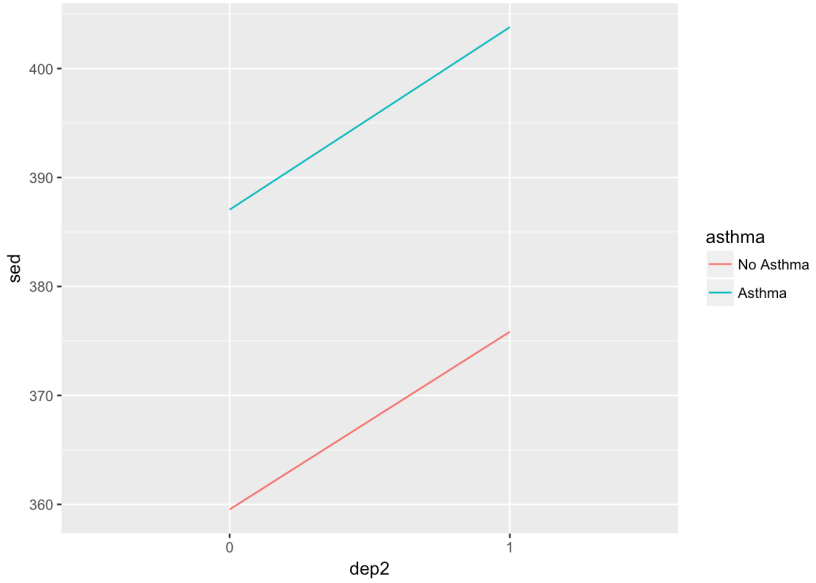
Box plots are great ways to assess the variability in your data. Below, we create a boxplot but change `p1`'s x variable so that it is the factor version of depression.

```
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 = 0.1, color = "chartreuse4") +
    geom_boxplot(alpha = 0.75, color = "dodgerblue4")
```

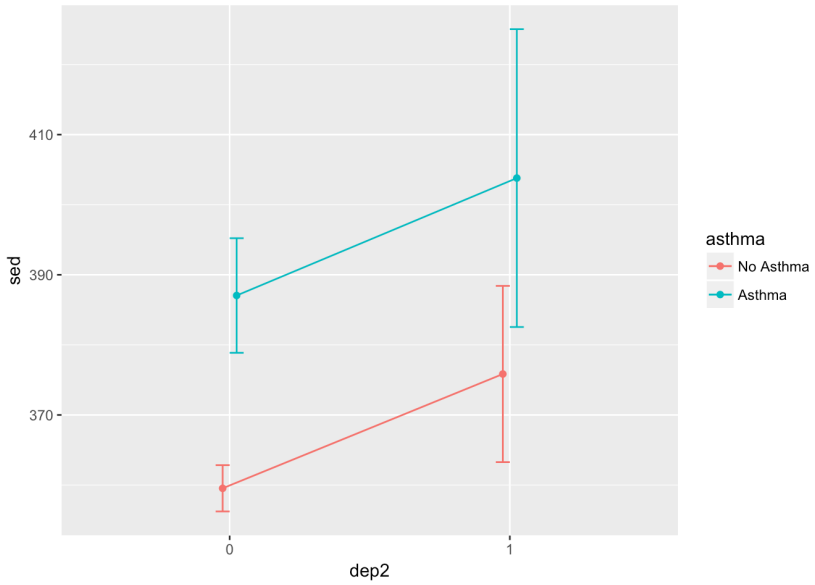



This now provides the (jittered) raw data points as well to highlight the noise and the number of observations in each group.

### *Bar Plots*

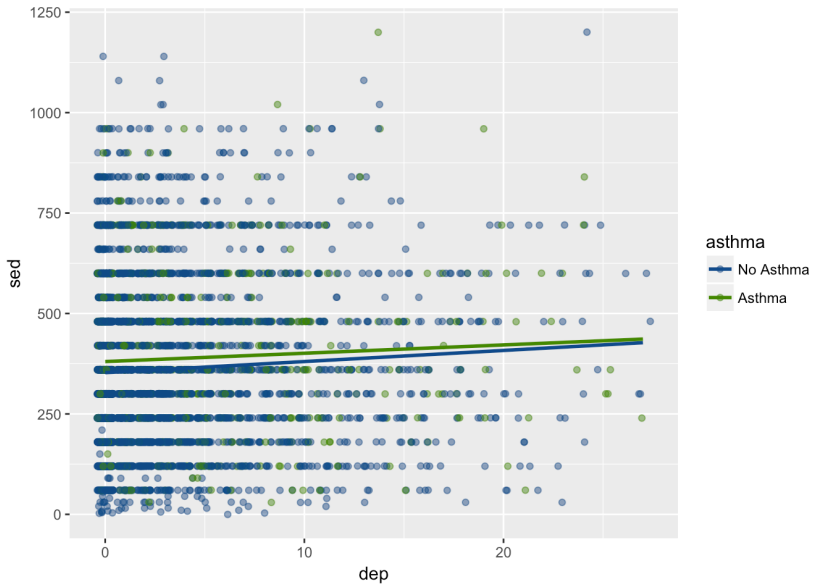
Bar plots are great ways to look at means and standard deviations for groups.

```
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 = 0.9)
ggplot(summed_data, aes(x = dep2, y = sed, group = asthma)) +
    geom_bar(aes(fill = asthma), stat = "identity",
        position = p, alpha = 0.8) + geom_errorbar(aes(ymin = sed -
    s_se, ymax = sed + s_se, color = asthma),
    position = p, width = 0.3)
```




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

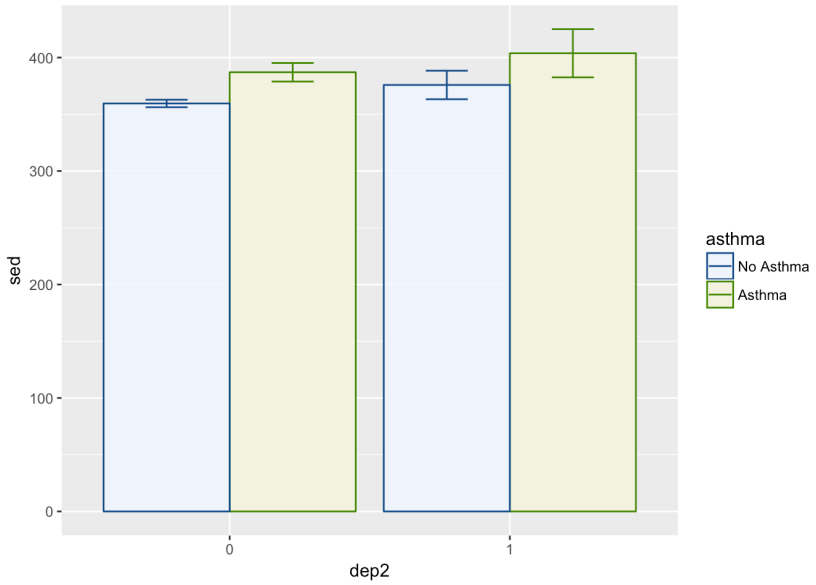
transparent so we can see the error bars better. We used the `position_dodge()` function to specify exactly how much dodge we wanted. In this way, we are able to line up the error bars and the bars. If we just use `position = "dodge"` we have less flexibility and control.

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

### *Line Plots*

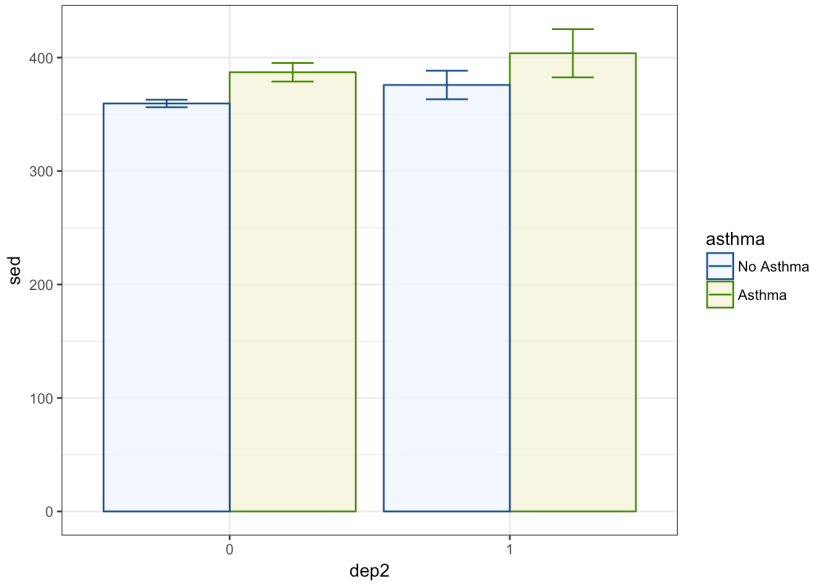
Line plots are particularly good at showing trends and relationships. Below we we use it to highlight the relationship between depression, sedentary behavior, and asthma.

```
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 = 0.1)
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 = 0.1, position = pos)
```

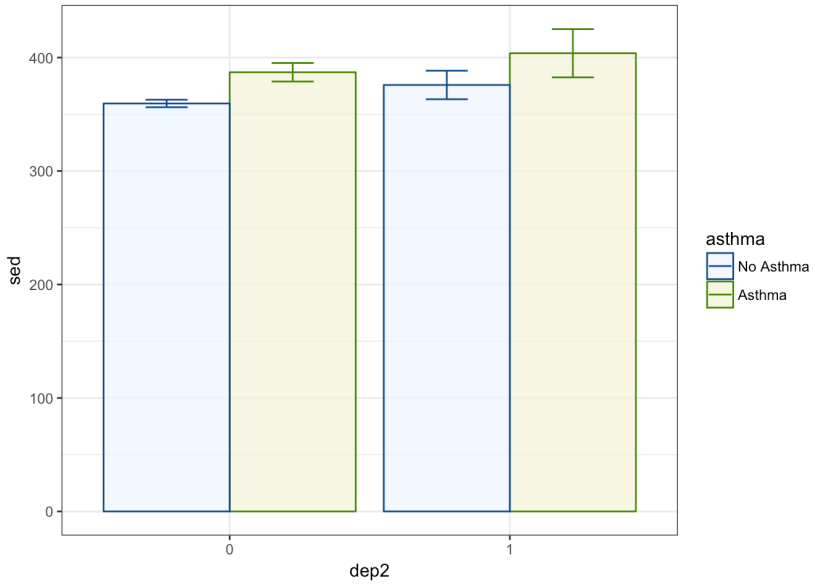



That looks a bit better. From here, let's go on to color schemes to make the plots a bit better.

## *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 = 0.5) +
    geom_smooth(aes(color = asthma), method = "lm",
        se = FALSE) + scale_color_manual(values = c("dodgerblue4",
    "chartreuse4"))
```



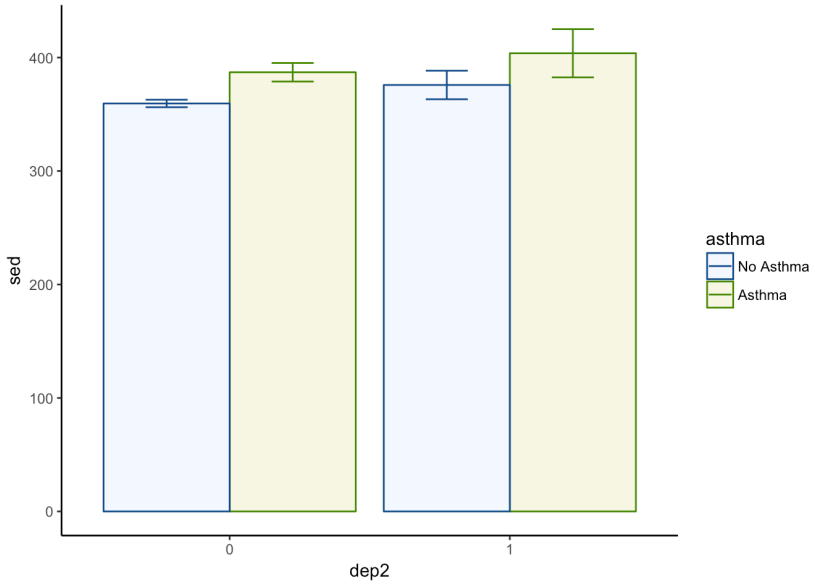

Depending on your personal taste, you can adjust it with any color. On my blog, I've posted the colors available in R (there are many).

> 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 ba
  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.

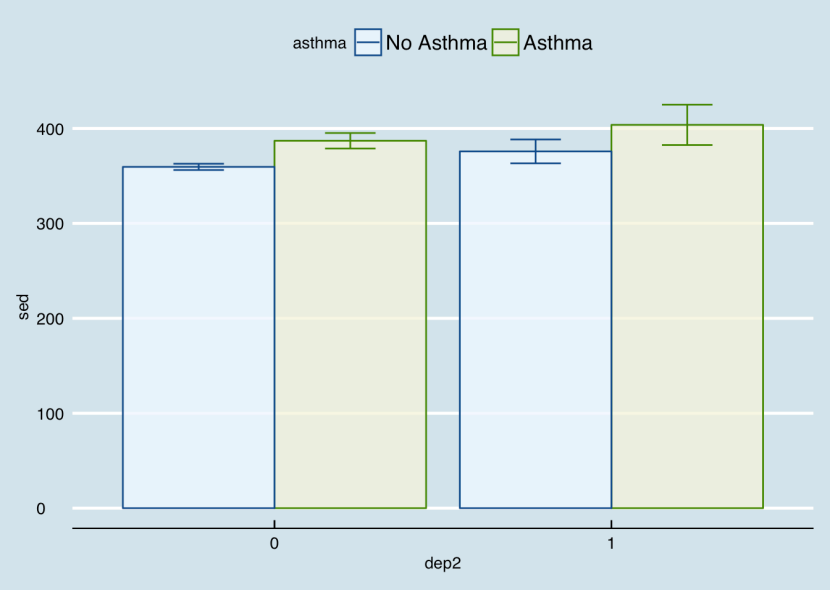
First, we are going to save the plot to simply show the different theming options.

```
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 ba
  scale_fill_manual(values = c("aliceblue", "beige"))
```

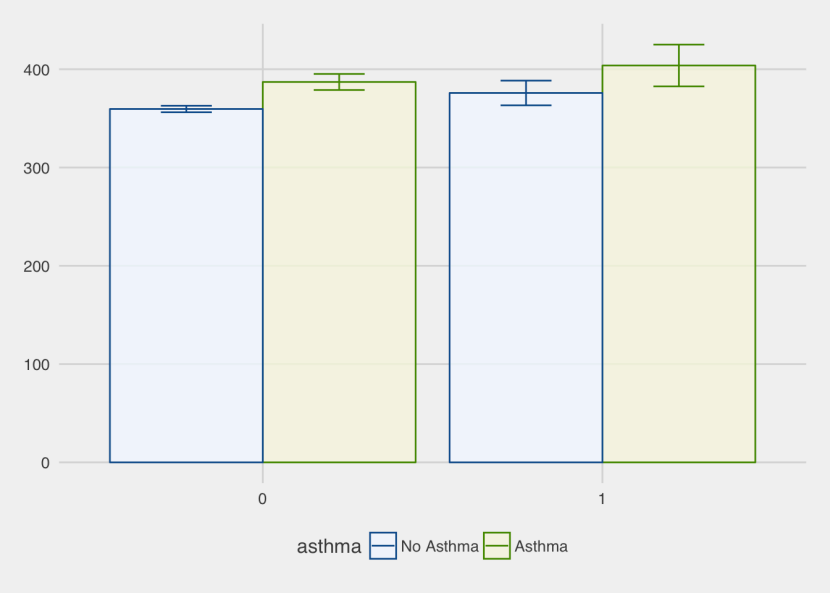
*Theme Black and White

```
p1 + theme_bw()
```



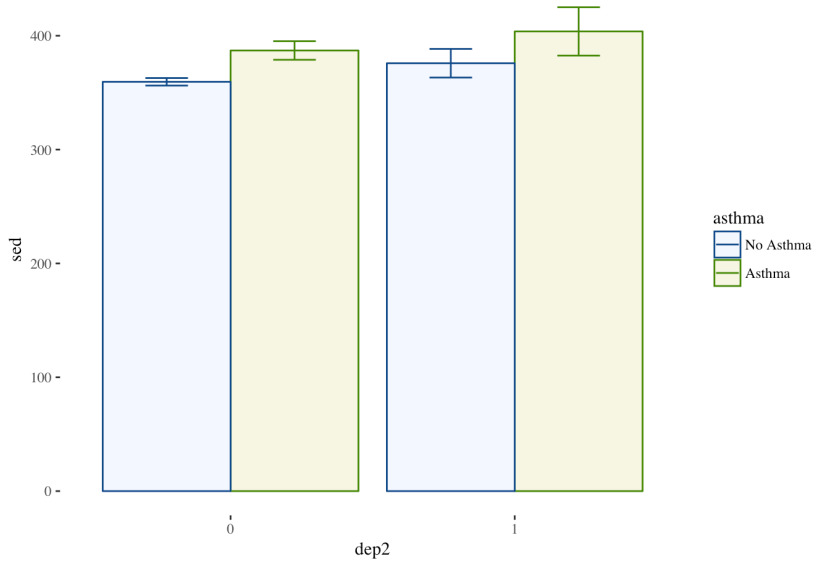

*Theme Classic

```
p1 + theme_classic()
```



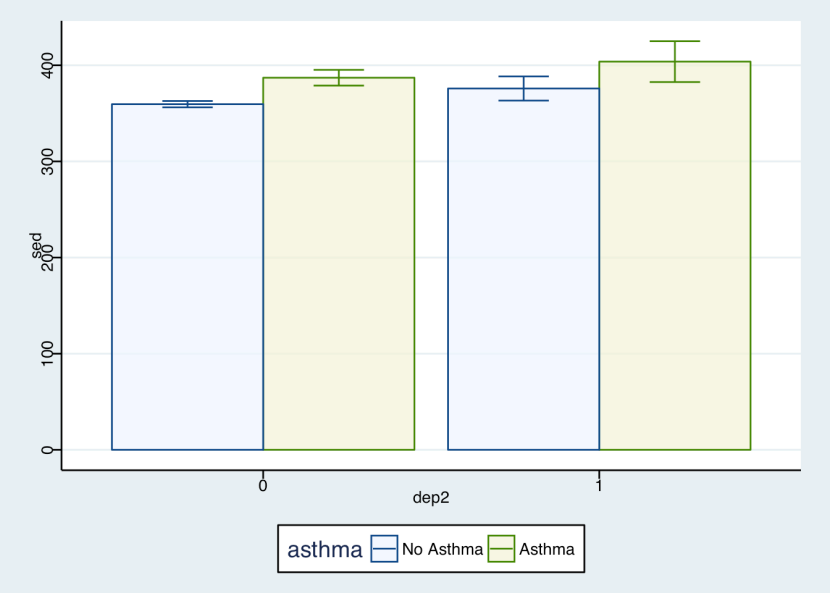

*Theme Minimal

```
p1 + theme_minimal()
```



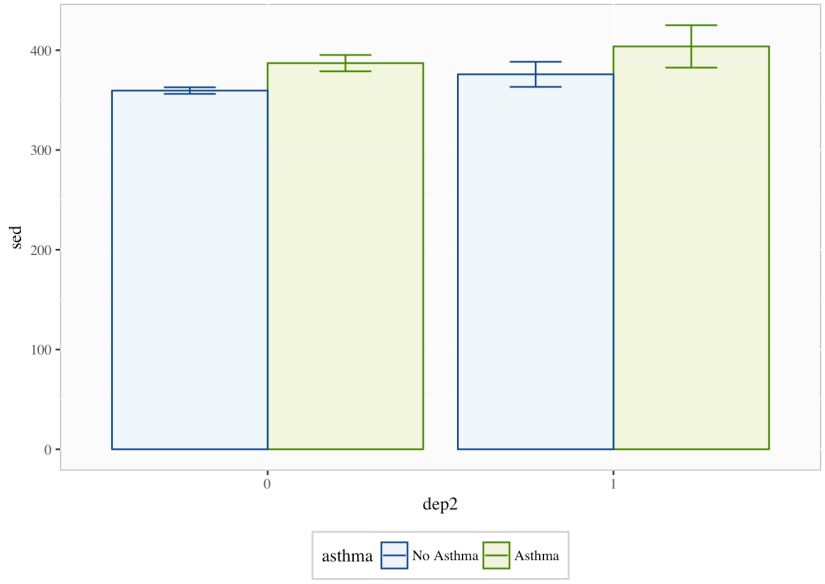

*Theme Economist (from `ggthemes`)

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



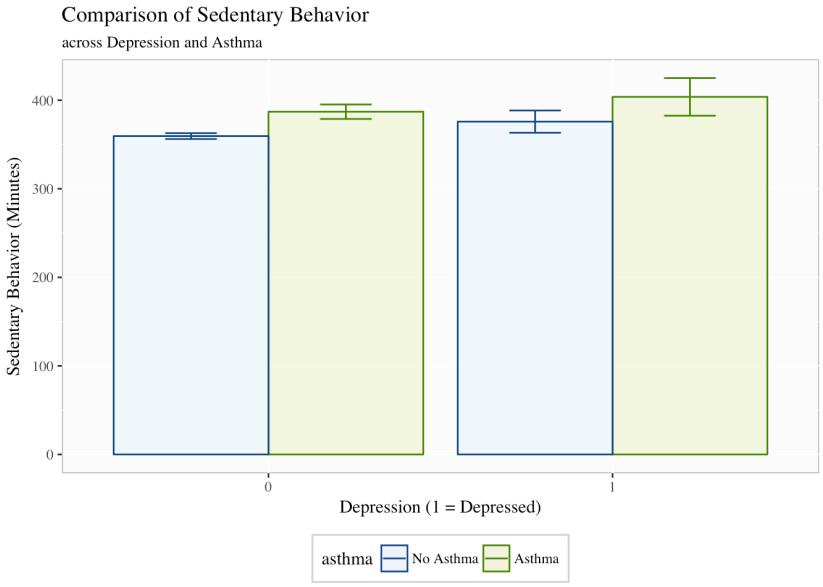

*Theme FiveThirtyEight (from `ggthemes`)

```
p1 + theme_fivethirtyeight()
```



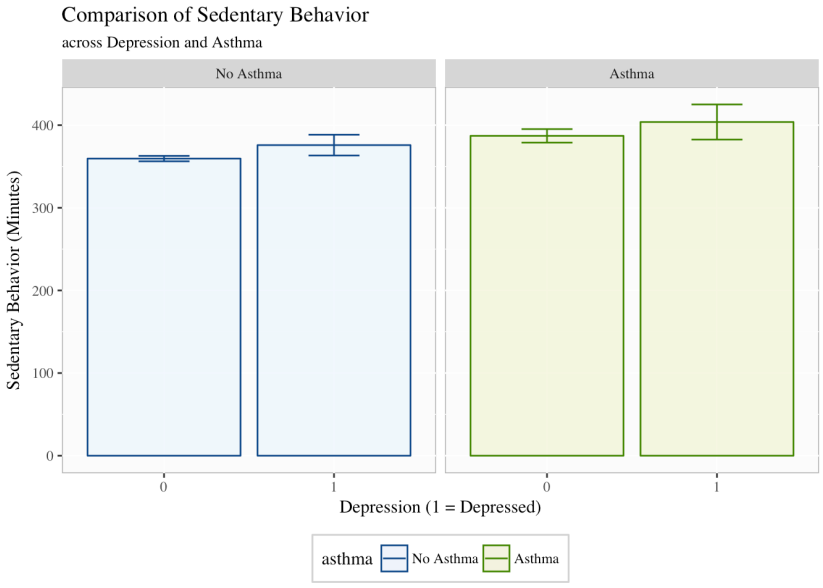

*Theme Tufte (from `ggthemes`)

```
p1 + theme_tufte()
```




*Theme Stata (from `ggthemes`)

```
p1 + theme_stata()
```




*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.

### *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 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

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 -1.56975
## 2   2 Var_Time1 -1.29243
## 3   3 Var_Time1 -0.40747
## 4   4 Var_Time1 -0.07925
## 5   5 Var_Time1  1.50440
## 6   6 Var_Time1 -0.44534
## 7   7 Var_Time1  0.75037
## 8   8 Var_Time1 -0.92503
## 9   9 Var_Time1 -1.56395
## 10 10 Var_Time1  1.59781
## 11  1 Var_Time2  0.45932
## 12  2 Var_Time2  0.96313
## 13  3 Var_Time2  0.93028
## 14  4 Var_Time2  0.12045
## 15  5 Var_Time2  0.83671
## 16  6 Var_Time2  0.67692
```

```
## 17  7 Var_Time2  0.25843
## 18  8 Var_Time2  0.17431
## 19  9 Var_Time2  0.26584
## 20 10 Var_Time2  0.79649
```

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 = 0.1)
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 = 0.1, 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)
```