Squeglia Lab:

Organic R Textbook

Ian Cero, PhD MStat

2022

Contents

Chapter 1

About

This is textbook created during live discussion with the Squeglia Research Group of the Medical University of South Carolina.

Chapter 2

Getting Started

Under most circumstances, getting started with R is a straightforward process of downloading and installing a few components. In what follows, we'll talk about what those components are, and the order in which you'll want to install them.

NOTE: the order of installation matters, so please be careful to follow the instructions in the order given below.

2.1 Overview of the R ecosystem

Most of the time you are using R for data analysis, you'll want to remember that you are working with a whole ecosystem of analysis tools. Understanding the different roles these tools serve in your project will help you keep track of the best way to use them - and hopefully make your R experience more intuitive.

The **R ecosystem** you'll be using for data analysis generally consists of three parts:

- The **R language**, which is a coding language (like Java or Python) that was optimized for talking to computers about statistical problems. When you download and install "R" (step 1, below), you are teaching your computer how to "speak" that R language.
- The **RStudio Integrated Development Environment (IDE)** is a program that you will use to make it easier to talk to your computer in the R language. Think of R as a language and RStudio as a chat app that has a bunch of features (e.g., your contacts list, spell check) that make the chat experience faster and easier for you.

- **R Packages** are collections of code that other people have written to make R perform particular tasks, usually around a them. For example, there are packages for making R perform new types of analyses, but also for streamlining data cleaning. You can download these packages with R's `install.packages()` command, so that your computer can use them too. Think of packages like special tricks you are teaching your computer. Once it learns the trick (i.e., installs the package) it can do that new trick with R over and over again, making youre live a lot easier.

2.2 Installation

2.2.1 Step 1 - Download and install the R language

The first step to a functioning R ecosystem on your computer is to install the R language on your computer. It's freely available at the Comprehensive R Archive Network (CRAN), which is an acronym you'll see a lot as we go forward. CRAN is just a group of programmers in charge of maintaining and updating the R language.

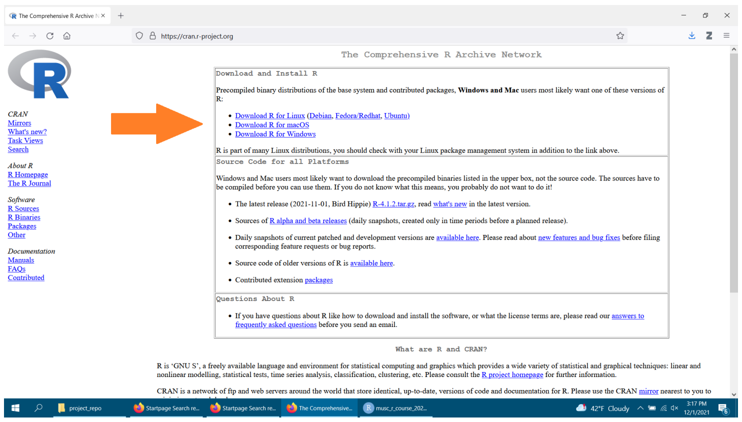
To install R, go to `https://cran.r-project.org/`. Then at the very top of the page, choose the installer that is right for your operating system (i.e., Windows, macOS, Linux).

> **HINT**: Depending on your operating system, the downloads page can be kind of intimidating. What you are looking for is the most updated version of R, which as of today (2021-12-01) is R 4.1.2. If you find that you want something to take you through the process at a more step-by-step pace, this tutorial (`https://www.datacamp.com/community/tutorials/installing-R-windows-mac-ubuntu`) should have an answer for each operating system.

2.2.2 Step 2 - Download and install the RStudio IDE

Several years ago, writing code in R was especially difficult because there was so much to keep track of and it was all hidden behind the code. The RStudio IDE fixed that for us by allowing us to continue coding in R, but this time with a collection of useful windows that keep track of what's happening in our code (e.g., what datasets do we have loaded? what plots have we generated?).

After you installed R, installing the RStudio IDE should be fairly straightforward. Just go to their Downloads page (`https://www.rstudio.com/products/rstudio/download/`) and choose the **Desktop Version**



The Comprehensive R Archive Network

CRAN
Mirrors
What's new?
Task Views
Search

About R
R Homepage
The R Journal

Software
R Sources
R Binaries
Packages
Other

Documentation
Manuals
FAQs
Contributed

Download and Install R

Precompiled binary distributions of the base system and contributed packages, **Windows and Mac** users most likely want one of these versions of R:

- Download R for Linux (Debian, Fedora/Redhat, Ubuntu)
- Download R for macOS
- Download R for Windows

R is part of many Linux distributions, you should check with your Linux package management system in addition to the link above.

Source Code for all Platforms

Windows and Mac users most likely want to download the precompiled binaries listed in the upper box, not the source code. The sources have to be compiled before you can use them. If you do not know what this means, you probably do not want to do it!

- The latest release (2021-11-01, Bird Hippie) R-4.1.2.tar.gz, read what's new in the latest version.
- Sources of R alpha and beta releases (daily snapshots, created only in time periods before a planned release).
- Daily snapshots of current patched and development versions are available here. Please read about new features and bug fixes before filing corresponding feature requests or bug reports.
- Source code of older versions of R is available here.
- Contributed extension packages

Questions About R

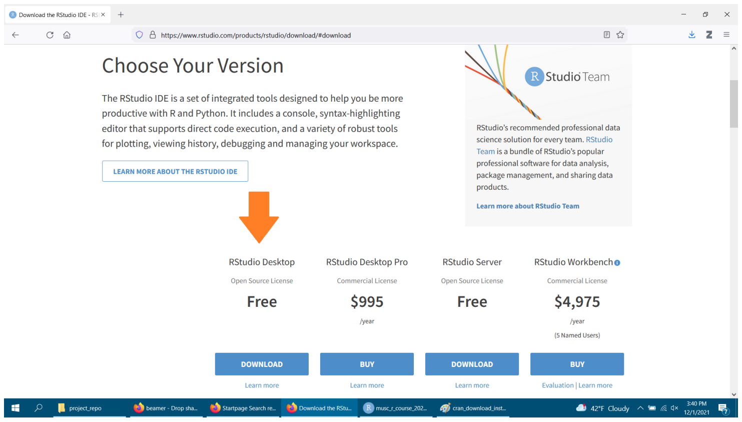
- If you have questions about R like how to download and install the software, or what the license terms are, please read our answers to frequently asked questions before you send an email.

What are R and CRAN?

R is 'GNU S', a freely available language and environment for statistical computing and graphics which provides a wide variety of statistical and graphical techniques: linear and nonlinear modelling, statistical tests, time series analysis, classification, clustering, etc. Please consult the R project homepage for further information.

CRAN is a network of ftp and web servers around the world that store identical, up-to-date, versions of code and documentation for R. Please use the CRAN mirror nearest to you to

Figure 2.1: https://cran.r-project.org/



Choose Your Version

The RStudio IDE is a set of integrated tools designed to help you be more productive with R and Python. It includes a console, syntax-highlighting editor that supports direct code execution, and a variety of robust tools for plotting, viewing history, debugging and managing your workspace.

RStudio's recommended professional data science solution for every team. RStudio Team is a bundle of RStudio's popular professional software for data analysis, package management, and sharing data products.

RStudio Desktop	RStudio Desktop Pro	RStudio Server	RStudio Workbench
Open Source License	Commercial License	Open Source License	Commercial License
Free	$995 /year	Free	$4,975 /year (5 Named Users)

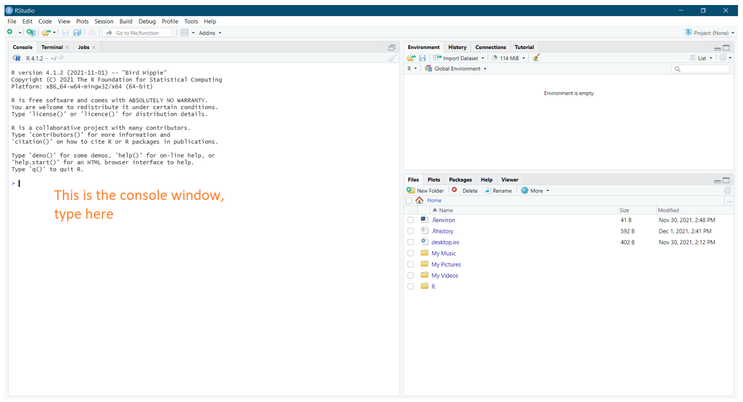
Figure 2.2: https://www.rstudio.com/products/rstudio/download/

NOTE: make sure you finish step 1 first! This will allow you to save several steps linking R and RStudio. This is because if R is installed first, RStudio will do the linking for you automatically.

2.2.3 Step 3 - Install the `tidyverse` package (optional)

Now that both R and RStudio are installed, let's open RStudio and install some packages.

1. Once you have Rstudio open, you should see several windows. Find the Console window.
2. Inside that window, type `install.packages('tidyverse')` and press ENTER.
   - R is case-sensitive, so make sure to type (or copy/paste) the command exactly.
   - This should start an installation process that takes a few minutes (no more than 10) and will install a package you will use basically every time you program in R - so it's very useful to have.
   - If you get an error message while installing, don't worry! That's pretty common and you've probably still done everything right. Just remind me in class and we will make sure to troubleshoot it for you.



```
R version 4.1.2 (2021-11-01) -- "Bird Hippie"
Copyright (C) 2021 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> 
```

Name	Size	Modified
.Renviron	41 B	Nov 30, 2021, 2:48 PM
.Rhistory	592 B	Dec 1, 2021, 2:41 PM
desktop.ini	402 B	Nov 30, 2021, 2:12 PM
My Music		
My Pictures		
My Videos		
R		

Figure 2.3: The RStudio IDE

2.3 A tour of RStudio

If R is a language, RStudio is a chat program that makes it easier to talk to your computer using that language. It includes multiple windows that help you keep track of the different parts of the conversation.

Although there are lots of tabs scattered throughout the overal RStudio application, there are generally 3 that we will use every day.

2.3.1 The console

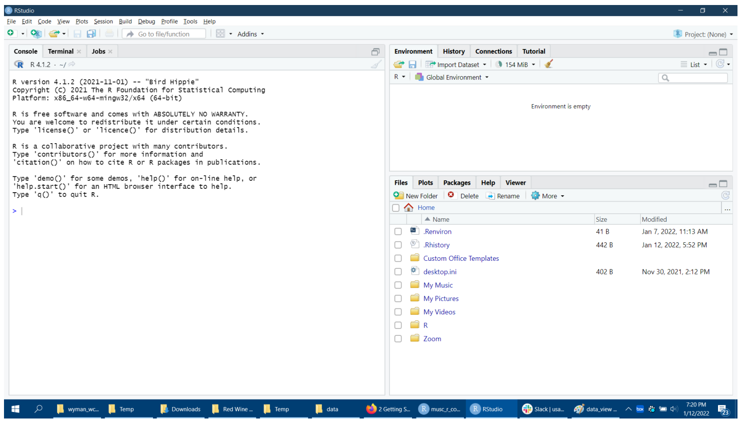
Shown in the left half (or sometimes lower left quarter) of the screen. The console is where you can talk to R live. Everything you enter into the console happens right away, which makes it really useful for quick calculations.

2.3.2 The environment

The Environment tab in the top right quadrant shows you every object you currently have imported into R. This is especially useful for keeping track of what you named your datasets (and whether your datasets even made it into R in the first place).

2.3.3 The lower right pane

There are many tabs in the lower right pane and you'll use most of them on a daily basis. The files tab shows you all the files in your current working directory (the file that R is paying attention to right now). The plots pane shows your plots, assuming you haven't told R to send them somewhere else. Lastly, the help pane will show you R's (very useful) help documentation, anytime you put a ? in front of a command (e.g., `?lm()` brings up the help file for the `lm()` command).



```
R version 4.1.2 (2021-11-01) -- "Bird Hippie"
Copyright (C) 2021 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> 
```

Name	Size	Modified
.Renviron	41 B	Jan 7, 2022, 11:13 AM
.Rhistory	442 B	Jan 12, 2022, 5:52 PM
Custom Office Templates		
desktop.ini	402 B	Nov 30, 2021, 2:12 PM
My Music		
My Pictures		
My Videos		
R		
Zoom		

Figure 2.4: The RStudio IDE

2.4 Your very first analysis

To give us a roadmap for our future work in R, we'll start with a basic analysis here. For demonstration purposes, we'll be doing a basic analysis of red wine and checking whether its chemical properties predict how well it's rated by professional tasters.

2.4.1 Step 1 - download the data

The first step is simply to download the data, **which can be downloaded here.**

You might be asked to create a Kaggle account or to log in with Google. Don't worry though, it's totally free.

2.4.2 Step 2 - Make an RStudio Project

Now that we have our data, we need a place to store it - along with all the other important things we'll be working on, like our code and analysis output. The best option is to create an RStudio Project, which is a special kind of folder that RStudio knows to keep track of. RStudio projects have a number of advantages, but for now all you need to know is that they make it easier to keep track of your data.

To make a project…

1. Navigate to File » New Project (sometimes this takes a few seconds to load after you click on it)
2. Select New Directory
3. Select New Project
4. In the Directory Name text box, write the name for your project. In this case, a good name might be something like "Red Wine Practice".
    - Note, you can change the directory you want your project folder too, but it's not necessary for this example.
    - Leave all the remaining boxes (Create git repository, Use renv with this project) **unchecked**.
5. Click Create Project

With your project now created, you should now see "Red Wine Practice" (or whatever you named your project in the top of your RStudio application window). Moreover, if you look to the Files pane on the lower right, you should see a file called `Red Wine Practice.Rproj`. Lastly, you should notice that your working directory is now called "Red Wine Practice".

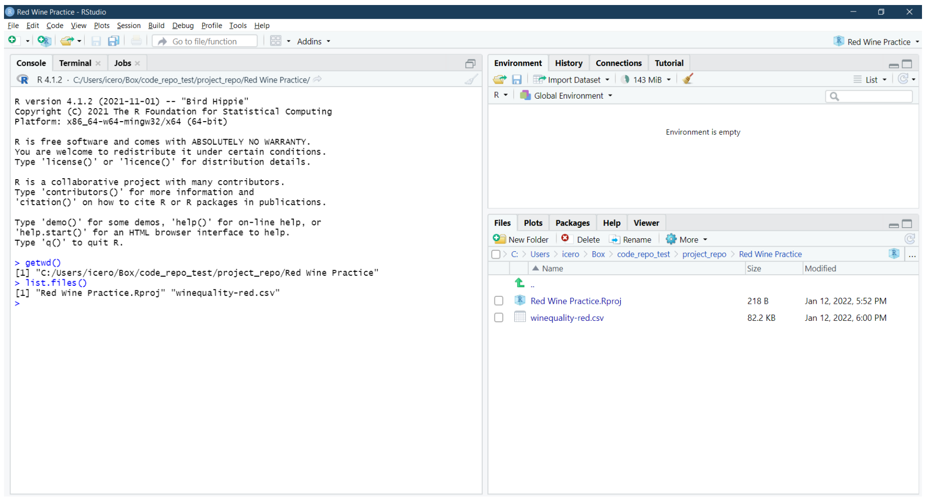
You can double-check this by typing `getwd()` (short for "get working directory") into the R Console on the bottom left and hit ENTER.

2.4.3 Step 3 - Get the data into your project folder

The quickest way to get your data into your project folder, is simply to copy/paste the `winequality-red.csv` you downloaded in Step 1 into your Red Wine Practice folder.

Where is that practice folder? Again, you can get the full path for your project folder simply by typing `getwd()` into the R console on the lower left and hitting ENTER.

To check whether your copy/paste operation worked, your can type `list.files()` into the R console. If it worked, you should see it listed along with your `Red Wine Practice.Rproj`


Red Wine Practice - RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
Red Wine Practice
Console
Terminal
Jobs
R 4.1.2 · C:/Users/icero/Box/code_repo_test/project_repo/Red Wine Practice/
Environment
History
Connections
Tutorial
Import Dataset
143 MiB
List
R
Global Environment
Environment is empty
Files
Plots
Packages
Help
Viewer
New Folder
Delete
Rename
More
C: > Users > icero > Box > code_repo_test > project_repo > Red Wine Practice

```
R version 4.1.2 (2021-11-01) -- "Bird Hippie"
Copyright (C) 2021 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> getwd()
[1] "C:/Users/icero/Box/code_repo_test/project_repo/Red Wine Practice"
> list.files()
[1] "Red Wine Practice.Rproj" "winequality-red.csv"
>
```

Name	Size	Modified
..		
Red Wine Practice.Rproj	218 B	Jan 12, 2022, 5:52 PM
winequality-red.csv	82.2 KB	Jan 12, 2022, 6:00 PM

Figure 2.5: Checking the copy/paste operation worked with 'list.files()'

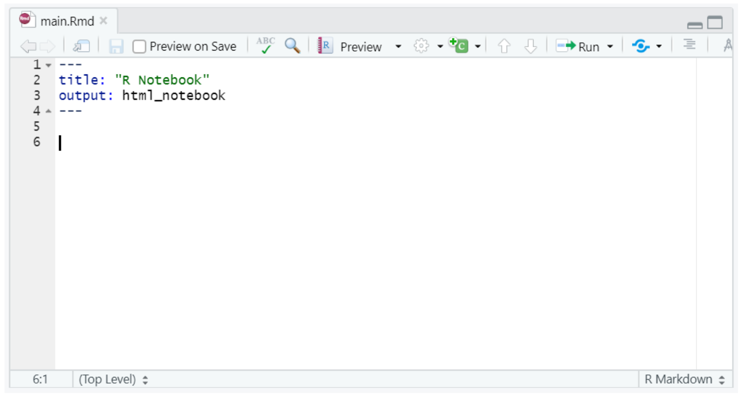
2.4.4 Step 4 - Open an Rmarkdown Notebook

We need a place to type our R commands, plus some notes to ourselves. The best way to do both of those things at the same time as an Rmarkdown notebook.

1. Navigate to File » New file » R Notebook

    - **NOTE**: You might get a message asking if you want to install some packages. Press OK / Yes - you *do* want to install them.

2. With the new notebook file opened, press CTRL+S to save the file under a different name. You can use whatever name you want. For this example, I will use `main.Rmd` to remind myself this is my main analysis file.
3. Inside your newly saved file, change the title from "R Notebook" to something more descriptive like "My first wine analysis".
4. Lastly, RStudio gave us a bunch of boilerplate code. We won't need that today, so delete everything below the second `---` at the top of the page, right under `output: html_notebook`. Your final document should then look something like the following.



```
1 ---
2 title: "R Notebook"
3 output: html_notebook
4 ---
5
6 
```

Figure 2.6: What your Rmarkdown file should look like before we start coding

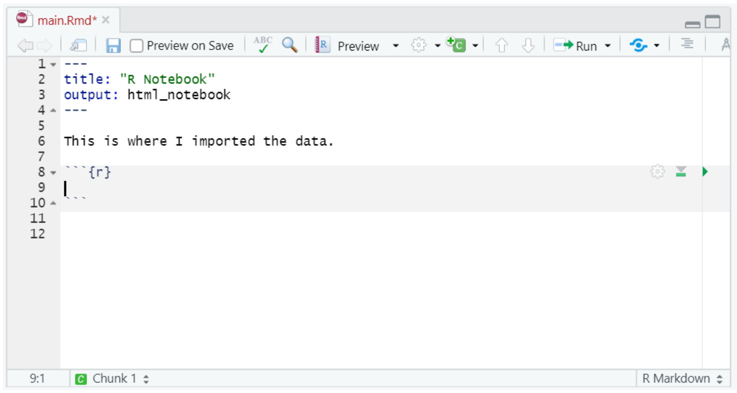
2.4.5 Step 5 - Create a space to code

Rmarkdown documents have whitespace and greyspace. Whitespace is where you type notes to yourself. Greyspace is where you type R commands. We call these greyspaces **code blocks**.

1. Anywhere below line 4, type a note to yourself like "This is where I imported the data."
2. Move your curser to a line below that note you just wrote (e.g., line 8). Then, press CTRL+ALT+I. This will create a grey codeblock.

2.4.6 Step 6 - Write a command to import the data

Importing data involves four things: the name of the datafile, a command to read the data into R, the "assignment operator" (written as `<-`), and the name you want R to call the imported data when you reference it later. Fortunately, this *sounds* much more complicated than it is.



```
1  ---
2  title: "R Notebook"
3  output: html_notebook
4  ---
5
6  This is where I imported the data.
7
8  ```{r}
9
10 ```
11
12
```

Figure 2.7: Adding a note to yourself and a grey code block

For now, just copy and paste the following commmand into the middle of your grey code block (for me, that is line 9).

```r
my_data <- read.csv('winequality-red.csv')
```

This is R code. It can be translated into English, but it's a little clunky. Also, generally code works from right to left. So, you would read this sentence as something like "Take the file called `winequality-red.csv`, read it into R, then call whatever comes out of that process `my_data`."

To get the R code to run, put your cursor inside the grey code block and press CTRL+SHIFT+ENTER. This runs all the code inside that block.

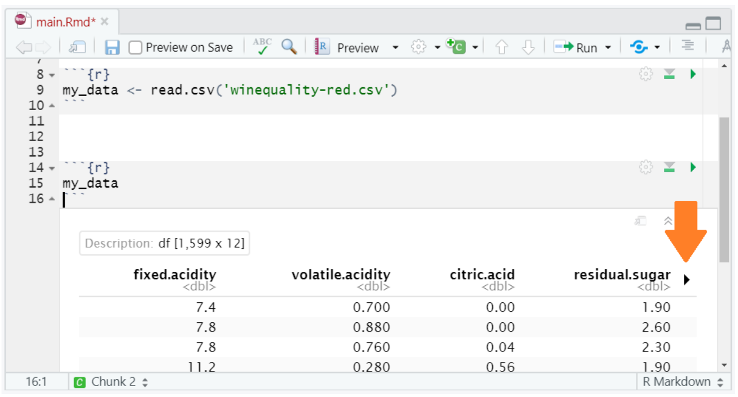
2.4.7 Step 7 - Look inside the data

Make a new code block a few lines down from the last one. Then type just the name of your imported data and run the block. You should see a preview of your dataset. To scroll through the other variables in the dataset, press the right-facing triangle on the right side of the preview.

2.4.8 Step 8 - Make a histogram

I think I'm most interested in the final variable in the dataset, `quality`. In this case, that is the quality of the wine - rated by professionals on a scale of 1 to 10. Let's see what the distribution looks like. For that we can use the `hist()` command (short for "histogram").

But what should we give our `hist()` command? We unfortunately can't give it the whole dataset. After all, we only want a histogram of one variable. How do



~~~
```{r}
my_data <- read.csv('winequality-red.csv')
```



```{r}
my_data
```
~~~

Description: df [1,599 x 12]

fixed.acidity <dbl>	volatile.acidity <dbl>	citric.acid <dbl>	residual.sugar <dbl>
7.4	0.700	0.00	1.90
7.8	0.880	0.00	2.60
7.8	0.760	0.04	2.30
11.2	0.280	0.56	1.90

Figure 2.8: Preview of the data

we specify that variable? We use the "selection operator", which we write as a $-symbol, like below.

```
hist(my_data$quality)
```

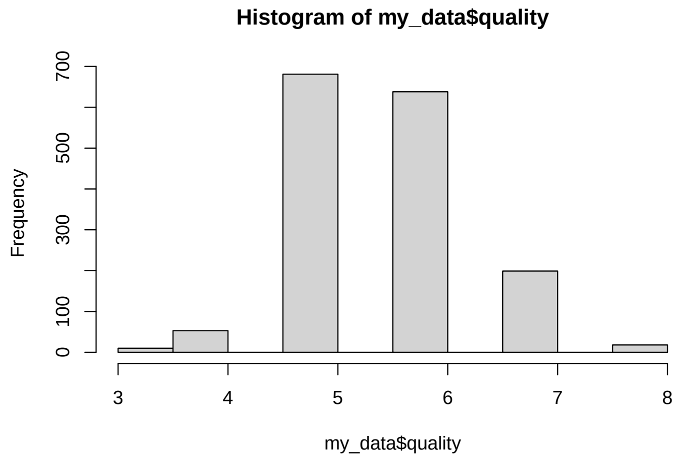


Figure 2.9: Histogram of wine quality ratings

2.4.9 Step 9 - Run a regression

Now that we have a sense of what our outcome variable (`quality`) looks like, let's see if we can investigate some of the chemical characteristics in wine associated with that outcome.

The ones that stick out to me are `pH` (acidity) and `alcohol` content because they seem like things that would really affect the taste. Let's use those as our predictors

To run a regression, we need just a few things (out of order): - The `lm()` command, which is short for "linear model". This is how R will know we want a regression. - Our data, named `my_data` - A regression formula, which tells R what the outcome variable and it's predictors are - The assignment operator again (`<-`), which tells us where to store the results - A name for where to store the results

Putting all of that together looks like this. One you have it all typed in (or copy/pasted), run the whole block with CTRL+SHIFT+ENTER.

```
my_results <- lm(
  formula = quality ~ pH + alcohol,
  data = my_data)
```

2.4.10 Step 10 - Get a summary of your results

You may have noticed that in Step 9, your results didn't show up anywhere after you ran your regression. That's because R stored them in `my_results`, just like it stored the outcome of the `read.csv()` command in `my_data`.

This often surprises people who come from other software packages, but that's okay. Our results are still easy to get. We just need to ask R for a summary of them.

```
summary(my_results)
```

```
##
## Call:
## lm(formula = quality ~ pH + alcohol, data = my_data)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -2.7153 -0.4066 -0.1105  0.5076  2.4584
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  4.42581    0.38742  11.424  < 2e-16 ***
## pH          -0.85011    0.11571  -7.347 3.23e-13 ***
## alcohol      0.38617    0.01676  23.036  < 2e-16 ***
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.6989 on 1596 degrees of freedom
## Multiple R-squared:  0.252,  Adjusted R-squared:  0.2511
## F-statistic: 268.9 on 2 and 1596 DF,  p-value: < 2.2e-16
```

This gives us a basic regression table with all of the same information you are used to. Under the *Coefficients* heading, we see the b-values / slopes (Estimate column), their standard errors, t-values, and p-values.

Next to the p-values, we see stars reminding us that our p-values are significant at the $< .001$ level. In fact, our p-values are so small, they have to be shown in scientific notation ("3.23e-13"). These can be treated as basically zero.

Lastly, at the very bottom, we also see the usual R-squared values (here, .252), our F-statistic, and degrees of freedom - everything we need to create a publication-ready regression table.

2.4.11 A Look forward

Over the last 10 steps, we ran through a basic version of essentially every R analysis you are likely to conduct in the future. This is example thus contains a useful workflow you will likely want to recreate in your future work:

- Create a project to store everything
- Import the data
- Visualize / explore the data
- Run main analysis model
- Summarize results

Although the data and models you use might be more complicated as your time in R progresses, it's helpful to remember that everything your are doing typically reduces to these fundamental steps.

Chapter 3

Basic R

3.1 Writing in R and Rmarkdown

3.1.1 Chatting with R

Using R is just a chat with the computer.

"Hey, R. What is $1 + 2$?"

```
1 + 2
```

```
## [1] 3
```

3.1.2 Rmarkdown tricks

- To make text **bold**, we add two **s around it.
- To make text *italicized*, we add just one * around it.
- If we need special characters (like * or $), then we just add a forward "\" in front of them (but not behind).
- Math symbols in your text are process with Latex, just put an "$" before and after your math. Like this, $y = x$ becomes $y = x$.

3.1.3 Code blocks

To make a code block, press CTRL+ALT+I.

```
banana <- 5
banana + 1
```

```
## [1] 6
```

3.2 Variables

Variables are values that I want to give names to and save for later.

3.2.1 The assignment operator

We make variables with the `<-` operator. This is called the *assignment operator* because it assigns values on the right to names on the left. If I want to know what the value of a variable is, I can run it alone on its own line.

```
my_special_var <- 1 + 2
my_special_var
```

```
## [1] 3
```

You can TECHNICALLY use `=` for assignment too. Never do this.

```
my_other_var = 12
my_other_var + my_special_var
```

```
## [1] 15
```

The `=` symbol gets also used for a few other things in R. So, using it to assign variables will make your code more confusing to you, when you go back to read it over later.

3.2.2 Numerics

Doubles

Doubles are decimal numbers, like $1.1, 2.2, 3.0$. If I make a number variable without doing anything special, R defaults to a double.

```
a <- 1.1
b <- 2.0
is.double(a)
```

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

```
is.double(b)
```

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

Integers

Integers must have an `L` after them. That is how R knows that you don't want a double, but instead want a "long-capable integer".

```
c <- 1L
d <- 1
is.integer(c)
```

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

```
is.integer(d)
```

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

Here is a useful cheatsheet for the different numeric operators and how they behave.

Operator	Expression	Result
+	10 + 3	13
-	10 - 3	7
*	10 * 3	30
/	10 / 3	3.333
^	10 ^ 3	1000
%/%	10 %/% 3	3
%%	10 %% 3	1

Why care about the difference?

Almost 99% of the time, this wont matter. But, with big data, integers take up must less memory.

```
my_integers <- seq(from = 1L, to = 1e6L, by = 1L)
my_doubles <- seq(from = 1.0, to = 1e6, by = 1.0)
object.size(my_integers)
```

```
## 4000048 bytes
```

```
object.size(my_doubles)
```

```
## 8000048 bytes
```

Note here that although we are using only whole numbers from 1 to 1 million, the first sequence (`my_integers`) is stored as an integer and the second sequence (`my_doubles`) is stored as a number that may include decimals. This second case needs more space (twice as much) to be allocated in advance, even if we never use those decimal places.

Again, this will almost never matter for most people, most of the time. However, it is good to be aware of for when your datasets get large (i.e., several million cases or more).

3.2.3 Characters

Characters are text symbols and they are made with either ”” or ”, either works.

```
a <- 'here is someone\'s text'
b <- "here is more text"
a
```

```
## [1] "here is someone's text"
```

```
b
```

```
## [1] "here is more text"
```

To combine two strings, I use `paste()`.

```
paste(a, b)
```

```
## [1] "here is someone's text here is more text"
```

If I dont want a space, then I used `paste0()`.

```
paste0(a, b)
```

```
## [1] "here is someone's texthere is more text"
```

3.2.4 Booleans

These are True and False values. You make them with the symbols `T` or `TRUE` and `F` or `FALSE`.

```
x <- T
y <- F
```

To compare them, we can use three operators.

- `&` is "and"
- `|` is "or"
- `!` is "not" (just give me the opposite of whatever is after me)

```
x & y # false
```

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

```
x | y # true
```

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

```
x & !y # true
```

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

We can also have nested equations

```
z <- F
x & !(y | z) # true
```

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

We can also compare numbers.

```
a <- 1
b <- 2
```

```
a < 1
```

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

```r
a <= 1
```

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

```r
a == 1
```

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

If I want to compare multiple numbers, I need to do it seperately.

```r
(a > 1) | (b > 1)
```

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

Remember that booleans are ultimately numeric values underneath.

```r
d <- T
k <- F
u <- 5
d*u
```

```
## [1] 5
```

```r
d*k
```

```
## [1] 0
```

```r
as.numeric(d)
```

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

```r
as.numeric(k)
```

```
## [1] 0
```

3.2.5 Special types

`NA` - missing

```
is.na(NA)
```

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

`NaN` - you did math wrong

```
0/0
```

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

`Inf` - infinity

```
-5/0
```

```
## [1] -Inf
```

3.3 Vectors

R is built is on vectors. Vectors are collections of a bunch of values of the same type.

```
my_vec <- c(1, 5, 3, 7)
my_vec
```

```
## [1] 1 5 3 7
```

If I try to put different types together, they go to the most primitive type (usually a character string).

```
my_other_vec <- c(22, 'orange', T)
my_other_vec
```

```
## [1] "22"     "orange" "TRUE"
```

```
my_third_vec <- c(T, F, 35)
my_third_vec
```

```
## [1]  1  0 35
```

We can also missing values.

```
my_fourth_vec <- c(1, 4, 5, NA)
my_fourth_vec
```

```
## [1]  1  4  5 NA
```

```
is.na(my_fourth_vec)
```

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

If I want to combine two vectors…

```
a <- c(1, 2, 3)
b <- c(3, 5, 7)
c(a, b)
```

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

A brief example of matrices

```
matrix(
  data = c(a, b),
  nrow = 2,
  byrow = T)
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    3    5    7
```

Sometimes I want special vectors, direct sequences of numbers. There are two ways to do this. If all I want is a integer sequence (made of doubles), then I use the "`<first number>:<last number>`".

```
1:5
```

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

```
5:1
```

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

Other times, I need to count by something other than one, so I use `seq(from = <start>, to = <end>, by = <number to count by>)`

```
seq(from = 1, to = 7, by = 1.3)
```

```
## [1] 1.0 2.3 3.6 4.9 6.2
```

Hint: for brevity, I can leave off function parameter names, **as long as I enter them in order**

```
seq(1, 7, by = 1.3)
```

```
## [1] 1.0 2.3 3.6 4.9 6.2
```

If I add a constant to a vector, then they all go up by that constant.

```
1:5 / 3
```

```
## [1] 0.3333333 0.6666667 1.0000000 1.3333333 1.6666667
```

I can do math with equal-length sequences too.

```
1:5 - seq(1, 4, by = .7)
```

```
## [1] 0.0 0.3 0.6 0.9 1.2
```

But they **must** be equal lengths.

```
1:5 / 1:4
```

```
## Warning in 1:5/1:4: longer object length is not a multiple
## of shorter object length
```

```
## [1] 1 1 1 1 5
```

To access the elements of a vector, I put a number OR booleans in brackets `[]`.

```
my_vec <- c('apple', 'orange', 'banana', 'pair')
my_vec[2]
```

```
## [1] "orange"
```

```
my_vec[2:4]
```

```
## [1] "orange" "banana" "pair"
```

```
my_vec[c(3, 2, 1, 4)]
```

```
## [1] "banana" "orange" "apple"  "pair"
```

I can also use bools.

```
my_other_vec <- c(1, 4, 6, 7, 9, 3, 9)
my_other_vec < 5
```

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

```
my_other_vec[my_other_vec < 5]
```

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

I can also use functions that return values to access vectors, if I am creative...

```
my_other_vec[max(my_other_vec) == my_other_vec]
```

```
## [1] 9 9
```

R also has special vectors that are pre-loaded. The most commonly used are `letters` and `LETTERS`, which return the lower-case letters and uppercase letters of the English alphabet, respectively.

```
vec <- c(1, 3, 4, 5, 3, 2, NA)
mean(vec, na.rm = T)
```

```
## [1] 3
```

3.4 Lists

« More on lists to come »

Lists are special vectors that can hold multiple types of elements, even vectors

```
my_vec <- c(4, 5, 6)
my_list <- list(1, 'banana', 3, NA, my_vec)
my_list
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] "banana"
##
## [[3]]
## [1] 3
##
## [[4]]
## [1] NA
##
## [[5]]
## [1] 4 5 6
```

3.5 Dataframes

3.5.1 Construction

Dataframes are spreadsheets. Under the hood of R, they are just lists of vectors, where all the vectors are required to be the same length. To make one, you can call the `data.frame()` function and put your vectors inside.

```
heights <- c(60, 65, 71, 72, 64)
sexes <- c('female', 'female', 'male', 'male', 'female')
shoes <- c('Adidas', 'Nike', 'Nike', 'Salvatore Ferragamo', 'Reebok')
df <- data.frame(height = heights, sex = sexes, shoes = shoes)
df
```

```
##   height    sex               shoes
## 1     60 female              Adidas
## 2     65 female                Nike
## 3     71   male                Nike
## 4     72   male Salvatore Ferragamo
## 5     64 female              Reebok
```

3.5.2 Built-in dataframes

R has numerous built-in datasets that are ideal for demonstration purposes. We can get access to them using the `data()` command. This will load the data into our session, so we can then look at it.

```
data('mtcars')
mtcars
```

```
##                      mpg cyl  disp  hp drat    wt  qsec vs
## Mazda RX4           21.0   6 160.0 110 3.90 2.620 16.46  0
## Mazda RX4 Wag       21.0   6 160.0 110 3.90 2.875 17.02  0
## Datsun 710          22.8   4 108.0  93 3.85 2.320 18.61  1
## Hornet 4 Drive      21.4   6 258.0 110 3.08 3.215 19.44  1
## Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0
## Valiant             18.1   6 225.0 105 2.76 3.460 20.22  1
## Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0
## Merc 240D           24.4   4 146.7  62 3.69 3.190 20.00  1
## Merc 230            22.8   4 140.8  95 3.92 3.150 22.90  1
## Merc 280            19.2   6 167.6 123 3.92 3.440 18.30  1
## Merc 280C           17.8   6 167.6 123 3.92 3.440 18.90  1
## Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0
## Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0
## Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0
## Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0
## Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0
## Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0
## Fiat 128            32.4   4  78.7  66 4.08 2.200 19.47  1
## Honda Civic         30.4   4  75.7  52 4.93 1.615 18.52  1
## Toyota Corolla      33.9   4  71.1  65 4.22 1.835 19.90  1
## Toyota Corona       21.5   4 120.1  97 3.70 2.465 20.01  1
## Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0
## AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0
## Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0
## Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0
## Fiat X1-9           27.3   4  79.0  66 4.08 1.935 18.90  1
## Porsche 914-2       26.0   4 120.3  91 4.43 2.140 16.70  0
## Lotus Europa        30.4   4  95.1 113 3.77 1.513 16.90  1
## Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0
## Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0
## Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0
## Volvo 142E          21.4   4 121.0 109 4.11 2.780 18.60  1
##                     am gear carb
## Mazda RX4            1    4    4
## Mazda RX4 Wag        1    4    4
## Datsun 710           1    4    1
## Hornet 4 Drive       0    3    1
## Hornet Sportabout    0    3    2
## Valiant              0    3    1
## Duster 360           0    3    4
## Merc 240D            0    4    2
## Merc 230             0    4    2
## Merc 280             0    4    4
## Merc 280C            0    4    4
## Merc 450SE           0    3    3
## Merc 450SL           0    3    3
## Merc 450SLC          0    3    3
## Cadillac Fleetwood   0    3    4
## Lincoln Continental  0    3    4
## Chrysler Imperial    0    3    4
## Fiat 128             1    4    1
## Honda Civic          1    4    2
## Toyota Corolla       1    4    1
## Toyota Corona        0    3    1
## Dodge Challenger     0    3    2
## AMC Javelin          0    3    2
## Camaro Z28           0    3    4
## Pontiac Firebird     0    3    2
## Fiat X1-9            1    4    1
## Porsche 914-2        1    5    2
## Lotus Europa         1    5    2
## Ford Pantera L       1    5    4
## Ferrari Dino         1    5    6
## Maserati Bora        1    5    8
## Volvo 142E           1    4    2
```

Some datasets do not come in the form of a dataframe right away, but they can be converted into one using the `as.data.frame()` function.

```
data(Seatbelts)
is.data.frame(Seatbelts)
```

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

```
seatbelts_df <- as.data.frame(Seatbelts)
is.data.frame(seatbelts_df)
```

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

3.6 Functions

A function is a piece of code that does work for you. It takes inputs and (usually) returns outputs. For example, the `sum()` function takes the sum of a numeric vector.

```
my_vec <- c(3, 6, 2, 3)
sum(my_vec)
```

```
## [1] 14
```

3.6.1 Getting help

If I ever need to know something about a function, I can put a question mark in front of it (no `()`s) and run that line. That will bring up the help document for that function.

```
?sum
```

3.6.2 Function parameters

In addition to the data they take as input, most functions have additional *parameters* (sometimes called "arguments", but they mean the same thing). Looking at its help file, the `sum()` function has two parameters:

- `...`, the numbers you want to sum
- `na.rm = FALSE`, which tells `sum()` whether you want to remove ('rm') missing values ('na') before summing.

Let's look at what happens when we try to `sum()` a vector with a missing value.

```
my_vec <- c(5, NA, 2, 3) # should be 10
sum(my_vec)
```

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

R tells us the answer is missing (`NA`) because at least one of the vector elements is missing. This is to be conservative and to force you never to ignore missing values by accident. But what do we do if we really do want to sum all available values, ignoring the missing values.

Again, looking at the help file, we can see that the `na.rm` parameter of the function is followed by `= FALSE`, under the **Usage** heading of that help document (look for `sum(..., na.rm = FALSE)`). This tells us that the parameter `na.rm`, which tells `sum()` whether to remove missing values from the calulation, defaults to `FALSE`.

To get `sum()` to ignore the missing values in our vector, we simply set `na.rm` to `TRUE` (or `T` for short).

```
sum(my_vec, na.rm = T) # should be 10
```

```
## [1] 10
```

3.7 Packages

Packages are collections of functions that someone else put together for you. You can install them using the `install.packages()` function, with the name of your package inside the `()` - don't forget to use either single (`' '`) or double quotes (`" "`) around the package name too.

```
install.packages('ggplot2')
```

Once installed, use the `library()` function to load your package into your R session. Note, you don't need quotes here.

```
library(ggplot2)
```

3.8 Error messages

Whenever R detects that something has gone wrong, it will send you an error message in the form of some scary-looking red text.

```
'100' / '2' # trying to divide two strings
```

```
## Error in "100"/"2": non-numeric argument to binary operator
```

Unfortunately, R is not especially smart and is usually bad at detecting exactly WHAT has gone wrong. In this case, all it knows is that `/` needs two numbers - one on either side - to work correctly. It detected something other than that, which is what it told us: there was a non-numeric argument (input) SOMEWHERE on either side of the `/` symbol.

Stumbling on an error and getting stuck is especially common. It's also common to get frustrated when you are stuck with the same error for more than a few minutes. For that reason, if you can't solve an error after a few quick tries, it's best NOT to beat your head against the wall. Instead, go to a place like **https://stackoverflow.com/**, which is a website devoted entirely to answering questions - most of which are about coding errors just like yours.

Simply copy and paste your error into the search box and look for someone who asked your question already. You'll notice that many people have had your same problem and have even produced some code you can copy/paste to fix your current issue.

3.9 Coding Conventions

R is a language, much like English or Spanish. It sometimes has rules for how you MUST say something in order for your computer to understand at all. For example, R won't let you add a number and a letter together because that wouldn't make sense mathematically.

```
1 + 'a'
```

```
## Error in 1 + "a": non-numeric argument to binary operator
```

Other times, R will let you do the same thing in more than one way. For example, I can name my variables with a mixture of capitals and lower-case letters.

```
apple <- 123
BANANANA <- 456
ClEmEnTiNe <- 789
```

Some of these options might be more confusing than others, but R will technically let you do them.

Other examples include using `<-` or `=` to assign values to variables. As discussed above, both with work.

```
peas <- 'tasty'
carrots = 'also tasty'
```

3.9.1 What should my code look like?

Whenever there are multiple options for how to code, it's worth thinking about whether one will be better for you than the others. If you come up with a consistent rule over time - like “never use capital letters in function names” - you’ve developed some **coding conventions**. These achieve a few things for you, but mostly we develop these informal language rules for clarity.

They make it easier for us to read our code, and for others to understand what we were doing when they look at our code later. Some common coding conventions most R users now employ are given below.

3.9.1.1 Common coding conventions in R

Never use `=` to assign variable values. Use the `<-` operator instead because it is more clear.

```r
a = 2 # bad

a <- 2 # good
```

Avoid using `.` to seperate words in your variable and function names because this makes it hard for people who come from other coding langauages to understand us. Use `_` instead.

```r
my.favorite.number <- 3.14159 # bad

my_favorite_number <- 3.14159 # good
```

Whenever possible, stick to lower-case variable names. It will make it easier for you to reference your variables later, without accidentally making a capitalization error.

```r
apple <- 123 # good
BANANANA <- 456 # bad
ClEmEnTiNe <- 789 # bad
```

Whenever possible, try to use single-quotes (`'`) for character strings, rather than double quotes (`"`). Single quotes are easier on your eyes when you are looking at a page full of code.

```r
peas <- "tasty" # bad
carrots = 'also tasty' # good
```

3.9.2 Official coding conventions

Coding conventions are so important, that many people have tried to publish some. You can think of these like stlye guides that many people agreed to use. The one most relevant to our own work here is Hadley Wickham’s guide at **https://style.tidyverse.org/**.

Chapter 4

Working with ABCD

In this section, we'll add specific content for the ABCD dataset issues as we uncover them over time.

4.1 Importing ABCD datafiles

One of the trickiest parts of working with the ABCD dataset is just getting the data into R. Under most non-ABCD circumstances, getting a file into R is as simple as loading the `tidyverse` and telling the `read_csv()` function where to look for your data.

```
library(tidyverse)

wine_df <- read_csv('data/winequality-red.csv')

head(wine_df)
```

```
## # A tibble: 6 x 12
##   `fixed acidity` `volatile acidity` `citric acid`
##             <dbl>              <dbl>         <dbl>
## 1             7.4               0.7           0
## 2             7.8               0.88          0
## 3             7.8               0.76          0.04
## 4            11.2               0.28          0.56
## 5             7.4               0.7           0
## 6             7.4               0.66          0
## # ... with 9 more variables: residual sugar <dbl>,
## #   chlorides <dbl>, free sulfur dioxide <dbl>,
## #   total sulfur dioxide <dbl>, density <dbl>, pH <dbl>,
## #   sulphates <dbl>, alcohol <dbl>, quality <dbl>
```

4.1.1 What's different about ABCD?

There are a few things that are unique about ABCD datasets that make them a little bit harder to import than normal:

1. They are tab-delimited, rather than comma delimited. This means that when they were being saved, the variables in the file were seperated by a tab (specifically, they were separated by the symbol "\t"), rather than a comma (","). In principle, there is nothing wrong with this. It is just a little unusual and so we need to use a special import function to deal with it (specifically, `read_delim()`.
2. The more complex problem is that ABCD datasets have - in an effort to be helpful - have variable names in the first row of data and variable descriptions in the second row. This confuses R, which is expecting the variable names to be in the first row only, then the data to start in the second row.

4.1.2 Walking through ABCD data importation

We'll do this in a few steps. If you're just looking to copy/paste the code, then skip to the end of this section. If you're looking for information about why we are doing what we are doing, then keep reading.

First, notice that we need to use the `read_delim()` command, rather than the usual `read_csv()`. This tells R we are expecting a delimited file. We also tell that function we are expecting the delimeter to be a tab (`delim = '\t'`), which we could figure out by opening the raw data in a basic text editor, like Notepad, and looking at it.

```
abcd_df <- read_delim('data/abcd_screen02.txt', delim = '\t')

head(abcd_df)
```

```
## # A tibble: 6 x 41
##   collection_id abcd_screen02_id dataset_id subjectkey
##   <chr>         <chr>            <chr>      <chr>
## 1 collection_id abcd_screen02_id dataset_id The NDAR Global~
## 2 2573          6579             47156      NDAR_INV0CZBUV4C
## 3 2573          6581             47156      NDAR_INV0D4C1R8X
## 4 2573          6588             47156      NDAR_INV0DKWEM1A
## 5 2573          6594             47156      NDAR_INV0E350J5D
## 6 2573          6609             47156      NDAR_INV0FM9MUTU
## # ... with 37 more variables: src_subject_id <chr>,
## #   interview_date <chr>, interview_age <chr>, sex <chr>,
## #   eventname <chr>, scrn2_select_language___1 <chr>,
## #   scrn_braces_v2 <chr>, scrn_future_braces <chr>,
## #   scrn_bracesdate_v2 <chr>, scrn_bracescallback_v2 <chr>,
## #   scrn_nr_hair_v2 <chr>, scrn_nr_hair_metal_v2 <chr>,
## #   scrn_nr_hair_remove <chr>, ...
```

We made some progress, but unfortunately, we've got these variable descriptors stuck in the top row of our data now. To get rid of them, we can re-assign the value of `abcd_df` to be a version of itself without it's first row.

```r
abcd_df <- abcd_df %>%
  filter(row_number() != 1)

head(abcd_df)
```

```
## # A tibble: 6 x 41
##   collection_id abcd_screen02_id dataset_id subjectkey
##   <chr>         <chr>            <chr>      <chr>
## 1 2573          6579             47156      NDAR_INV0CZBUV4C
## 2 2573          6581             47156      NDAR_INV0D4C1R8X
## 3 2573          6588             47156      NDAR_INV0DKWEM1A
## 4 2573          6594             47156      NDAR_INV0E350J5D
## 5 2573          6609             47156      NDAR_INV0FM9MUTU
## 6 2573          6630             47156      NDAR_INV0HFHMCFZ
## # ... with 37 more variables: src_subject_id <chr>,
## #   interview_date <chr>, interview_age <chr>, sex <chr>,
## #   eventname <chr>, scrn2_select_language___1 <chr>,
## #   scrn_braces_v2 <chr>, scrn_future_braces <chr>,
## #   scrn_bracesdate_v2 <chr>, scrn_bracescallback_v2 <chr>,
## #   scrn_nr_hair_v2 <chr>, scrn_nr_hair_metal_v2 <chr>,
## #   scrn_nr_hair_remove <chr>, ...
```

That helped, but R still thinks that all of our variables are character strings. We need to tell R that some of our variables might be numbers and that we want it to guess which ones those are. We can do that with the very handy `type_convert()` function.

This function isn't perfectly accurate at guessing what your underlying data types are, but after testing it on several ABCD datasets, it has yet to make a mistake. So, although the safest practice is technically to double-check every column in your dataset, it is should be safe to assume `type_convert()` is almost always right.

```
abcd_df <- type_convert(abcd_df)

head(abcd_df)
```

```
## # A tibble: 6 x 41
##   collection_id abcd_screen02_id dataset_id subjectkey
##           <dbl>            <dbl>      <dbl> <chr>
## 1          2573             6579      47156 NDAR_INV0CZBUV4C
## 2          2573             6581      47156 NDAR_INV0D4C1R8X
## 3          2573             6588      47156 NDAR_INV0DKWEM1A
## 4          2573             6594      47156 NDAR_INV0E350J5D
## 5          2573             6609      47156 NDAR_INV0FM9MUTU
## 6          2573             6630      47156 NDAR_INV0HFHMCFZ
## # ... with 37 more variables: src_subject_id <chr>,
## #   interview_date <chr>, interview_age <dbl>, sex <chr>,
## #   eventname <chr>, scrn2_select_language___1 <dbl>,
## #   scrn_braces_v2 <dbl>, scrn_future_braces <dbl>,
## #   scrn_bracesdate_v2 <chr>, scrn_bracescallback_v2 <dbl>,
## #   scrn_nr_hair_v2 <dbl>, scrn_nr_hair_metal_v2 <dbl>,
## #   scrn_nr_hair_remove <dbl>, ...
```

4.1.3 Copy/paste-able code

Putting it all together, we can import an ABCD dataset like so.

```
abcd_df <- read_delim('data/abcd_screen02.txt', delim = '\t') %>%
  filter(row_number() != 1) %>%
  type_convert()
```

4.2 A general-purpose ABCD dataset import function

If that's easy for your to remember, then feel free to type those three lines every time. On the other hand, if you have **multiple datasets** you need to import, it is safer to make a **function** that can repeat the process for you several times in **exactly the same way each time**.

```
read_abcd <- function(file_path){
  read_delim(file_path, delim = '\t') %>%
    filter(row_number() != 1) %>%
    type_convert()
}
```

With this function, we can now load three different datasets according to exactly the same rules each time.

```
df1 <- read_abcd('data/abcd_lpds01.txt')
df2 <- read_abcd('data/abcd_lpmh01.txt')
df3 <- read_abcd('data/abcd_lpsaiq01.txt')
```

4.2.1 Getting rid of import messages

Although import messages are useful for understanding whether your data have made it into R, they are much less helpful (even overwhelming) when you are importing several datasets at once. To handle this, we can modify our `read_abcd()` function to suppress import messages.

```
read_abcd_quietly <- function(file_path){
  suppressMessages(
    expr = read_delim(file_path, delim = '\t') %>%
      filter(row_number() != 1) %>%
      type_convert())
}

abcd_df <- read_abcd_quietly('data/abcd_lpds01.txt')
```

4.3 Importing a whole folder of ABCD datasets

One of the great advantages of a large data repository is the ability to incorporate multiple datasets in a single analysis. But this introduces a new problem for how to get all of those datasets into R to perform such an analysis.

4.3.1 As easy strategy that unfortunately won't scale

A first guess that most people use is simply to import them all explicitly, like we did above. This is a great approach for a small number of files, but would not work for importing more than that (e.g., for "high-dimensional" analyses, like machine learning).

```
# This is the same example as above

df1 <- read_abcd('data/abcd_lpds01.txt')
df2 <- read_abcd('data/abcd_lpmh01.txt')
df3 <- read_abcd('data/abcd_lpsaiq01.txt')
```

```
# ...

df100 <- read_abcd('data/abcd_lpksad01.txt')

# Imagine how hard it would be to type out
# 100 dataset names (correctly)!
```

Instead, we can simply tell R that we want to import an entire folder of datasets, which in this case we named the `many_datasets` folder. Note, this trick uses functions from the `purrr` package of the `tidyverse`. These functions are a little trick and we cover them at various points later in this textbook. For now, all you need to know is that they take a list of objects (in this case, filenames) and perform the same process for each one.

The first step is to get a list of files inside your folder of interest. Also, make sure to use `full.names = T` to get the file **path** in addition to each file's name.

```
files_to_import <- list.files(
  path = 'data/many_datasets',
  full.names = T)

files_to_import
```

```
## [1] "data/many_datasets/abcd_lpds01.txt"
## [2] "data/many_datasets/abcd_lpksad01.txt"
## [3] "data/many_datasets/abcd_lpmh01.txt"
## [4] "data/many_datasets/abcd_lpohstbi01.txt"
## [5] "data/many_datasets/abcd_lpsaiq01.txt"
## [6] "data/many_datasets/abcd_medhxss01.txt"
## [7] "data/many_datasets/abcd_screen02.txt"
## [8] "data/many_datasets/abcd_socdev_child_emr01.txt"
```

The second step is to get the names we want to assign to each dataset, once it is imported into R. We use a few tricks here, including the `map_chr()` function from `purrr` and the `str_extract()` function from the `stringr` package. Both of these functions are again covered later in the textbook, but are mentioned here in case you simply want to copy/paste code in a hurry and explore the details later.

```
df_names <- map_chr(
  .x = files_to_import,
  .f = ~ str_extract(.x, 'abcd_[\\w|\\d]*'))

df_names
```

```
## [1] "abcd_lpds01"              "abcd_lpksad01"
## [3] "abcd_lpmh01"              "abcd_lpohstbi01"
## [5] "abcd_lpsaiq01"            "abcd_medhxss01"
## [7] "abcd_screen02"            "abcd_socdev_child_emr01"
```

The final step has two parts we execute simultaneously: importing each file on the list (the `files_to_import` list) and assigning the imported file to an R object (using the `df_names` list we just made).

```
my_datasets <- map(files_to_import, read_abcd_quietly) %>%
  set_names(df_names)
```

With this process completed, we can now import an entire folder of datasets and store them in a single object (`my_datasets`). Whenever we want to reference a specific one, we can just use the `$` operator to access it.

```
head(my_datasets$abcd_lpds01)
```

```
## # A tibble: 6 x 160
##   collection_id abcd_lpds01_id dataset_id subjectkey
##           <dbl>          <dbl>      <dbl> <chr>
## 1          2573          61501      47217 NDAR_INV1FW43D9V
## 2          2573          61535      47217 NDAR_INV1KZTEZF5
## 3          2573          61646      47217 NDAR_INV1LC5DBRK
## 4          2573          64616      47217 NDAR_INV4AYYAKWZ
## 5          2573          64701      47217 NDAR_INV4JZNJZVZ
## 6          2573          64711      47217 NDAR_INV4KKHGHCL
## # ... with 156 more variables: src_subject_id <chr>,
## #   interview_age <dbl>, interview_date <chr>, sex <chr>,
## #   eventname <chr>, demo_l_p_select_language___1 <dbl>,
## #   demo_prim_l <dbl>, demo_brthdat_v2_l <dbl>,
## #   demo_ed_v2_l <dbl>, demo_gender_id_v2_l <dbl>,
## #   demo_nat_lang_l <dbl>, demo_nat_lang_2_l <dbl>,
## #   demo_dual_lang_v2_l <dbl>, ...
```

Or

```
head(my_datasets$abcd_medhxss01)
```

```
## # A tibble: 6 x 34
##   collection_id abcd_medhxss01_id dataset_id subjectkey
##           <dbl>             <dbl>      <dbl> <chr>
## 1          2573             32816      47391 NDAR_INV0EWGP0~
## 2         2573            32821      47391 NDAR_INV0F82C6~
## 3         2573            32831      47391 NDAR_INV0G2N59~
## 4         2573            32837      47391 NDAR_INV0GPKYM~
## 5         2573            32839      47391 NDAR_INV0GVW93~
## 6         2573            32843      47391 NDAR_INV0GZM9U~
## # ... with 30 more variables: src_subject_id <chr>,
## #   interview_date <chr>, interview_age <dbl>, sex <chr>,
## #   eventname <chr>, medhx_ss_4b_p <dbl>,
## #   medhx_ss_5b_p <dbl>, medhx_ss_6a_times_p <dbl>,
## #   medhx_ss_6b_times_p <dbl>, medhx_ss_6c_times_p <dbl>,
## #   medhx_ss_6d_times_p <dbl>, medhx_ss_6e_times_p <dbl>,
## #   medhx_ss_6f_times_p <dbl>, ...
```

Chapter 5

Visualization

Perhaps R's single greatest advantage over other software packages is its ability to produce publication-ready figures quickly. Although there are multiple approaches to doing so, by far the most popular is to use `ggplot2`, which is a sub-package of the `tidyverse`.

5.1 Overview

The "gg" in `ggplot` stands for "Grammar of Graphics" and is a key concept for understanding how `ggplot` works. In short, the designers of this package argue that making any 2D figure (and maybe 3D ones too!) involves a set of rules - a "grammar" - that describes how to go from the data to the picture. In what follows, we'll build up that grammmar and see how we can add "words" (figure elements, like points and color) to a basic plot to make it more complex.

We start by importing a basic cancer dataset. Note that in this case, we are importing a `.sav` file from SPSS, so we need to use the `haven::read_spss()` function, instead of `read_abcd()` or `read_csv()`.

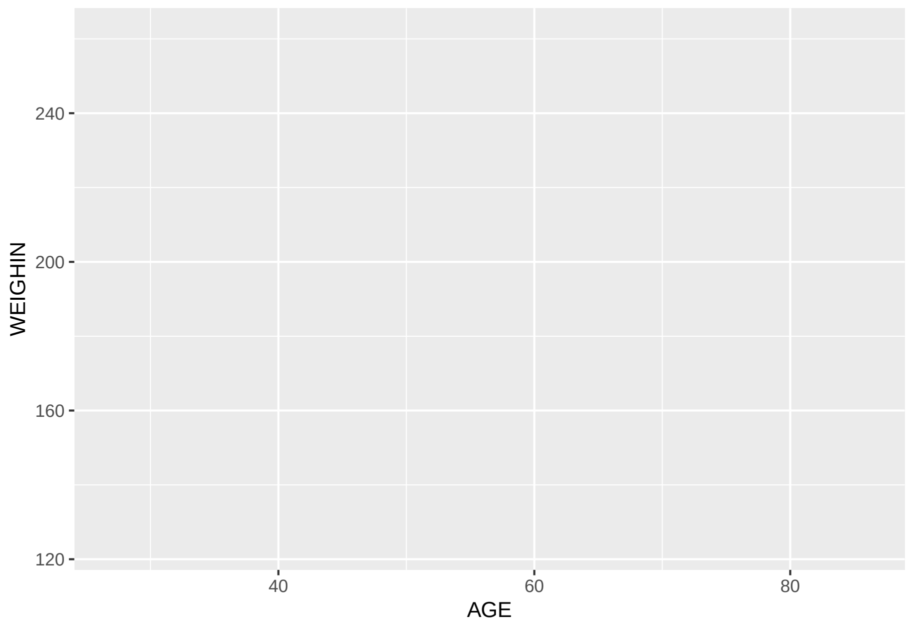
```
library(tidyverse)

my_data <- haven::read_spss('data/cancer.sav')
```

5.2 `ggplot` Mappings

The first step to producing a plot is to call the `ggplot()` function, which takes two arguments: the dataset that you want to plot from and a `mapping`. The first argument is obvious, but what is a mapping? In short, it is a set of instructions for how `ggplot` should turn columns of your dataset into features of your plot. In our case, we just tell it that we want the `AGE` variable to represent our x-axis and `WEIGHIN` to represent our y-axis.

We save the result to an object `p` (for "plot"), which we will add pieces to as we go. To see how our plot looks for far, we just call it on a line all by itself.

```r
p <- ggplot(data = my_data, mapping = aes(x = AGE, y = WEIGHIN))

p
```

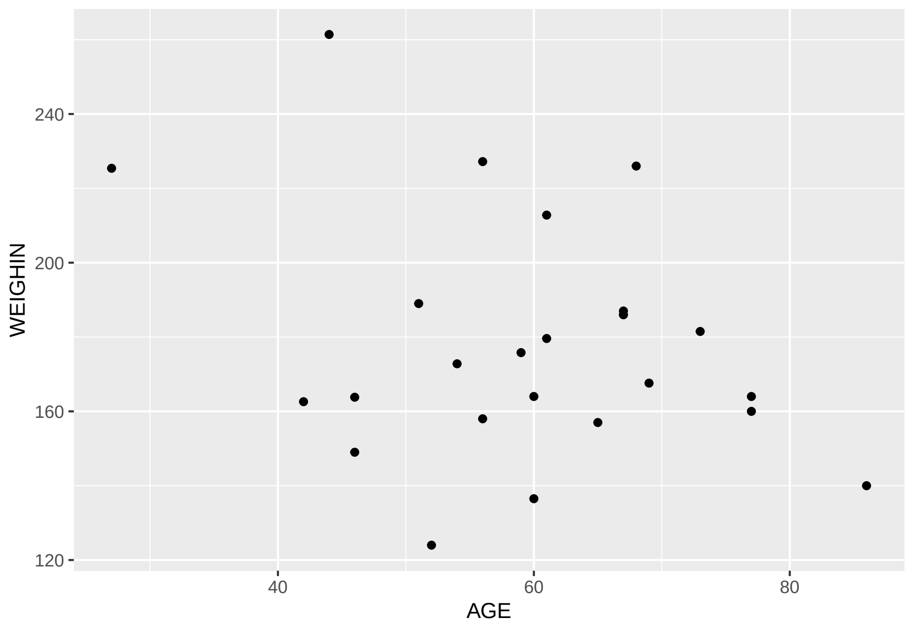


So far, it doesn't look like much, but that's because all we've done is specify our data an our axes. Still, we can see that we've made some progress. The x-axes looks to include some age-like numbers from 20 - 80. Likewise, the y-axis looks like it includes some weight-like numbers, from 120 - 250. We got at least that right.

5.3 Adding geoms to the plot

Now we need to add some `geoms` ("geometric objects", like points) to our plot.

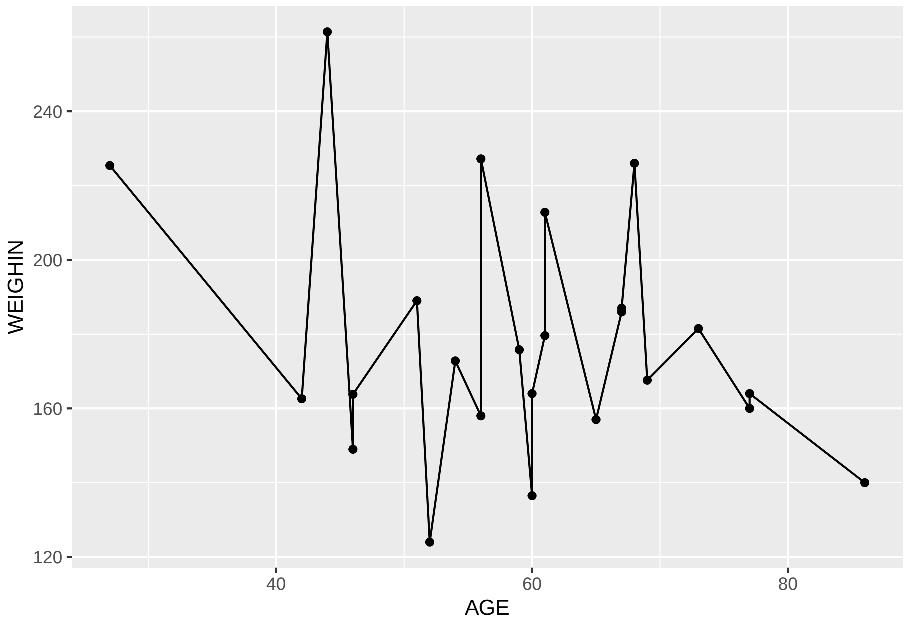
```
p + geom_point()
```



Our plot now has some real content to it. We can easily see the (weakly positive) relationship between age and weight, represented by these points. There are many other geoms you can add, some of which will make more and less sense.

For example, we can connect adjacent points in the dataset with `geom_line()`. This would make a lot of sense for longitudinal data, but makes less sense here.

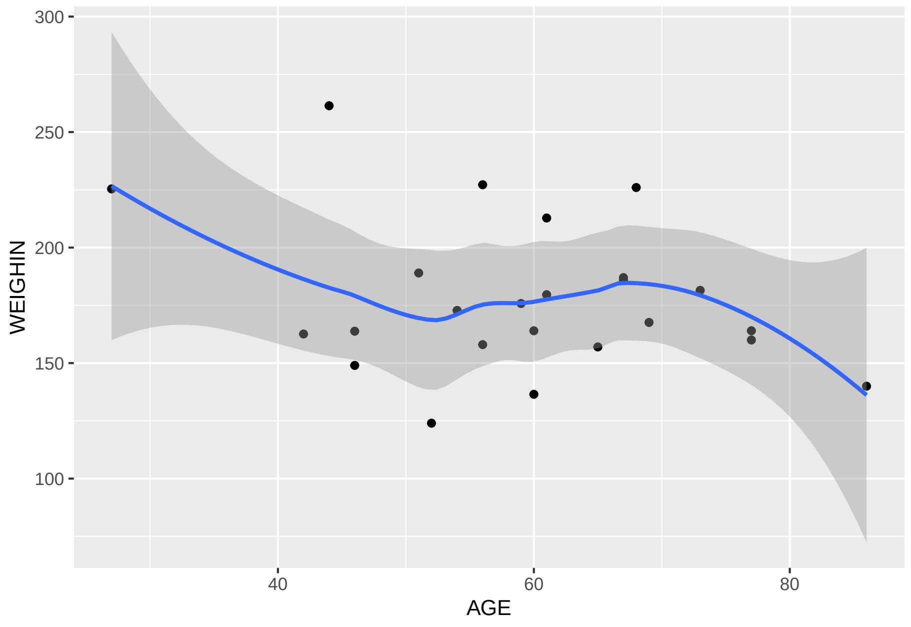
```
p +
  geom_point() +
  geom_line()
```



A more appropriate geom to add might be `geom_smooth()`, which will give us a smoothed line intended to summarize the relationship between our X and Y axes.

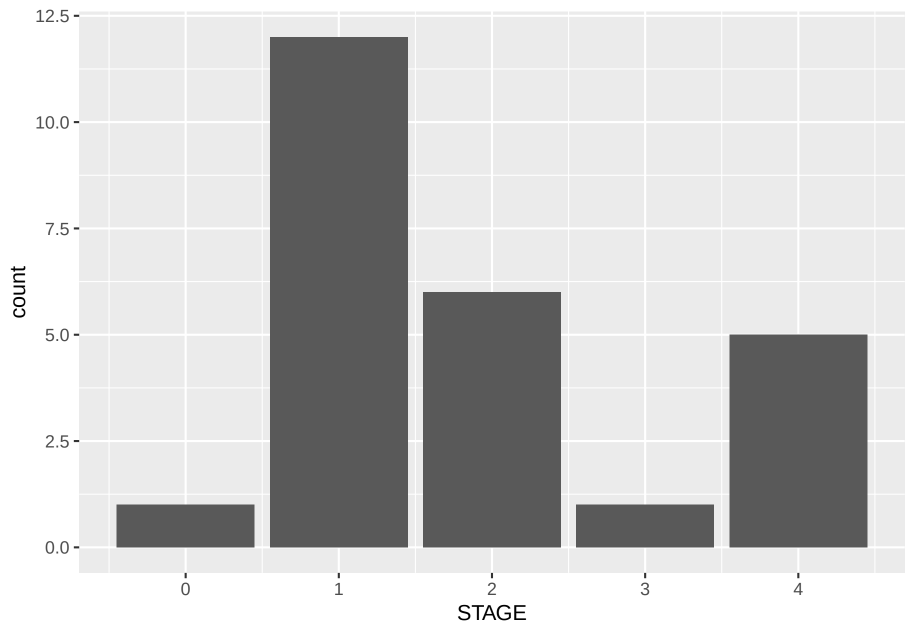
```
p +
  geom_point() +
  geom_smooth()
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```

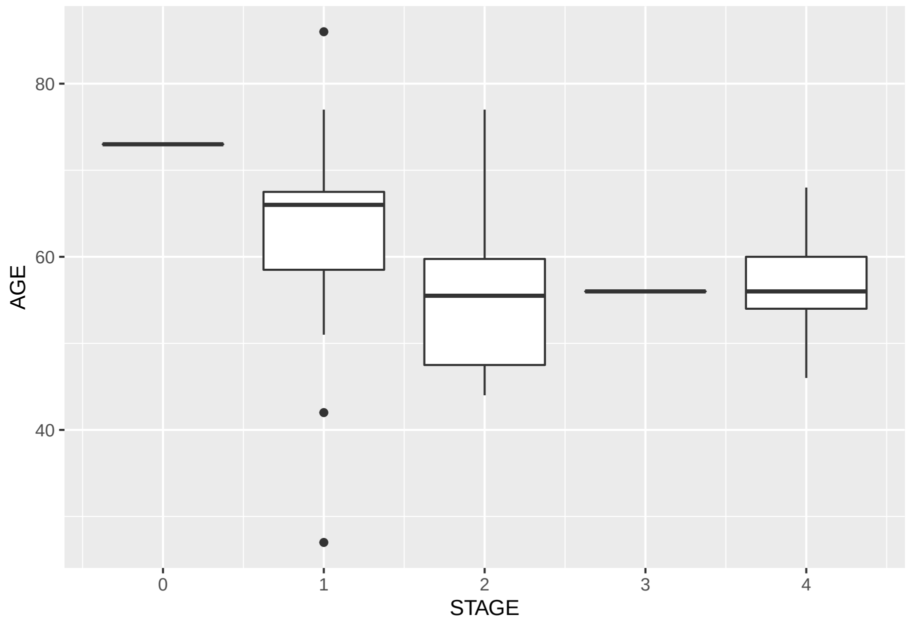


To give some quick examples, here are some other geoms that could be applied even to this relatively small dataset. Note that depending on the geom, which might need to specify our aesthetic mappings differently.

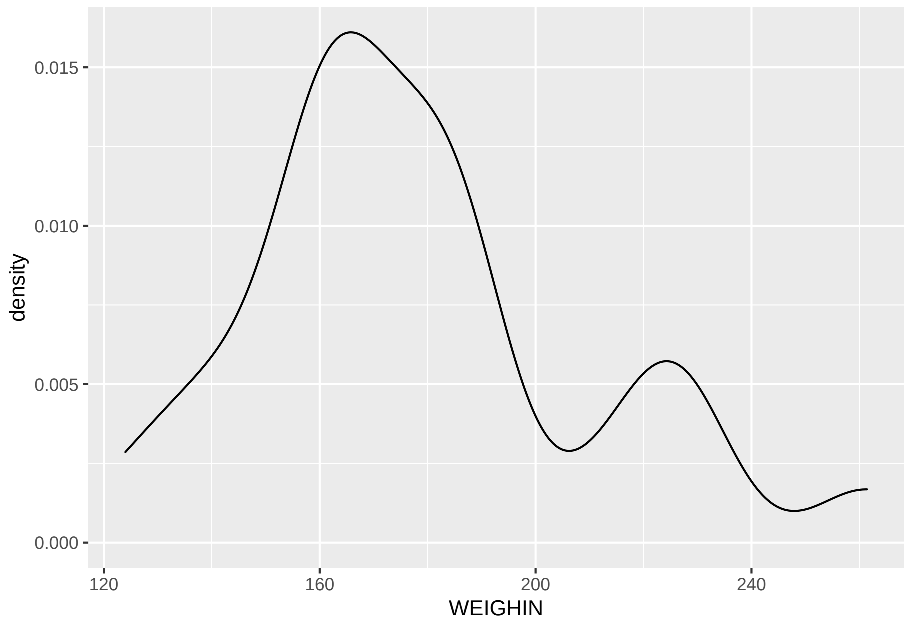
```
ggplot(my_data, aes(x = STAGE)) +
  geom_bar()
```



```r
ggplot(my_data, aes(x = STAGE, y = AGE, group = STAGE)) +
  geom_boxplot()
```



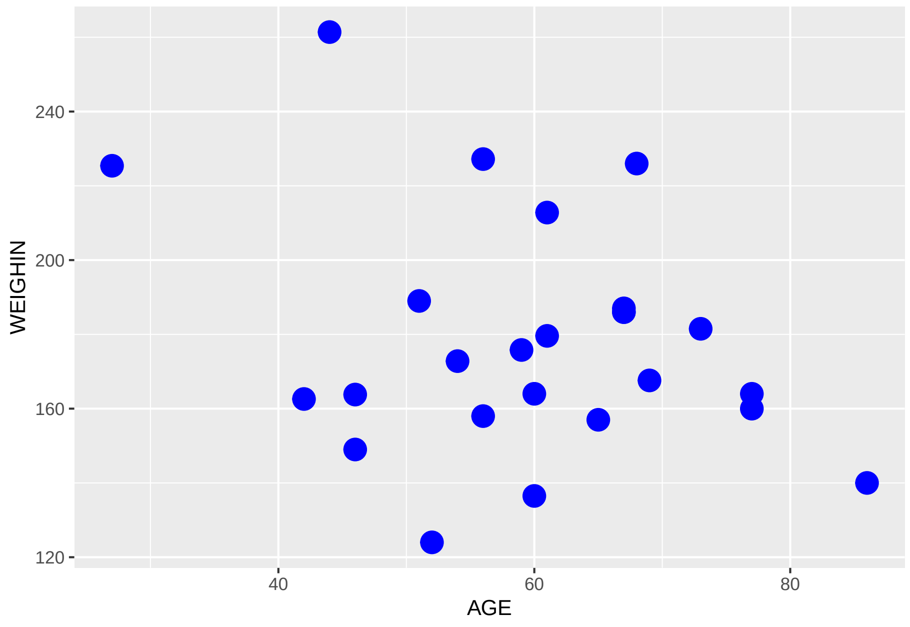
```r
ggplot(my_data, aes(x = WEIGHIN)) +
  geom_density()
```



5.4 Geom Options and Mappings

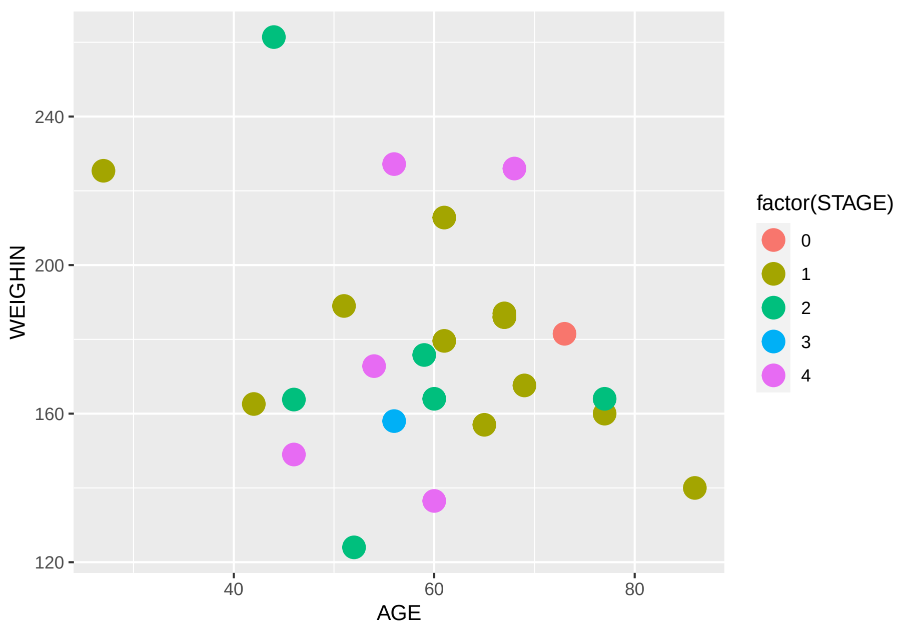
At this point, it is important to mention that all geoms have a variety of options you can apply to them. For example, if I want to increase or decrease my points, I can change the size of them, like so. At the same time, I could also change their color.

```r
p +
  geom_point(size = 5, color = 'blue')
```



But the real power of `ggplot` comes from the fact that you can force your geoms to react to your data by feeding them their own **mapping** (see earlier section for description of what mapppings are). For example, I might want the size of all points to remain large, but have the color of the points change by the stage of cancer the patient is in. To achieve that, I feed the `STAGE` variable to the `mapping` argument of `geom_point()`, like so.

```
p +
  geom_point(
    mapping = aes(color = factor(STAGE)),
    size = 5)
```



Conveniently, whenever you feed a mapping to a geom (i.e., tell `ggplot` to react to a feature of your dataset), it will automatically create a legend for you.

5.5 Saving plots

Once you are satisfied with your plot, you can save it with the convenient `ggsave()` function. This function will guess which kind of image you want to save (e.g., `.png`, `.jpg`), based on the output file name you give it.

```
ggsave(filename = 'my_plot.png', plot = p)
```

```
## Saving 6.5 x 4.5 in image
```

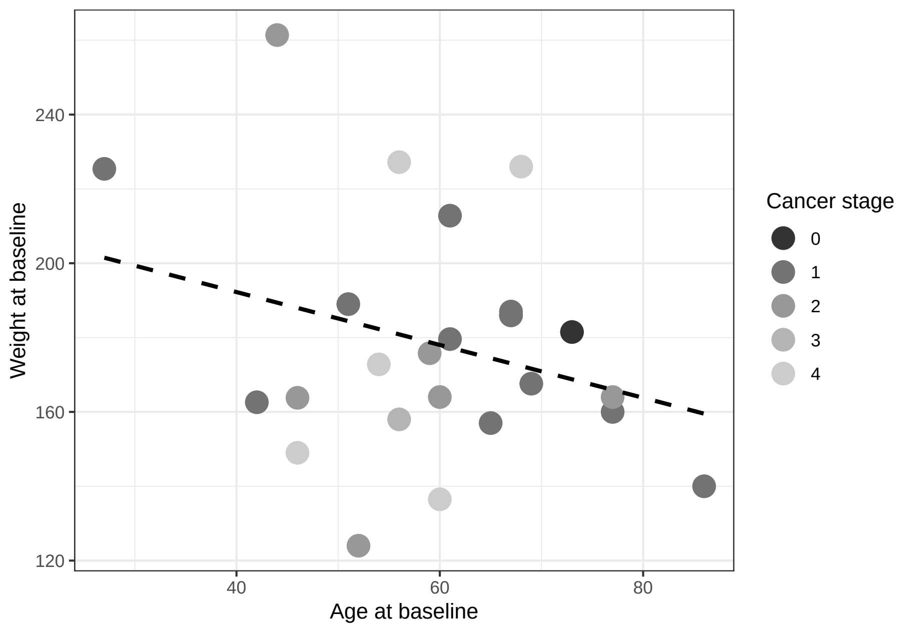
5.6 Other features to look for

The `ggplot2` package is a large collection of functions, that is designed to be as flexible as possible. For this reason, there are too many of them to cover in just one introductory chapter. However, the documentation on ggplot - which you can again reach with the `?` operator - is rich and should help you continually expand you `ggplot` skills. To highlight the kind of options you might want to investigate for creating a publication-ready figure, we leave the example below.

This example shows a few things (e.g., changing themes, adding a line of best fit, colored dots forced to grey scale), including how few lines of code can produce a high quality figure.

```
ggplot(data = my_data, mapping = aes(x = AGE, y = WEIGHIN)) +
  geom_point(
    mapping = aes(color = factor(STAGE)),
    size = 5) +
  geom_smooth(
    method = 'lm',
    se = F,
    color = 'black',
    linetype = 2) +
  scale_color_grey() +
  theme_bw() +
  labs(
    x = 'Age at baseline',
    y = 'Weight at baseline',
    color = 'Cancer stage')
```

```
## `geom_smooth()` using formula 'y ~ x'
```



Chapter 6

Data Wrangling

If rapid publication-ready figures is R's greatest advantage over other software packages, then rapid data manipulation is a close second.

6.1 Base R

As we already covered in an earlier chapter, R's base packages make quick work of performing the same operation for every item in a list of numbers.

```r
my_vec <- c(5, 4, 3, 7)
my_roots <- sqrt(my_vec)
```

To achieve the same thing in Python, you would need a much wordier list comprehension.

```python
my_vec = [5, 4, 3, 7]
my_roots = [sqrt(num) for num in my_vec]
```

6.2 The new way: `dplyr`

Although the base R packages are effective on their own, they suffer from a few drawbacks. For one, they are not always consistent with one another. Moreover, data manipulation almost always involves several steps and the base R approach is not especially useful for making data manipulation pipelines.

To solve these and many other problems, the `dplyr` (pronounced "DEE-ply-er") packages was developed with a consistent set of human-readable functions that always (a) take a dataframe as their first argument and (b) return a dataframe as their output. Together, this allows you to string them together in a convenient human-readable pipeline.

Like `ggplot`, the `dplyr` package is "opinionated." It thinks that data manipulation should almost always be done a certain way. Specifically, if you are working with "rectangular" data - that is, data that can be cleanly expressed in the form of a spreadsheet - then you'll be able to accomplish almost all of your data manipulation with the following "verbs" (functions).

6.3 The only data verbs you'll ever need

For simplicity, we'll again use the cancer dataset from the visualization chapter. We do this because, although an ABCD dataset would be more topical, they are often very large, which would make it hard to eye-ball whether our data manipulation went according to plan.

```
library(tidyverse)

my_data <- haven::read_spss('data/cancer.sav')
```

6.3.1 `rename()`

The cancer dataset is great the way it is, simple and organized. But it has one feature we might want to change right away, which is its naming conventions.

```
names(my_data)
```

```
## [1] "ID"        "TRT"       "AGE"       "WEIGHIN"   "STAGE"
## [6] "TOTALCIN"  "TOTALCW2"  "TOTALCW4"  "TOTALCW6"
```

To rename any one of these variables, simply call the `rename()` function, and tell it which old variable you want to give which new name. Don't forget to save your result into an object too, so that your changes don't disappear into nowhere.

```
my_data <- my_data %>%
  rename(subject_id = ID, condition = TRT)

names(my_data)
```

```
## [1] "subject_id" "condition"  "AGE"        "WEIGHIN"
## [5] "STAGE"      "TOTALCIN"   "TOTALCW2"   "TOTALCW4"
## [9] "TOTALCW6"
```

This is great if we just want to change a few variable names, but sometimes we want to change all of them in a particular way. For that, we can use `rename_with()`, which will apply a function to every variable name. For example, we might want to turn all of the variable names to lowercase. To achieve that, we just feed the `tolower()` function, which turns anything it comes across into lowercase, to the `rename_with()` function. Now all of our variable names are in the same case and we are less likely to make a mistake later.

```
my_data <- my_data %>%
  rename_with(tolower)

names(my_data)
```

```
## [1] "subject_id" "condition"  "age"        "weighin"
## [5] "stage"      "totalcin"   "totalcw2"   "totalcw4"
## [9] "totalcw6"
```

6.3.2 `select()`

This verb allows you to retain only a subset of variables. You can do this by naming them explicitly...

```
my_data %>%
  select(subject_id, condition, age)
```

```
## # A tibble: 25 x 3
##    subject_id condition   age
##         <dbl>     <dbl> <dbl>
##  1          1         0    52
##  2          5         0    77
##  3          6         0    60
##  4          9         0    61
##  5         11         0    59
##  6         15         0    69
##  7         21         0    67
##  8         26         0    56
##  9         31         0    61
## 10         35         0    51
## # ... with 15 more rows
```

… or by using one of `dplyr`'s "select helper" functions, like `starts_with()` and `ends_with()`.

```
my_data %>%
  select(starts_with('total'))
```

```
## # A tibble: 25 x 4
##    totalcin totalcw2 totalcw4 totalcw6
##       <dbl>    <dbl>    <dbl>    <dbl>
##  1        6        6        6        7
##  2        9        6       10        9
##  3        7        9       17       19
##  4        6        7        9        3
##  5        6        7       16       13
##  6        6        6        6       11
##  7        6       11       11       10
##  8        6       11       15       15
##  9        6        9        6        8
## 10        6        4        8        7
## # ... with 15 more rows
```

6.3.3 `mutate()`

Assuming we have selected our major variables of interest, we can now use `mutate()` to change existing columns or make new ones. For example, if we wanted to compute the average of `totalcw2`, `totalcw4`, and `totalcw6`, we could do it like this.

```
my_data <- my_data %>%
  mutate(meanc = (totalcw2 + totalcw4 + totalcw6)/3)

my_data %>%
  select(subject_id, meanc) %>%
  head()
```

```
## # A tibble: 6 x 2
##   subject_id meanc
##        <dbl> <dbl>
## 1          1  6.33
## 2          5  8.33
## 3          6 15
## 4          9  6.33
## 5         11 12
## 6         15  7.67
```

6.3.4 `group_by()` and `summarize()`

These two verbs are technically distinct, but are almost always used in combination. To explain, `group_by()` tells R to do whatever comes NEXT separately for each group. In turn, `summarize()` is like `mutate()` in that it makes new columns; however, it makes new columns by **aggregating** information across rows in a given group. Thus, when using `summarize()` to make a new column, you will also end up with just one row per group, like so.

```
my_data %>%
  group_by(condition, stage) %>%
  summarise(ave_weight = mean(weighin))
```

```
## `summarise()` has grouped output by 'condition'. You can
## override using the `.groups` argument.
```

```
## # A tibble: 8 x 3
## # Groups:   condition [2]
##   condition stage ave_weight
##       <dbl> <dbl>      <dbl>
## 1         0     1       180.
## 2         0     2       155.
## 3         0     3       158
## 4         0     4       143.
## 5         1     0       182.
## 6         1     1       179.
## 7         1     2       196.
## 8         1     4       209.
```

As you can see, we can group by multiple variables at the same time and quickly get the kind of information we would need for a demographics table - in this case, the average weight of subjects by both condition and cancer stage.

Chapter 7

Basic hypthothesis tests

So far, we have used R to manipulate our data and to provide some summary statistics, including visualizations of those statistics. In this section, we'll start conducting our first statistical inference tests.

7.1 Not your grandfather's statistical tests

Experience has shown that it is worthwhile to take a moment to describe the R statistical inference process to users who come from other platforms (e.g., SPSS, SAS). In those other platforms, the statistical inference process is all about getting test results printed to the screen. In practice, that process looks like (a) importing some data, (b) cleaning it, then (c) applying some test and receiving output on the screen. If your only goal is to produce statistical output, this is a totally natural workflow and it makes sense those platforms use it.

In contrast, R has broader goals. It is a full-service programming language capable of interacting with your operating system, scraping web data from servers across the world, and even developing web apps. Because of this, the data analysis process is focused on producing a **named object** that represents the results of an analysis, which can then be incorporated into a broader coding pipeline.

If that sounds complicated, don't worry, it just involves one more step than what you are used to. In R, you analysis workflow looks like this:

1. Import data
2. Clean data
3. Use a pre-existing function to produce an analysis object for your (e.g., run a statistical test and save everything about the results)
4. Extract a summary of the results of the analysis (the one new step).

7.2 Steps 1 & 2 - Load and clean data

In this case, we'll use an ABCD dataset containing KSADS diagnostic information for a large number of patients. This dataset includes a number of interesting variables, including whether the child has been bullied, what their average grades are in school, and how many times they've been hospitalized. We start by copy/pasting our custom `read_abcd_quietly()` function from the **Working with ABCD** section of this textbook, then use it to load the dataset.

```
library(tidyverse)

read_abcd_quietly <- function(file_path){
  suppressMessages(
    expr = read_delim(file_path, delim = '\t') %>%
      filter(row_number() != 1) %>%
      type_convert())
}

df <- read_abcd_quietly('data/abcd_lpksad01.txt')
```

During the cleaning process, we'll rename our key variables to make things a little easier to follow. Additionally, this is a longitudinal dataset, so each patient appears in it multiple times. We'll code which timepoint a patient belongs to, based on the dates of their visits. In this case though, all of our analyses will be cross-sectional. So once we have computed our `time` variable, we'll simply filter the first one for each patient, resulting in a cross-sectional (baseline) version of the dataset for analysis.

```
df <- df %>%
  arrange(src_subject_id) %>%
  select(
    id = src_subject_id,
    age = interview_age,
    is_bullied = kbi_p_c_bully_l,
    num_hospitalizations = kbi_ss_c_mental_health_p_l,
    grades = kbi_p_grades_in_school_l) %>%
  group_by(id) %>%
  mutate(n_timepoints = n()) %>%
  ungroup() %>%
  filter(n_timepoints == 3) %>%
  mutate(
    grades = ifelse(
      test = grades == 6 | grades == -1,
      yes = NA,
      no = grades),
    too_kool_4_skool = case_when(
      grades == 1 ~ 'nerds',
      grades == 2 | grades == 3 ~ 'besties',
      grades > 3 ~ 'teen movie cool kids')) %>%
  group_by(id) %>%
  arrange(age) %>%
  mutate(time = row_number()) %>%
  filter(time <= 2) %>%
  ungroup()
```

As a bit of practice with factor variables, we'll also convert the `is_bullied` variable to a factor. This will make it easier to use the upcoming tests.

```
df <- df %>%
  mutate(
    is_bullied = ifelse(
        test = is_bullied %in% c(1, 2),
        yes = is_bullied,
        no = NA) %>%
      factor(levels = c(1, 2), labels = c('yes', 'no')))

df
```

```
## # A tibble: 12,070 x 8
##    id                  age is_bullied num_hospitaliza~ grades
##    <chr>             <dbl> <fct>                 <dbl>  <dbl>
##  1 NDAR_INV06DE9Y0L    117 no                        0      1
##  2 NDAR_INV13BCLD41    117 no                        0      2
##  3 NDAR_INV88HZ7ZCH    117 no                        0      1
##  4 NDAR_INVA0NWYU17    117 no                        0      1
##  5 NDAR_INVBZZ8KWTC    117 no                        0      2
##  6 NDAR_INVCYVYJKMV    117 no                        0      2
##  7 NDAR_INVFBEG8E1Z    117 no                        0     NA
##  8 NDAR_INVH2NBUFF1    117 no                        0      2
##  9 NDAR_INVH8J67ZNJ    117 yes                       0      2
## 10 NDAR_INVHD5LEW4G    117 no                        0      1
## # ... with 12,060 more rows, and 3 more variables:
## #   n_timepoints <int>, too_kool_4_skool <chr>, time <int>
```

7.3 Steps 3 & 4 - Create and unpack analysis objects

7.3.1 Independant samples t-test

To see how the analysis process works in R, we'll start with the familiar independent samples t-test. Like all future analyses we'll conduct, we'll use a pre-existing function to do the work for us. These functions (generally!) take two arguments:

- the dataset you want to analyze
- a formula describing the variables we want to use in our analysis. This is almost always the first argument the function takes and almost always follows the form `DEPENDANT VARIABLE ~ INDEPENDANT_VARIABLES`.

In this case, let's ask whether a patient's average grades differ by whether they were bullied. To do that, we call the `t.test` function on our data and we save the results to an object called `fit`. You can name the object anything you want, but it is customary to name it `fit`, after "fitted model."

```r
fit <- t.test(grades ~ is_bullied, data = df)
```

If you're like most new R users, this is where you might get confused. Where are the results?

Don't worry, nothing has gone wrong. The results are saved in the `fit` object now. All we need to do is get them out.

For simple tests that don't involve a lot of complex mathematical tricks (e.g., SEM), you can often just call the fitted object by itself, like so.

```r
fit
```

```
##
##  Welch Two Sample t-test
##
## data:  grades by is_bullied
## t = 13.131, df = 2243.2, p-value < 2.2e-16
## alternative hypothesis: true difference in means between group yes and group no is 
## 95 percent confidence interval:
##  0.2487967 0.3361517
## sample estimates:
## mean in group yes  mean in group no
##          1.872532          1.580058
```

This is fine for a t-test because there is not much to them. Here, we have all the information we need, including a p-value.

For more complex analyses though (e.g., even regression), you'll want to use a helper function to summarize your results. R has the built in `summary()` function, which will work for almost all analyses, but unfortunately not t-tests.

```
summary(fit)
```

```
##             Length Class  Mode
## statistic   1      -none- numeric
## parameter   1      -none- numeric
## p.value     1      -none- numeric
## conf.int    2      -none- numeric
## estimate    2      -none- numeric
## null.value  1      -none- numeric
## stderr      1      -none- numeric
## alternative 1      -none- character
## method      1      -none- character
## data.name   1      -none- character
```

Instead, we'll use the `tidy()` function from the `broom` package. This gives us a one-line dataframe with the results of the t-test.

```
broom::tidy(fit)
```

```
## # A tibble: 1 x 10
##   estimate estimate1 estimate2 statistic  p.value parameter
##      <dbl>     <dbl>     <dbl>     <dbl>    <dbl>     <dbl>
## 1    0.292      1.87      1.58      13.1 5.28e-38     2243.
## # ... with 4 more variables: conf.low <dbl>,
## #   conf.high <dbl>, method <chr>, alternative <chr>
```

This is great for generating tables and modifying the output for publication.

```
fit %>%
  broom::tidy() %>%
  select(t = statistic, p = p.value) %>%
  round(3)
```

```
## # A tibble: 1 x 2
##       t     p
##   <dbl> <dbl>
## 1  13.1     0
```

7.3.2 Paired t-tests

Paired t-tests can be conducted using a similar process. To demonstrate a longitudinal result though, we'll first filter (retain) only patients who had non-missing grades for at least two timepoints.

Once that's done, all we need to do to make the test a paired one is to set the `paired` argument to true.

```
complete_data_df <- df %>%
  group_by(id) %>%
  filter(all(!is.na(grades)) & n() == 2)

fit <- t.test(grades ~ time, data = complete_data_df, paired = T)

fit
```

```
##
##  Paired t-test
##
## data:  grades by time
## t = 1.0065, df = 5459, p-value = 0.3142
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.01406087  0.04373120
## sample estimates:
## mean of the differences
##              0.01483516
```

Again, we can also call `broom::tidy()` to get the results as a dataframe.

```
broom::tidy(fit)
```

```
## # A tibble: 1 x 8
##   estimate statistic p.value parameter conf.low conf.high
##      <dbl>     <dbl>   <dbl>     <dbl>    <dbl>     <dbl>
## 1   0.0148      1.01   0.314      5459  -0.0141    0.0437
## # ... with 2 more variables: method <chr>,
## #   alternative <chr>
```

7.3.3 Correlation

Correlation is computed using the `cor()` function. It is similar to t-tests in R, but asks that you explicitly tell it which columns you want. It also requires that all of its variables are numeric, so we'll coerce our `is_bullied` variable to a numeric one to make sure `cor()` plays nice.

Note that with `cor()`, we can easily specify how we want to handle missing data and which kind of correlation we want. Remember, to see all of the special tricks you can do with a function, place a `?` in front of it and then run that line.

```
fit <- cor(
  x = as.numeric(df$is_bullied),
  y = df$grades,
  use = 'complete.obs',
  method = 'spearman')

fit
```

```
## [1] -0.1271988
```

At this point, you've probably noticed you're missing the p-value you probably wanted. For a variety of unimportant (but admittedly irritating) historical reasons, we use a different function for that: `cor.test()`. Luckily though, it works the same way as all of the others.

```
fit <- cor.test(
  x = as.numeric(df$is_bullied),
  y = df$grades,
  method = 'pearson')

fit
```

```
##
##  Pearson's product-moment correlation
##
## data:  as.numeric(df$is_bullied) and df$grades
## t = -14.926, df = 11389, p-value < 2.2e-16
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  -0.1564845 -0.1204603
## sample estimates:
##        cor
## -0.1385182
```

```
broom::tidy(fit)
```

```
## # A tibble: 1 x 8
##   estimate statistic  p.value parameter conf.low conf.high
##      <dbl>     <dbl>    <dbl>     <int>    <dbl>     <dbl>
## 1   -0.139     -14.9 6.56e-50     11389   -0.156    -0.120
## # ... with 2 more variables: method <chr>,
## #   alternative <chr>
```

Chapter 8

Linear Regression

Linear regression is the bread and butter of statistical analysis. Unsurprisingly then, it is also the bread and butter of R's statistical analysis framework. That is, most of the R ecosystem uses linear regression as a kind of template. So, even things that are not regressions are specified like them (see the last section, where we use a regression-like formula to specify the variable in our t-tests).

For that reason, getting comfortable with linear regression will tend to automatically make you comfortable with a lot of other tasks in R too! To see this, try to compare this section to the next one on logistic regression. Even though those models are fit with totally different distributions and assumptions, to you the user, they will look and feel basically the same.

Note, the steps to conducting a linear regression are the same as for the basic hypothesis tests in the last chapter.

8.1 Steps 1 & 2 - Load and clean the data

To maximize comparability with the last chapter, we'll stick to the same dataset and cleaning procedure. To make it so you don't have to flip back and forth between those chapters, though, we'll re-describe the process here…

In this case, we'll use an ABCD dataset containing KSADS diagnostic information for a large number of patients. This dataset includes a number of interesting variables, including whether the child has been bullied, what their average grades are in school, and how many times they've been hospitalized. We start by copy/pasting our custom `read_abcd_quietly()` function from the **Working with ABCD** section of this textbook, then use it to load the dataset.

```
library(tidyverse)

read_abcd_quietly <- function(file_path){
  suppressMessages(
    expr = read_delim(file_path, delim = '\t') %>%
      filter(row_number() != 1) %>%
      type_convert())
}

df <- read_abcd_quietly('data/abcd_lpksad01.txt')
```

During the cleaning process, we'll rename our key variables to make things a little easier to follow. Additionally, this is a longitudinal dataset, so each patient appears in it multiple times. We'll code which timepoint a patient belongs to, based on the dates of their visits. In this case though, all of our analyses will be cross-sectional. So once we have computed our `time` variable, we'll simply filter the first one for each patient, resulting in a cross-sectional (baseline) version of the dataset for analysis.

```
df <- df %>%
  arrange(src_subject_id) %>%
  select(
    id = src_subject_id,
    sex,
    age = interview_age,
    is_bullied = kbi_p_c_bully_l,
    num_hospitalizations = kbi_ss_c_mental_health_p_l,
    grades = kbi_p_grades_in_school_l) %>%
  group_by(id) %>%
  mutate(n_timepoints = n()) %>%
  ungroup() %>%
  filter(n_timepoints == 3) %>%
  mutate(
    grades = ifelse(
      test = grades == 6 | grades == -1,
      yes = NA,
      no = grades),
    too_kool_4_skool = case_when(
      grades == 1 ~ 'nerds',
      grades == 2 | grades == 3 ~ 'besties',
      grades > 3 ~ 'teen movie cool kids')) %>%
  group_by(id) %>%
  arrange(age) %>%
  mutate(time = row_number()) %>%
  filter(time <= 2) %>%
  ungroup()
```

Also, to simplify our upcoming analyses, we'll round all the ages to the nearest year (from below, with the `floor()` function) and convert the sex of the patient to a factor variable. This will make out output clearer.

```
df <- df %>%
  mutate(
    age = floor(age/12),
    sex = factor(sex, levels = c('M', 'F'), ordered = F))
```

8.2 Steps 3 & 4 - Fit the model and summarize it

Like with basic hypothsis testing in the last section, we first fit a model and save its attributes (results) to an object, usually named `fit`. The linear regression function is `lm()` for "linear model" and it takes two main arguments:

- The formula for the regression, specified like this - `Outcome ~ Predictor1 + Predictor2 + ....`
- The data you want to analyze

Importantly, R makes it easy to specify interaction terms too. You just need to do the multiplication of your interaction variables in the formula argument. R will handle everything under the hood for you, including dummy coding and the inclusion of lower order terms (i.e., the formula `y ~ x*z` will produce `y ~ x + z + x*z` for you automatically).

```
fit <- lm(
  formula = grades ~ age*is_bullied + sex,
  data = df)

summary(fit)
```

```
##
## Call:
## lm(formula = grades ~ age * is_bullied + sex, data = df)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -0.7295 -0.5309 -0.5192  0.4691  3.4808
```

```
##
## Coefficients:
##                  Estimate Std. Error t value Pr(>|t|)
## (Intercept)     1.7744384  0.1011702  17.539   <2e-16 ***
## age            -0.0050832  0.0090995  -0.559    0.576
## is_bullied      0.0039272  0.0135917   0.289    0.773
## sexF           -0.1928057  0.0142466 -13.533   <2e-16 ***
## age:is_bullied -0.0003888  0.0011983  -0.324    0.746
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.7594 on 11389 degrees of freedom
##   (676 observations deleted due to missingness)
## Multiple R-squared:  0.01596,    Adjusted R-squared:  0.01562
## F-statistic: 46.19 on 4 and 11389 DF,  p-value: < 2.2e-16
```

Here we can see the results of our regression on full display, but perhaps with some labels that are new to you. That's okay, they correspond to all of the traditional regression table objects: estimate is the slope of each term on the left, the standard error comes next, then the test statistic (slope/standard error), and its associated p-value. Note, that when numbers get really small, R will return them in scientific notation. So, the number `<2e-16` means really, really, really small.

Lastly, below the main table, you can see the R-sqauared value, the model F-statistic, its degrees of freedom, and their associated p-value.

8.2.1 Broom can help export the results

If you're like many people, you're running a regression with the intention of publishing the results. This summary is nice, but it is hard to get into Excel or other spreadsheet programs to build a publication-ready table.

To solve that, we can again use the `broom::tidy` method to get a dataframe-based depiction of our results. On it's own, that might not seem important, but it makes the results much easier to export to a csv.

```r
my_results <- broom::tidy(fit)

write_csv(my_results, 'my_results.csv')
```

Chapter 9

Logistic Regression

In this chapter, we'll see the advantage of R's linear regression-centric statistical analysis style (i.e., in which basically everything works the same way as a linear regression). Specifically, we'll switch to a different form of statistical model, with different assumptions and observe that almost everything is the same.

This has the advantage that we need to relearn very little from model to model, and that we can feel more confident that we got our new model (in this case a logistic one) right on the first try.

9.1 Steps 1 & 2 - Import and clean data

To maximize comparability with the regression and hypotehsis testing chapters, we'll again use the KSADS ABCD dataset. We can again import that dataset with our custom functions from earlier chapters.

```r
library(tidyverse)

read_abcd_quietly <- function(file_path){
  suppressMessages(
    expr = read_delim(file_path, delim = '\t') %>%
      filter(row_number() != 1) %>%
      type_convert())
}

df <- read_abcd_quietly('data/abcd_lpksad01.txt')
```

With the dataset loaded, we can now engage in some basic cleanup - the same as in previous chapters.

```
df <- df %>%
  arrange(src_subject_id) %>%
  select(
    id = src_subject_id,
    age = interview_age,
    sex,
    num_hosp = kbi_ss_c_mental_health_p_l,
    grades = kbi_p_grades_in_school_l) %>%
  group_by(id) %>%
  arrange(age) %>%
  mutate(
    age = floor(age / 12),
    sex = factor(sex, levels = c('M', 'F')),
    time = row_number(),
    n_timepoints = max(time)) %>%
  ungroup() %>%
  filter(time == 1) %>%
  filter(grades %in% 1:5) %>%
  select(id, age, sex, grades, num_hosp)

df
```

```
## # A tibble: 9,727 x 5
##    id                  age sex   grades num_hosp
##    <chr>             <dbl> <fct>  <dbl>    <dbl>
##  1 NDAR_INV9XU9GFCB      9 M          1        0
##  2 NDAR_INV029PWCFY      9 M          1        0
##  3 NDAR_INV06DE9Y0L      9 M          1        0
##  4 NDAR_INV0BL9EL2Y      9 F          1        0
##  5 NDAR_INV13BCLD41      9 M          2        0
##  6 NDAR_INV1APPZYY8      9 F          1        0
##  7 NDAR_INV2B9KMD5C      9 F          1        0
##  8 NDAR_INV2RYEWWRN      9 M          1        0
##  9 NDAR_INV3NT6ML17      9 F          1        0
## 10 NDAR_INV5FKNM21M      9 F          1        0
## # ... with 9,717 more rows
```

One difference, though, is that we'll need a dichotomous variable to analyze in our logistic regression. Here, we turn our `num_hosp` variable (i.e., the number of times a given patient has been hospitalized since last interview) into a dichotomous one, `ever_hosp` (i.e., has the patient been hospitalized at all since the past interview?).

```r
df <- df %>%
  mutate(ever_hosp = num_hosp > 0)
```

9.2 Steps 3 & 4 - Fit the model and summarize it

To fit a logistic regression, we follow roughly the same steps as a linear one, with just a few changes.

First, we need to use `glm()` instead of the regular `lm()`. This is because we are fitting a "generalized linear model" (the family logistic regression belong to) instead of a traditional "general linear model."

The second change is that we need to tell R which kind of GLM we want to fit (there are many). We do this by telling it what we think the outcome distribution is like (e.g., normal, binomial, etc). In this case, the family of distributions that logistic regressions use is the "binomial" one, so to get a logistic regression we simply feed the argument `'binomial'` to the `family` parameter.

```r
fit <- glm(
  formula = ever_hosp ~ sex + age,
  data = df,
  family = 'binomial')
```

Once the model is fit, we can ask for a summary the same way that we would with a linear regression and we'll see that the table we get is essentially the same.

```r
summary(fit)
```

```
##
## Call:
## glm(formula = ever_hosp ~ sex + age, family = "binomial", data = df)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -0.1402  -0.0871  -0.0686  -0.0540   3.7269
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) -10.8298     3.3850  -3.199  0.00138 **
## sexF         -0.8932     0.4792  -1.864  0.06234 .
## age           0.4779     0.3146   1.519  0.12874
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 311.95  on 9719  degrees of freedom
## Residual deviance: 305.73  on 9717  degrees of freedom
##   (7 observations deleted due to missingness)
## AIC: 311.73
##
## Number of Fisher Scoring iterations: 9
```

9.2.1 Where are my odds ratios?

Most research scientists are used to thinking about logistic regression results in terms of odds ratios. Unfortunately, R won't give you odds ratios by default. You need to compute them. Fortunately, it is really easy, once you remember that odds ratios are equal to:

$$OR = e^b = \exp(b)$$

Thus, the quickest way to get them is to extract your coefficients from your fit object, then use `exp()` on them.

```
my_odds_ratios <- fit %>%
  coef() %>%
  exp() %>%
  round(digits = 2)

my_odds_ratios
```

```
## (Intercept)        sexF         age
##        0.00        0.41        1.61
```

If you are in the process of creating a regression table, though, you might prefer to use `broom::tidy()` to get a dataframe of your `fit` object, then modify that with `mutate()`. This will make your results easier to export in .csv format. If all you want are the numbers,

```
results <- fit %>%
  broom::tidy() %>%
  mutate(or = exp(estimate)) %>%
  rename(b = estimate) %>%
  mutate(across(where(is.numeric), .fns = ~ round(.x, 3)))

results
```

```
## # A tibble: 3 x 6
##   term              b std.error statistic p.value    or
##   <chr>         <dbl>     <dbl>     <dbl>   <dbl> <dbl>
## 1 (Intercept) -10.8       3.38      -3.20   0.001 0
## 2 sexF         -0.893     0.479     -1.86   0.062 0.409
## 3 age           0.478     0.315      1.52   0.129 1.61
```

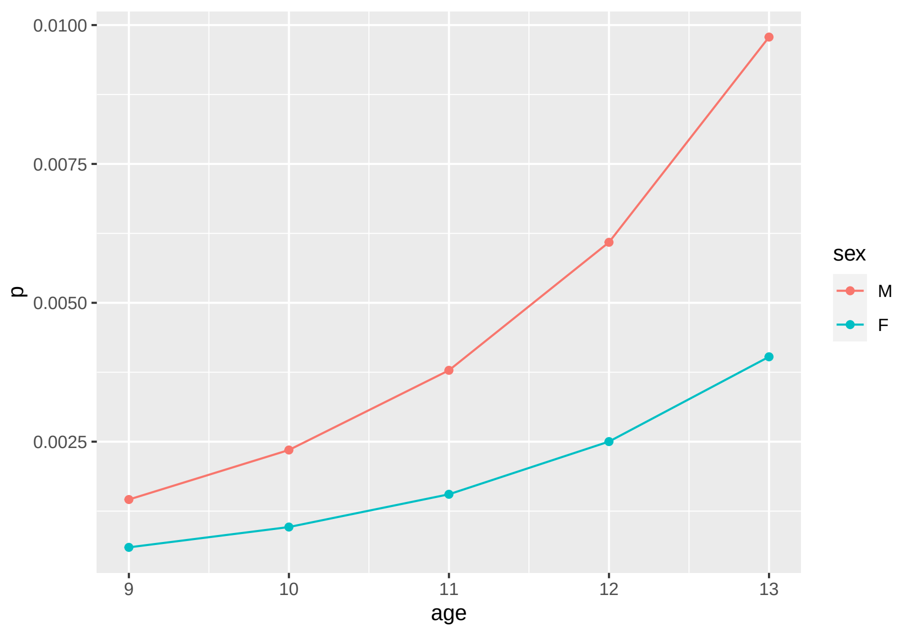
```
write_csv(results, 'logistic_results.csv')
```

9.3 Plotting predicted values

One question that has come up in the past is how to produce a plot of predicted values for a regression, including logistic ones.

This is possible by mixing tricks from `broom` and `ggplot` together, like so.

```
predicted_vals <- fit %>%
  broom::augment() %>%
  select(sex, age, log_odds = .fitted) %>%
  mutate(odds = exp(log_odds), p = odds/(1 + odds)) %>%
  unique()

ggplot(predicted_vals, aes(x = age, y = p, color = sex)) +
  geom_point() +
  geom_line()
```


0.0100
0.0075
0.0050
0.0025
p
9
10
11
12
13
age
sex
M
F

Chapter 10

Introduction to machine learning

For many years, the field of statistics has proceeded along two different lines of development: one focused on **inference** and the other focused on **prediction**. Most readers of this particular document will have been trained in the **inference tradition**, in which the goal of a statistical analysis is to use sample information to make an inference about a broader population. However, in this chapter we will focus on the **prediction tradition**, in which our goal is to use limited sample information to build a model capable of making accurate out-of-sample predictions about new individuals we might come across.

10.1 Wait, aren't those the same thing?

Often, yes, models that are good at making accurate inferences about some population parameter are also good at making predictions about new individuals from that population. The reverse is also often true: models that make accurate predictions are also probably models that can tell us something about the features of a broader population.

However, these two approaches also often imply some differences too. For example, in the inference traditional, **interpretability and explainability** are key. In that tradition, we are deeply skeptical of a model with a high R^2 value, but which can't be quickly explained to us. In contrast, in the prediction tradition, explainability still matters, but not nearly as much. If we can show that a model reliably produces good predictions, we are generally satisfied.

10.2 The `tidymodels` framework

If your goals are different, the tools you use to achieve those goals will generally be different too. Unlike in many other chapters, where everything is just a new flavor of the same old linear regression, in this chapter we introduce a whole new modelling framework, implemented in the `tidymodels` package.

This package was created out of a desire to unify (and thus simplify) the machine learning process in R, which is currently dominated by an array of diverse packages - many of which do the same thing, slightly differently. To help you get through your analysis quickly (and in a less error prone way) than sorting through a bunch of different packages, `tidymodels` will handle a lot under the hood, leaving you with a simple **baking metaphor** to guide you through the process.

10.3 Steps 1 & 2 - Load data and do some very basic cleaning

Again, we load the KSADS dataset with our custom function and do some very basic cleaning. We also load both the `tidyverse` and `tidymodels` packages because we'll be using both throughout.

```
library(tidyverse)
library(tidymodels)
```

```
## Warning: package 'tidymodels' was built under R version
## 4.1.3
```

```
## Registered S3 method overwritten by 'tune':
##   method                   from
##   required_pkgs.model_spec parsnip
```

```
## -- Attaching packages ------------------ tidymodels 0.1.4 --
```

```
## v broom        0.7.11     v rsample      0.1.1
## v dials        0.1.0      v tune         0.1.6
## v infer        1.0.0      v workflows    0.2.4
## v modeldata    0.1.1      v workflowsets 0.2.1
## v parsnip      0.2.0      v yardstick    0.0.9
## v recipes      0.2.0
```

```
## Warning: package 'dials' was built under R version 4.1.3
## Warning: package 'infer' was built under R version 4.1.3

## Warning: package 'modeldata' was built under R version 4.1.3

## Warning: package 'parsnip' was built under R version 4.1.3

## Warning: package 'recipes' was built under R version 4.1.3

## Warning: package 'rsample' was built under R version 4.1.3

## Warning: package 'tune' was built under R version 4.1.3

## Warning: package 'workflows' was built under R version 4.1.3

## Warning: package 'yardstick' was built under R version 4.1.3

## -- Conflicts --------------------- tidymodels_conflicts() --
## x scales::discard() masks purrr::discard()
## x dplyr::filter()   masks stats::filter()
## x recipes::fixed()  masks stringr::fixed()
## x dplyr::lag()      masks stats::lag()
## x yardstick::spec() masks readr::spec()
## x recipes::step()   masks stats::step()
## x tune::tune()      masks parsnip::tune()
## * Search for functions across packages at https://www.tidymodels.org/find/
```

```
read_abcd_quietly <- function(file_path){
  suppressMessages(
    expr = read_delim(file_path, delim = '\t') %>%
      filter(row_number() != 1) %>%
      type_convert())
}

df <- read_abcd_quietly('data/abcd_lpksad01.txt') %>%
  rename(grade_drop = kbi_p_c_drop_in_grades_l) %>%
  filter(grade_drop %in% c(1, 2)) %>%
  mutate(
    grade_drop = factor(
      x = grade_drop,
      levels = c(1, 2),
      labels = c('yes', 'no'))) %>%
  group_by(subjectkey) %>%
  filter(interview_age == min(interview_age))
```

10.4 Step 3 - Split training and test samples

Recall from above that our goal with machine learning is to make good out-of-sample predictions. To test whether our model has done that, we need to split our sample into two parts: a **training** sample that we use to fit and calibrate our model (usually 80% of the total sample) and a smaller **test** sample we can use to evaluate how good that model is at predicting scores it has not seen before (usually 20%). We can do this with the `initial_split()` function.

```
df_split <- initial_split(df, prop = .80)

df_split
```

```
## <Analysis/Assess/Total>
## <8282/2071/10353>
```

As we can see, this splits our data into analysis (training) and assess (test) sub-components. If we want to look at a particular sub-component, we can simply ask for it with a simple helper function. In this case, we use the well-named `training()` function to get a look at which data ended up in the training set.

```
df_split %>%
  training() %>%
  head()
```

```
## # A tibble: 6 x 90
## # Groups:   subjectkey [6]
##   collection_id abcd_lpksad01_id dataset_id subjectkey
##           <dbl>            <dbl>      <dbl> <chr>
## 1          2573            28975      47218 NDAR_INV65X59CTR
## 2          2573            26835      47218 NDAR_INV5JKLGU54
## 3          2573            47030      47218 NDAR_INVURUR52NZ
## 4          2573            47593      47218 NDAR_INVY04AT16M
## 5          2573            41483      47218 NDAR_INVN2CVFJPN
## 6          2573            22789      47218 NDAR_INV030W95VP
## # ... with 86 more variables: src_subject_id <chr>,
## #   interview_age <dbl>, interview_date <chr>, sex <chr>,
## #   eventname <chr>, kbi_l_p_select_language___1 <dbl>,
## #   kbi_p_c_live_full_time_l <dbl>,
## #   kbi_p_c_guard_l___1 <dbl>, kbi_p_c_guard_l___2 <dbl>,
## #   kbi_p_c_guard_l___3 <dbl>, kbi_p_c_guard_l___4 <dbl>,
## #   kbi_p_c_guard_l___5 <dbl>, ...
```

10.5 Step 4 - Build a data processing recipe

As mentioned above, the `tidymodels` framework is built around a cooking metaphor. The idea is that you start with a **recipe**, which is a set of instructions for what you want to do to process your data.

We do this by extracting our data, then telling R that we want a `recipe()` and feed it a formula indicating what we want to use as our outcome variable and what we want to use as our predictors. In this case, the formula `grade_drop ~ .` means “use `grade_drop` as the outcome and everything else you can find as a predictor.

After that, we use any number of `step_...()` functions. These are the steps in our recipe. There are dozens of helpful ones build into `tidymodels`, so make sure to browse the documentation for ones you might like.

In this case, we’ll use a few common ones:

- `step_rm()` removes variables much like `select()` does. In this case, we remove variables that don’t have anything that would help with predicting grade drop (e.g., the `subjectkey` is a randomly generated variable and has nothing to do with grade changes).
- `step_filter_missing()` filters out variables that have too much missing data, based on a threshold we set. In this case, we want to get rid of ANY missing data.
- `step_nzv()` removes variables that have zero (or low) variance, indicating everyone has the same or similar scores and we wont get much information out of those variables.
- `step_corr()` removes variables that are highly correlated with one another because they offer little *unique* information.

Once all of this is done, we call `prep()` to finalize the recipe building process.

```
df_recipe <- df_split %>%
  training() %>%
  recipe(grade_drop ~ .) %>%
  step_rm(ends_with('id') | matches('subjectkey')) %>%
  step_filter_missing(all_predictors(), threshold = 0) %>%
  step_nzv(all_predictors()) %>%
  step_corr(all_numeric_predictors(), threshold = .50) %>%
  prep()

df_recipe
```

```
## Recipe
##
```

```
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor         89
##
## Training data contained 8282 data points and 8282 incomplete rows.
##
## Operations:
##
## Variables removed collection_id, abcd_lpksad01_id, da... [trained]
## Missing value column filter removed kbi_p_c_best_friend_l... [trained]
## Sparse, unbalanced variable filter removed kbi_p_c_guard_l___1... [trained]
## Correlation filter on kbi_p_conflict_causes... [trained]
```

The output we get is helpful here, indicating how many variables are involved in our recipe.

10.6 Step 5 - Extract the preprocessed data

With our preprocessing recipe prepped, we can now `bake()` it. This means we implement the preprocessing instructions on the datasets.

To do this, we say we want to take the recipe, and then bake it using our dataset of choice.

We'll do this once for our training data and once for our testing data. Note, for our testing data we also filter to have only complete cases on our outcome variable as well, which will make evaluation easier later.

```
df_training <- df_recipe %>%
  bake(training(df_split))

df_testing <- df_recipe %>%
  bake(testing(df_split)) %>%
  filter(complete.cases(.))
```

10.7 Step 6 - Fit a model

With our data prepped and baked, it is time to finally fit a model. In this case, we'll fit a random forest model, using the `rand_forest()` function.

Underneath the hood, `rand_forest()` is capable of using a bunch of different "engines" (R packages) to do its computations, each with their own quirks. You won't notice them because `rand_forest()` will make everything run smoothly for you. However, you do need to tell it which engine to use with `set_engine()`. In this case, we'll tell it to use the basic `'ranger'` engine because that is a popular package.

Lastly, with all the options set, we tell R to `fit()` our model, using the familiar formula / data combination of arguments, much like linear regression. The only trick here is again the use of the `.` in our formula, which means "use everything you can find"

```
df_ranger <- rand_forest(trees = 100, mode = 'classification') %>%
  set_engine('ranger') %>%
  fit(grade_drop ~ ., data = df_training)

df_ranger
```

```
## parsnip model object
##
## Ranger result
##
## Call:
##  ranger::ranger(x = maybe_data_frame(x), y = y, num.trees = ~100,      num.threads = 1, verbos
##
## Type:                             Probability estimation
## Number of trees:                  100
## Sample size:                      8282
## Number of independent variables:  20
## Mtry:                             4
## Target node size:                 10
## Variable importance mode:         none
## Splitrule:                        gini
## OOB prediction error (Brier s.):  0.09923773
```

Great! Our output gives us some information about our model. But… it didn't give use a ton of information about how good that model is.

10.8 Step 7 - Evaluate the model

To get information about how well our model did predicting new data, we need to use it to make predictions about our test data, then see how close those predictions were to the actually observed data in that test set that we set aside.

To do that, we take our model, then call `predict()` on our testing data. After that, we use `bind_cols()` to take those predictions and place them in a new dataframe, right next to the observed testing data. With all of that done, we can then use the `metrics()` function to get some performance metrics. All we need to do is tell `metrics()` which column represents the true score and which one is the prediction estimated by the model.

```
df_ranger %>%
  predict(df_testing) %>%
  bind_cols(df_testing) %>%
  metrics(truth = grade_drop, estimate = .pred_class)
```

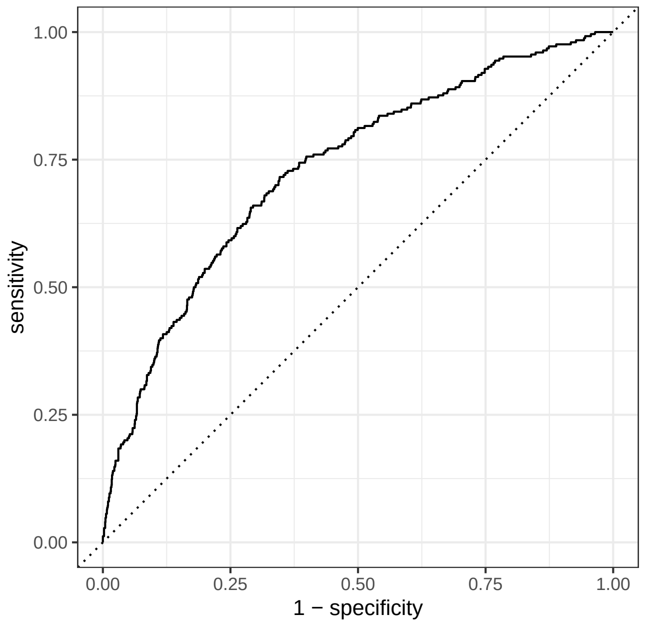
```
## # A tibble: 2 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy binary        0.879
## 2 kap      binary        0.0813
```

Here, we can see that our model is about 88% accurate at predicting a grade drop for a student it has never seen before.

10.9 Step 8 - Plot your data

It is also possible to plot the success of your model, using a similar pipeline. But this time instead of using `metrics()`, we ask R to give us a `roc_curve()` and to graph it with `autoplot()`. This gives us a publication-ready figure, in just two extra lines!

```
df_ranger %>%
  predict(df_testing, type = 'prob') %>%
  bind_cols(df_testing) %>%
  roc_curve(truth = grade_drop, estimate = .pred_yes) %>%
  autoplot()
```


1.00
0.75
0.50
0.25
0.00
sensitivity
0.00
0.25
0.50
0.75
1.00
1 − specificity

Chapter 11

Multilevel Models

In a multilevel model, we try to account not only for group-level variation, but individual variation as well. As a reminder, the way that we do this is by proposing:

1. There is an overall slope and intercept for our regression model, just like in a traditional linear regression.
2. But, there might be some variation from person to person or day to day in the value of that **slope**, **intercept**, or **both**.
    - You can think of this like it being generally true that sleep loss leads to decreased mood the next day, but for some people it is worse than others (**variable slopes**).
    - Alternatively, you might think the damage done by sleep loss is roughly the same from person to person - or perhaps that an intervention is roughly as effective for all people - but that different people start with varying degrees of sleep lost (**variable intercepts**).
    - Lastly, you might think **both**.

The trick to remember is that we are just specifying a traditional regression, *plus* allowing some of its parameters to be slightly different from person to person.

11.1 Steps 1 & 2 - Import and clean the data

Again to maximize comparability with previous chapters on regression techniques, we'll use the same KSADS ABCD dataset.

```r
library(tidyverse)

read_abcd_quietly <- function(file_path){
  suppressMessages(
    expr = read_delim(file_path, delim = '\t') %>%
      filter(row_number() != 1) %>%
      type_convert())
}

df <- read_abcd_quietly('data/abcd_lpksad01.txt')
```

But along the way, it is important to note that this is a **longitudinal** dataset with many participants providing multiple data points to us. You can thus think of those data points as providing multiple pieces of information for each person, which we can use to estimate (for example) their own personal intercept in a multilevel model.

```r
df <- df %>%
  select(
    id = src_subject_id,
    age = interview_age,
    sex,
    grade_drop = kbi_p_c_drop_in_grades_l,
    grades = kbi_p_grades_in_school_l) %>%
  group_by(id) %>%
  arrange(age) %>%
  mutate(
    age = floor(age / 12),
    sex = factor(sex, levels = c('M', 'F')),
    time = row_number(),
    n_timepoints = max(time),
    grade_drop = factor(
      x = grade_drop,
      levels = c(1, 2),
      labels = c('yes', 'no'))) %>%
  ungroup() %>%
  filter(grades %in% 1:5) %>%
  select(id, age, sex, grades, grade_drop, time)
```

11.2 Steps 3 & 4 - Fit the model and summarize it

As we saw with logistic regression, R's regression-centric statistical framework makes transition to a new type of regression easy for us to think about: almost everything is the same, we just need to tweak a few specifics.

The first thing to note is that we'll need to load the `lme4` package, which contains a function to fit a multilevel model. Next, we use the `lmer()` function to fit the model, rather than the traditional `lm()` for a classic linear regression.

After that, the last obvious change we need to make is that we need to tell R which parameters we want to vary and what our nesting structure is. Fortunately, that is pretty easy too. We just add it to our regression formula like this `+ (params_to_vary | vary_by_what)`.

As you can see below, the randomly varying part of our model is specified as `(1 | id)`. Because `1` is R's indicator for an intercept and `id` is our dataset's variable indicating which person we are talking about (remember each ID will have multiple timepoints), this specification will produce a regression where the slope stays constant for everyone, BUT everyone has their own special intercept (i.e. a random intercept only model).

```r
library(lme4)
```

```
## Loading required package: Matrix
```

```
##
## Attaching package: 'Matrix'
```

```
## The following objects are masked from 'package:tidyr':
##
##     expand, pack, unpack
```

```r
fit <- lmer(
  formula = grades ~ age + sex + (1 | id),
  data = df)

summary(fit)
```

```
## Linear mixed model fit by REML ['lmerMod']
## Formula: grades ~ age + sex + (1 | id)
##    Data: df
##
```

```
## REML criterion at convergence: 50990.9
##
## Scaled residuals:
##     Min      1Q  Median      3Q     Max
## -4.3199 -0.3357 -0.1901  0.2137  5.2364
##
## Random effects:
##  Groups   Name        Variance Std.Dev.
##  id       (Intercept) 0.4181   0.6466
##  Residual             0.2304   0.4800
## Number of obs: 24771, groups:  id, 10301
##
## Fixed effects:
##              Estimate Std. Error t value
## (Intercept)  1.654237   0.044601  37.089
## age          0.011209   0.003846   2.915
## sexF        -0.200611   0.014240 -14.088
##
## Correlation of Fixed Effects:
##      (Intr) age
## age  -0.975
## sexF -0.160  0.008
```

As you can see, the output is again quite similar to the regular `lm()` output. However, there are two twists.

1. Because we have a random intercept, the intercept now has a group-level mean (under Fixed Effects) AND it has its own variance (under Random Effects).
2. We're missing p-values! Find out how to get those below.

11.2.1 Where are my p-values?

Multilevel models are mathematically complicated, under the hood. For reasons we wont go into here, there are many ways to compute their p-values and the `lme4` package doesn't want to make the choice for you out of an abundance of caution.

To get our p-values, we can use the `lmerTest` package, which will override the `lmer()` function to include p-values. Thus, if we run our same model again, but with `lmerTest` loaded we should get what we expected.

```
library(lmerTest)
```

```
##
## Attaching package: 'lmerTest'

## The following object is masked _by_ '.GlobalEnv':
##
##     carrots

## The following object is masked from 'package:lme4':
##
##     lmer

## The following object is masked from 'package:recipes':
##
##     step

## The following object is masked from 'package:stats':
##
##     step
```

```
fit <- lmer(
  formula = grades ~ age + sex + (1 | id),
  data = df)

summary(fit)
```

```
## Linear mixed model fit by REML. t-tests use
##   Satterthwaite's method [lmerModLmerTest]
## Formula: grades ~ age + sex + (1 | id)
##    Data: df
##
## REML criterion at convergence: 50990.9
##
## Scaled residuals:
##     Min      1Q  Median      3Q     Max
## -4.3199 -0.3357 -0.1901  0.2137  5.2364
##
## Random effects:
##  Groups   Name        Variance Std.Dev.
##  id       (Intercept) 0.4181   0.6466
##  Residual             0.2304   0.4800
## Number of obs: 24771, groups:  id, 10301
##
## Fixed effects:
##               Estimate Std. Error         df t value
## (Intercept)  1.654e+00  4.460e-02  2.018e+04  37.089
## age          1.121e-02  3.846e-03  1.888e+04   2.915
## sexF        -2.006e-01  1.424e-02  1.013e+04 -14.088
##             Pr(>|t|)
## (Intercept)  < 2e-16 ***
## age          0.00357 **
## sexF         < 2e-16 ***
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Correlation of Fixed Effects:
##      (Intr) age
## age  -0.975
## sexF -0.160  0.008
```

11.2.2 Where are my intraclass correlations (ICCs)?

One measure of the strength of variation for a random effect in a multilevel model is the Intraclass Correlation Coefficient (ICC). These tell you the percentage of variation in the your estimate can be attributed to random variation from person to person.

Unfortunately, `lmer()` doesn't compute this on its own, so we need to ask the `performance` package to do it for us.

```
performance::icc(fit)
```

```
## # Intraclass Correlation Coefficient
##
##      Adjusted ICC: 0.645
##   Conditional ICC: 0.635
```

11.3 Logistic multilevel models

So far, we've fit just linear multilevel models, but note that the transition to a logistic version is the same as the transition from a traditional linear model to a traditional logistic one:

- `lm()` becomes `glm()`; `lmer()` becomes `glmer()`
- We need to specify the family of non-linear model we are using, which is again "binomial"

Then everything else is the same.

```
fit <- glmer(
  formula = grade_drop ~ age + sex + (1 | id),
  data = df,
  family = 'binomial')

summary(fit)
```

```
## Generalized linear mixed model fit by maximum likelihood
##   (Laplace Approximation) [glmerMod]
##  Family: binomial  ( logit )
## Formula: grade_drop ~ age + sex + (1 | id)
##    Data: df
##
##      AIC      BIC   logLik deviance df.resid
##  20086.6  20119.0 -10039.3  20078.6    24440
##
## Scaled residuals:
##     Min      1Q  Median      3Q     Max
## -3.0384  0.2134  0.2559  0.3115  1.0890
##
## Random effects:
##  Groups Name        Variance Std.Dev.
##  id     (Intercept) 2.064    1.437
## Number of obs: 24444, groups:  id, 10275
##
## Fixed effects:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)  5.87861    0.26782  21.950   <2e-16 ***
## age         -0.33012    0.02241 -14.728   <2e-16 ***
## sexF         0.46215    0.05188   8.908   <2e-16 ***
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Correlation of Fixed Effects:
##      (Intr) age
## age  -0.985
## sexF -0.049 -0.023
## optimizer (Nelder_Mead) convergence code: 0 (OK)
## Model failed to converge with max|grad| = 0.00490794 (tol = 0.002, component 1)
```

Chapter 12

Structural Equation Modeling

Structural equation modeling (SEM) can be thought of as a series of regressions, all estimated simultaneously. This allows the relationship between some variables to provide information on the relationship between others. To get a feel for how this works, consider that if we have three variables (A, B, C) and two negative correlations (A is negatively correlated with B, B is negatively correlated with C), we know that A and C MUST be positively correlated. Thus, if I estimate all of these correlations simultaneously, I can get a more precise estimate for all of their relationships by incorporating all possible information.

There are many kinds of SEMS, so we'll cover just a few of the most common ones:

- Exploratory factor analysis (EFA), in which we ask whether some hidden factor accounts for the observed correlation between several items (usually items from the same survey scale). In the case of EFA, we also don't really have any prior knowledge as to what this hidden variable might be like, so we let the model figure it out for us.

- Confirmatory factor analysis (CFA), which is like EFA, but we force the model to fit an *a priori* idea that we have about the relationship between our observed variables and the hidden variable that explains their correlations. After we force a specific model, we then evaluate how well it fit our observed data.

- Path models. These are often what people think of when they hear "SEM." These are typically just a series of regressions (paths) connecting variables in a kind of spider web of relationships.

12.1 Step 1 and 2 - Load data and clean

Because we want to evaluate the factors underlying particular scales, we'll start by using the `diff_emotion_reg_p01.txt` dataset, which includes data from an emotion regulation survey scale.

```
library(tidyverse)

read_abcd_quietly <- function(file_path){
  suppressMessages(
    expr = read_delim(file_path, delim = '\t') %>%
      filter(row_number() != 1) %>%
      type_convert())
}

df <- read_abcd_quietly('data/diff_emotion_reg_p01.txt') %>%
  mutate(across(
    .cols = starts_with('ders_'),
    .fns = ~ ifelse(.x == 7, NA, .x)))
```

12.2 EFA

12.2.1 Steps 3 & 4

To conduct an EFA, we'll use functions from the `psych` package. This process is luckily relatively straightforward. All we need to do is select the variables we want to analyze, then feed them to the `fa()` function. The `fa()` function also wants to know what the max number of factors we want to look for is, so we'll tell it 6 (it this case, we know this scale has 6 factors because it has been extensively studied before).

```
library(psych)
```

```
## Warning: package 'psych' was built under R version 4.1.3
```

```
##
## Attaching package: 'psych'
```

```
## The following objects are masked from 'package:scales':
##
##     alpha, rescale
## The following objects are masked from 'package:ggplot2':
##
##     %+%, alpha
```

```
ders_df <- df %>%
  select(starts_with('ders_') & ends_with('_p'))

fit <- fa(ders_df, nfactors = 6)
```

```
## Loading required namespace: GPArotation
```

```
summary(fit)
```

```
##
## Factor analysis with Call: fa(r = ders_df, nfactors = 6)
##
## Test of the hypothesis that 6 factors are sufficient.
## The degrees of freedom for the model is 247  and the objective function was  3.01
## The number of observations was  6251  with Chi Square =  18775.94  with prob <  0
##
## The root mean square of the residuals (RMSA) is  0.02
## The df corrected root mean square of the residuals is  0.03
##
## Tucker Lewis Index of factoring reliability =  0.837
## RMSEA index =  0.11  and the 10 % confidence intervals are  0.108 0.111
## BIC =  16617.04
##  With factor correlations of
##      MR5  MR2  MR1  MR4  MR3  MR6
## MR5 1.00 0.56 0.53 0.61 0.55 0.57
## MR2 0.56 1.00 0.31 0.51 0.45 0.63
## MR1 0.53 0.31 1.00 0.33 0.46 0.38
## MR4 0.61 0.51 0.33 1.00 0.50 0.47
## MR3 0.55 0.45 0.46 0.50 1.00 0.45
## MR6 0.57 0.63 0.38 0.47 0.45 1.00
```

Unfortunately, the output we get from `summary()` doesn't tell us which items load onto which factors. To get that, we need to extract them from the `fit` object directly.

```
fit$loadings
```

```
##
## Loadings:
```

```
##                                MR5    MR2    MR1    MR4
## ders_attn_awareness_p                 0.953
## ders_clear_feelings_p                 0.902
## ders_emotion_overwhelm_p        0.119  0.543 -0.107  0.128
## ders_feelings_attentive_p             0.662
## ders_feelings_care_p
## ders_feelings_know_p                  0.278         0.112
## ders_upset_ack_p                0.174  0.243 -0.163  0.120
## ders_upset_angry_p                            0.149  0.635
## ders_upset_ashamed_p            0.169  0.129  0.546  0.271
## ders_upset_behavior_control_p   0.224         0.228  0.303
## ders_upset_behavior_p           0.853
## ders_upset_better_p             0.311         0.161  0.112
## ders_upset_concentrate_p        0.673
## ders_upset_control_p            0.504        -0.265  0.349
## ders_upset_depressed_p          0.138         0.218
## ders_upset_difficulty_p         0.262  0.272         0.366
## ders_upset_embarrassed_p                      0.162  0.677
## ders_upset_emotion_overwhelm_p  0.521         0.247
## ders_upset_esteem_p             0.200         0.489
## ders_upset_feel_better_p        0.345  0.130  0.416
## ders_upset_fixation_p           0.834
## ders_upset_focus_p              0.365  0.220 -0.139  0.397
## ders_upset_guilty_p                           0.640  0.132
## ders_upset_irritation_p         0.231  0.108  0.548  0.130
## ders_upset_long_time_better_p   0.606  0.136
## ders_upset_lose_control_p       0.657                0.106
## ders_upset_out_control_p        0.304  0.171  0.211  0.383
## ders_upset_time_p
## ders_upset_weak_p                             0.495
##                                MR3    MR6
## ders_attn_awareness_p
## ders_clear_feelings_p
## ders_emotion_overwhelm_p
## ders_feelings_attentive_p
## ders_feelings_care_p                  0.903
## ders_feelings_know_p                  0.511
## ders_upset_ack_p                0.184  0.281
## ders_upset_angry_p                    0.185
## ders_upset_ashamed_p
## ders_upset_behavior_control_p
## ders_upset_behavior_p
## ders_upset_better_p             0.111  0.140
## ders_upset_concentrate_p       -0.104  0.192
## ders_upset_control_p            0.275
## ders_upset_depressed_p          0.581
## ders_upset_difficulty_p
## ders_upset_embarrassed_p              0.217
## ders_upset_emotion_overwhelm_p
## ders_upset_esteem_p            0.296  0.128
## ders_upset_feel_better_p       0.176
## ders_upset_fixation_p
## ders_upset_focus_p             0.150
## ders_upset_guilty_p            0.121
## ders_upset_irritation_p               0.112
## ders_upset_long_time_better_p         0.138
## ders_upset_lose_control_p
## ders_upset_out_control_p       0.109 -0.185
## ders_upset_time_p              0.997
## ders_upset_weak_p              0.216
##
##                  MR5   MR2   MR1   MR4   MR3   MR6
## SS loadings    3.977 2.845 2.105 1.718 1.723 1.426
## Proportion Var 0.137 0.098 0.073 0.059 0.059 0.049
## Cumulative Var 0.137 0.235 0.308 0.367 0.426 0.476
```

12.3 CFA

12.3.1 Step 3 & 4

Above, we did an exploratory factor analysis, but now we want to run a confirmatory one. To do that, we'll use the `lavaan` package, which is the most popular pacakge in R for SEM-based analyses.

Like a basic regression, we'll need a model formula and some data. But unlike a basic regression, CFA and broader SEM usually have several formulas at once. To handle this, we'll write down our model as a string and save it in a `my_model` variable to make things easier to read.

After that, we just send the model to the `cfa()` function, which does the rest of the work for us. A quick call to `summary()` and we're done!

```
library(lavaan)
```

```
## Warning: package 'lavaan' was built under R version 4.1.3
```

```
## This is lavaan 0.6-11
## lavaan is FREE software! Please report any bugs.
```

```
##
## Attaching package: 'lavaan'
## The following object is masked from 'package:psych':
##
##     cor2cov
```

```
my_model <- '
  b =~ ders_attn_awareness_p + ders_clear_feelings_p +
    ders_emotion_overwhelm_p + ders_feelings_attentive_p +
    ders_feelings_care_p + ders_feelings_know_p + ders_upset_ack_p

  c =~ ders_upset_angry_p + ders_upset_control_p +
    ders_upset_difficulty_p + ders_upset_embarrassed_p +
    ders_upset_focus_p + ders_upset_out_control_p

  a =~ ders_upset_ashamed_p + ders_upset_behavior_control_p +
    ders_upset_depressed_p + ders_upset_esteem_p +
    ders_upset_feel_better_p + ders_upset_guilty_p +
    ders_upset_irritation_p + ders_upset_time_p +
    ders_upset_weak_p

  d =~ ders_upset_behavior_p + ders_upset_better_p +
    ders_upset_concentrate_p + ders_upset_emotion_overwhelm_p +
    ders_upset_fixation_p + ders_upset_long_time_better_p +
    ders_upset_lose_control_p
'

fit <- cfa(
  model = my_model,
  data = df)

summary(fit, fit.measures = T)
```

```
## lavaan 0.6-11 ended normally after 300 iterations
##
##   Estimator                                         ML
##   Optimization method                           NLMINB
##   Number of model parameters                        64
##
##                                                   Used       Total
##   Number of observations                          6147        6251
##
## Model Test User Model:
##
##   Test statistic                               35452.539
##   Degrees of freedom                                 371
##   P-value (Chi-square)                             0.000
```

```
##
## Model Test Baseline Model:
##
##   Test statistic                            184927.008
##   Degrees of freedom                               406
##   P-value                                        0.000
##
## User Model versus Baseline Model:
##
##   Comparative Fit Index (CFI)                    0.810
##   Tucker-Lewis Index (TLI)                       0.792
##
## Loglikelihood and Information Criteria:
##
##   Loglikelihood user model (H0)            -877024.922
##   Loglikelihood unrestricted model (H1)    -859298.653
##
##   Akaike (AIC)                             1754177.844
##   Bayesian (BIC)                           1754608.163
##   Sample-size adjusted Bayesian (BIC)      1754404.788
##
## Root Mean Square Error of Approximation:
##
##   RMSEA                                          0.124
##   90 Percent confidence interval - lower         0.123
##   90 Percent confidence interval - upper         0.125
##   P-value RMSEA <= 0.05                          0.000
##
## Standardized Root Mean Square Residual:
##
##   SRMR                                           0.058
##
## Parameter Estimates:
##
##   Standard errors                             Standard
##   Information                                 Expected
##   Information saturated (h1) model          Structured
##
## Latent Variables:
##                    Estimate  Std.Err  z-value  P(>|z|)
##   b =~
##     drs_ttn_wrnss_    1.000
##     drs_clr_flngs_    0.928    0.011   85.836    0.000
##     drs_mtn_vrwhl_    0.730    0.015   49.818    0.000
##     drs_flngs_ttn_    1.071    0.017   62.857    0.000
##     drs_flngs_cr_p    0.985    0.013   73.134    0.000
##      drs_flngs_knw_    1.020    0.013   80.144    0.000
##      ders_upst_ck_p    0.760    0.012   63.093    0.000
##    c =~
##      drs_pst_ngry_p    1.000
##      drs_pst_cntrl_    0.702    0.008   84.481    0.000
##      drs_pst_dffcl_    0.773    0.011   71.472    0.000
##      drs_pst_mbrrs_    0.986    0.012   81.388    0.000
##      ders_pst_fcs_p    0.790    0.009   87.752    0.000
##      drs_pst_t_cnt_    0.870    0.012   72.614    0.000
##    a =~
##      drs_pst_shmd_p    1.000
##      drs_pst_bhvr__    0.830    0.014   60.223    0.000
##      drs_pst_dprss_    0.871    0.010   86.053    0.000
##      ders_pst_stm_p    1.022    0.010  100.690    0.000
##      drs_pst_fl_bt_    0.931    0.009  102.532    0.000
##      drs_pst_glty_p    1.128    0.012   94.694    0.000
##      drs_pst_rrttn_    1.003    0.009  114.024    0.000
##      ders_upst_tm_p    0.758    0.012   64.555    0.000
##      ders_upst_wk_p    1.073    0.016   68.799    0.000
##    d =~
##      drs_pst_bhvr_p    1.000
##      drs_pst_bttr_p    1.041    0.017   61.130    0.000
##      drs_pst_cncnt_    1.059    0.011   94.852    0.000
##      drs_pst_mtn_v_    1.246    0.014   91.898    0.000
##      drs_pst_fxtn_p    1.187    0.011  105.360    0.000
##      drs_pst_lng___    1.139    0.012   92.068    0.000
##      drs_pst_ls_cn_    0.977    0.010   93.973    0.000
##
## Covariances:
##                    Estimate  Std.Err  z-value  P(>|z|)
##    b ~~
##      c             1306.707   31.057   42.074    0.000
##      a             1314.713   32.965   39.882    0.000
##      d              977.803   23.843   41.010    0.000
##    c ~~
##      a             1889.998   43.083   43.868    0.000
##      d             1409.192   31.338   44.968    0.000
##    a ~~
##      d             1647.825   35.449   46.485    0.000
##
## Variances:
##                    Estimate  Std.Err  z-value  P(>|z|)
##     .drs_ttn_wrnss_  513.518   11.572   44.378    0.000
##     .drs_clr_flngs_  406.478    9.368   43.391    0.000
##     .drs_mtn_vrwhl_ 1338.981   25.163   53.212    0.000
##     .drs_flngs_ttn_ 1574.552   30.665   51.347    0.000
##    .drs_flngs_cr_p  842.167   17.209   48.936    0.000
##    .drs_flngs_knw_  652.877   14.069   46.406    0.000
##    .ders_upst_ck_p  785.343   15.308   51.303    0.000
##    .drs_pst_ngry_p  762.693   16.279   46.851    0.000
##    .drs_pst_cntrl_  396.881    8.388   47.316    0.000
##    .drs_pst_dffcl_  864.936   16.978   50.945    0.000
##    .drs_pst_mbrrs_  905.787   18.706   48.423    0.000
##    .ders_pst_fcs_p  428.311    9.333   45.890    0.000
##    .drs_pst_t_cnt_ 1041.786   20.543   50.713    0.000
##    .drs_pst_shmd_p  645.893   13.927   46.376    0.000
##    .drs_pst_bhvr__ 2521.764   46.826   53.854    0.000
##    .drs_pst_dprss_ 1100.835   21.405   51.429    0.000
##    .ders_pst_stm_p  918.441   18.823   48.794    0.000
##    .drs_pst_fl_bt_  714.048   14.770   48.343    0.000
##    .drs_pst_glty_p 1376.005   27.494   50.048    0.000
##    .drs_pst_rrttn_  538.793   12.109   44.496    0.000
##    .ders_upst_tm_p 1781.098   33.250   53.566    0.000
##    .ders_upst_wk_p 3045.811   57.202   53.247    0.000
##    .drs_pst_bhvr_p  341.520    7.431   45.958    0.000
##    .drs_pst_bttr_p 1865.524   34.818   53.580    0.000
##    .drs_pst_cncnt_  568.218   11.578   49.079    0.000
##    .drs_pst_mtn_v_  874.766   17.593   49.723    0.000
##    .drs_pst_fxtn_p  482.445   10.491   45.987    0.000
##    .drs_pst_lng___  726.835   14.628   49.688    0.000
##    .drs_pst_ls_cn_  499.455   10.135   49.279    0.000
##     b              1331.862   32.715   40.710    0.000
##     c              1973.464   48.190   40.951    0.000
##     a              2650.912   58.812   45.074    0.000
##     d              1316.773   29.564   44.539    0.000
```

12.4 SEM

12.4.1 Steps 3 & 4

The CFA above shows us that there are several latent variables that underlie the scale in this dataset, which we have labelled `a` through `d`. Do the scores on one factor predict scores on another? Does the sex of a subject predict their scores on these hidden variables? SEM is the technique that answers this question. All we need to to to get there is to combine our existing model with some existing commands.

Here, we are saying that we expect variable `a` to be predicted by variable `b`, as well as subject sex.

To demonstrate another trick in the next line, we'll tell R that we want our model to force the correlation between `b` and `c` to be exactly 0. Why do this? In this case, there isn't really a reason, but in broader SEM it helps to have this tool in your belt, so we demonstrate how to force model constraintes here in case you ever need them.

```
new_model <- paste(
  my_model,
  '
  a ~ b + sex
  b ~~ 0*c')
```

With our new model specified, we can safely proceed to estimate our SEM and ask for a summary in the usual way.

```
fit <- sem(
  model = new_model,
  data = df,
  estimator = 'MLR',
  missing = 'ML')

summary(fit)
```

```
## lavaan 0.6-11 ended normally after 433 iterations
##
##   Estimator                                         ML
##   Optimization method                           NLMINB
##   Number of model parameters                        91
##
##   Number of observations                          6251
##   Number of missing patterns                         2
##
## Model Test User Model:
##                                               Standard      Robust
##   Test Statistic                             44875.439     969.709
##   Degrees of freedom                               402         402
##   P-value (Chi-square)                           0.000       0.000
##   Scaling correction factor                                 46.277
##        Yuan-Bentler correction (Mplus variant)
##
## Parameter Estimates:
##
##   Standard errors                             Sandwich
##   Information bread                           Observed
##   Observed information based on                Hessian
##
```

```
## Latent Variables:
##                    Estimate  Std.Err  z-value  P(>|z|)
##   b =~
##     drs_ttn_wrnss_    1.000
##     drs_clr_flngs_    0.922    0.045   20.477    0.000
##     drs_mtn_vrwhl_    0.712    0.118    6.055    0.000
##     drs_flngs_ttn_    1.073    0.087   12.339    0.000
##     drs_flngs_cr_p    0.971    0.203    4.783    0.000
##     drs_flngs_knw_    1.002    0.170    5.885    0.000
##     ders_upst_ck_p    0.739    0.169    4.381    0.000
##   c =~
##     drs_pst_ngry_p    1.000
##     drs_pst_cntrl_    0.716    0.102    7.009    0.000
##     drs_pst_dffcl_    0.771    0.102    7.584    0.000
##     drs_pst_mbrrs_    0.993    0.022   46.136    0.000
##     ders_pst_fcs_p    0.796    0.091    8.765    0.000
##     drs_pst_t_cnt_    0.870    0.075   11.650    0.000
##   a =~
##     drs_pst_shmd_p    1.000
##     drs_pst_bhvr__    0.813    0.080   10.207    0.000
##     drs_pst_dprss_    0.867    0.093    9.342    0.000
##     ders_pst_stm_p    1.022    0.070   14.501    0.000
##     drs_pst_fl_bt_    0.921    0.068   13.504    0.000
##     drs_pst_glty_p    1.134    0.056   20.143    0.000
##     drs_pst_rrttn_    1.001    0.053   18.806    0.000
##     ders_upst_tm_p    0.758    0.106    7.132    0.000
##     ders_upst_wk_p    1.076    0.079   13.565    0.000
##   d =~
##     drs_pst_bhvr_p    1.000
##     drs_pst_bttr_p    1.026    0.099   10.395    0.000
##     drs_pst_cncnt_    1.058    0.106    9.990    0.000
##     drs_pst_mtn_v_    1.228    0.126    9.718    0.000
##     drs_pst_fxtn_p    1.184    0.108   10.985    0.000
##     drs_pst_lng___    1.138    0.108   10.570    0.000
##     drs_pst_ls_cn_    0.979    0.111    8.784    0.000
##
## Regressions:
##                    Estimate  Std.Err  z-value  P(>|z|)
##   a ~
##     b                 0.994    0.179    5.553    0.000
##     sex              -0.200    1.011   -0.198    0.843
##
## Covariances:
##                    Estimate  Std.Err  z-value  P(>|z|)
##   b ~~
##     c                 0.000
##     d               331.406  212.968    1.556    0.120
##   c ~~
##     d              1085.534  369.380    2.939    0.003
##
## Intercepts:
##                    Estimate  Std.Err  z-value  P(>|z|)
##    .drs_ttn_wrnss_    6.262    0.548   11.437    0.000
##    .drs_clr_flngs_    5.890    0.503   11.719    0.000
##    .drs_mtn_vrwhl_    4.458    0.577    7.724    0.000
##    .drs_flngs_ttn_    7.605    0.710   10.711    0.000
##    .drs_flngs_cr_p    6.691    0.589   11.359    0.000
##    .drs_flngs_knw_    6.395    0.576   11.110    0.000
##    .ders_upst_ck_p    5.698    0.503   11.334    0.000
##    .drs_pst_ngry_p    5.373    0.666    8.069    0.000
##    .drs_pst_cntrl_    3.283    0.472    6.960    0.000
##    .drs_pst_dffcl_    5.246    0.577    9.098    0.000
##    .drs_pst_mbrrs_    5.460    0.678    8.058    0.000
##    .ders_pst_fcs_p    4.418    0.519    8.510    0.000
##    .drs_pst_t_cnt_    4.830    0.642    7.523    0.000
##    .drs_pst_shmd_p    6.082    1.722    3.531    0.000
##    .drs_pst_bhvr__    9.136    1.587    5.757    0.000
##    .drs_pst_dprss_    5.656    1.579    3.581    0.000
##    .ders_pst_stm_p    6.690    1.784    3.750    0.000
##    .drs_pst_fl_bt_    5.815    1.624    3.581    0.000
##    .drs_pst_glty_p    8.072    1.998    4.040    0.000
##    .drs_pst_rrttn_    6.049    1.748    3.460    0.001
##    .ders_upst_tm_p    6.030    1.464    4.118    0.000
##    .ders_upst_wk_p    9.695    2.009    4.826    0.000
##    .drs_pst_bhvr_p    3.928    0.519    7.562    0.000
##    .drs_pst_bttr_p    7.784    0.732   10.635    0.000
##    .drs_pst_cncnt_    5.052    0.577    8.761    0.000
##    .drs_pst_mtn_v_    5.647    0.689    8.194    0.000
##    .drs_pst_fxtn_p    5.027    0.617    8.153    0.000
##    .drs_pst_lng___    4.801    0.630    7.627    0.000
##    .drs_pst_ls_cn_    3.824    0.535    7.154    0.000
##     b                 0.000
##     c                 0.000
##    .a                 0.000
##     d                 0.000
##
## Variances:
##                    Estimate  Std.Err  z-value  P(>|z|)
##    .drs_ttn_wrnss_  492.585  294.777    1.671    0.095
##    .drs_clr_flngs_  399.547  235.111    1.699    0.089
##    .drs_mtn_vrwhl_ 1359.647  332.156    4.093    0.000
##    .drs_flngs_ttn_ 1539.259  390.781    3.939    0.000
##    .drs_flngs_cr_p  854.744  343.692    2.487    0.013
##    .drs_flngs_knw_  674.489  274.379    2.458    0.014
##    .ders_upst_ck_p  813.010  249.011    3.265    0.001
##    .drs_pst_ngry_p  771.454  257.955    2.991    0.003
##    .drs_pst_cntrl_  371.548  123.848    3.000    0.003
##    .drs_pst_dffcl_  890.127  253.745    3.508    0.000
##    .drs_pst_mbrrs_  909.902  298.617    3.047    0.002
##    .ders_pst_fcs_p  427.696  158.159    2.704    0.007
##    .drs_pst_t_cnt_ 1062.867  289.565    3.671    0.000
##    .drs_pst_shmd_p  644.235  201.146    3.203    0.001
##    .drs_pst_bhvr__ 2586.715  451.488    5.729    0.000
##    .drs_pst_dprss_ 1106.487  269.756    4.102    0.000
##    .ders_pst_stm_p  901.646  241.954    3.727    0.000
##    .drs_pst_fl_bt_  751.413  192.760    3.898    0.000
##    .drs_pst_glty_p 1318.051  322.353    4.089    0.000
##    .drs_pst_rrttn_  534.993  171.841    3.113    0.002
##    .ders_upst_tm_p 1771.492  359.545    4.927    0.000
##    .ders_upst_wk_p 3005.514  507.729    5.920    0.000
##    .drs_pst_bhvr_p  331.459  106.528    3.111    0.002
##    .drs_pst_bttr_p 1897.752  392.567    4.834    0.000
##    .drs_pst_cncnt_  561.320  186.090    3.016    0.003
##    .drs_pst_mtn_v_  921.990  256.741    3.591    0.000
##    .drs_pst_fxtn_p  480.142  158.951    3.021    0.003
##    .drs_pst_lng___  721.445  217.179    3.322    0.001
##    .drs_pst_ls_cn_  487.095  161.691    3.012    0.003
##     b              1362.952  424.069    3.214    0.001
##     c              1920.212  424.074    4.528    0.000
##    .a              1342.507  419.220    3.202    0.001
##     d               998.399  337.969    2.954    0.003
```

12.4.2 Modification indices

Not all models fit well. When they don't it is often evidence that something in the model is mis-specified (e.g., forced to 0, when it should be estimated). **Modification indices** give you hints at which parameters you might want to free and they are easy to get with the `modindices()` function.

```r
modindices(fit, sort = TRUE, maximum.number = 5)
```

```
##                          lhs op                      rhs
## 32                         b ~~                        c
## 190   ders_attn_awareness_p ~~    ders_clear_feelings_p
## 367      ders_upset_angry_p ~~ ders_upset_embarrassed_p
## 509 ders_upset_depressed_p ~~        ders_upset_time_p
## 296   ders_feelings_care_p ~~     ders_feelings_know_p
##           mi      epc sepc.lv sepc.all sepc.nox
## 32  3645.559 1344.429   0.831    0.831    0.831
## 190 2936.500  454.475 454.475    1.024    1.024
## 367 2263.767  645.209 645.209    0.770    0.770
## 509 2124.146  878.919 878.919    0.628    0.628
## 296 1691.296  484.229 484.229    0.638    0.638
```

Chapter 13

Strings, should you care?

When it comes to learning R, there are two topics that almost no one ever *asks* for, but they still often appreciate having after the fact. The first of these is string manipulation.

Here, we show off some of R's text manipulation abilities. If you see something you like, feel free to incorporate it.

13.1 A recurring example

Throughout this exercise, we'll imagine we have an ongoing research study with multiple sources of important information:

1. A `contact_df`, which includes identifying contact information about the subjects in our study, some of which we need for administrative purposes, but some of which we also need for analysis (e.g., birth dates to calculate age, which will be used as a covariate in a regression.).

2. A `mood_df`, which contains the substantive information the study is about, including a mood score.

Because each dataset only includes three people, they are easy to take a glance at below.

```
library(tidyverse)

contact_df <- read_csv('data/contact_info.csv')
```

```
## Rows: 3 Columns: 11
## -- Column specification ------------------------------------
## Delimiter: ","
## chr (8): form_1_timestamp, fname, lname, phone, address,...
## dbl (2): record_id, form_1_complete
## lgl (1): redcap_survey_identifier
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
mood_df <- read_csv('data/mood_data.csv')
```

```
## Rows: 3 Columns: 3
## -- Column specification ------------------------------------
## Delimiter: ","
## chr (1): name
## dbl (2): age, mood_score
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
contact_df
```

```
## # A tibble: 3 x 11
##   record_id redcap_survey_iden~ form_1_timestamp fname lname
##       <dbl> <lgl>               <chr>            <chr> <chr>
## 1         1 NA                  4/14/2022 14:30  Ian   Cero
## 2         2 NA                  4/14/2022 14:32  Test  McFa~
## 3         3 NA                  4/14/2022 14:34  Exam~ McFa~
## # ... with 6 more variables: phone <chr>, address <chr>,
## #   tax_day <chr>, tax_day_wish <chr>, open_ended <chr>,
## #   form_1_complete <dbl>
```

```
mood_df
```

```
## # A tibble: 3 x 3
##   name              age mood_score
##   <chr>           <dbl>      <dbl>
## 1 CERO, Ian          25          6
## 2 MCFAKE, Test       26          9
## 3 MCFAKE, Example    25          3
```

13.2 Join the datasets

One of the first, most obvious things we want to do might be joining these two datasets. That *should* be easy because we have names in both of them.

But wait! The names aren't formatted in the same way. In the mood dataset, they are formatted like "CERO, Ian", but in the contact dataset, they are formatted with two seperate variables: `fname` = Ian, `lname` = Cero. All of the joining operations we know require that at least one column is *identical* across the datasets. That's how we know which scores to link.

To solve this, we can simply create a new `name` variable in the contact dataframe, using `paste()` and `toupper()`. Now, we will have a column with identical formatting in each dataframe.

```
contact_df <- contact_df %>%
  mutate(
    name = paste(toupper(lname), fname, sep = ', '))

contact_df <- contact_df %>%
  left_join(mood_df, by = 'name')
```

13.2.1 Direction / order of operations

Can we go the other way around, from mood to contact? Yes, but takes more work, as you can see below. This is an important lesson about R's string functions: there is typically more than one way to solve a problem, but they are not all equally valuable. Sometimes, one is much harder than the others.

```
mood_df <- mood_df %>%
  mutate(
    lname = str_extract(name, '\\w+') %>%
      tolower() %>%
      tools::toTitleCase() %>%
      str_replace('Mcf', 'McF'),
    fname = str_extract(name, ', \\w+') %>%
      str_remove(', '))

mood_df %>%
  left_join(contact_df, by = c('fname', 'lname'))
```

```
## # A tibble: 3 x 17
##   name.x          age.x mood_score.x lname  fname  record_id
##   <chr>           <dbl>        <dbl> <chr>  <chr>      <dbl>
## 1 CERO, Ian          25            6 Cero   Ian            1
## 2 MCFAKE, Test       26            9 McFake Test           2
## 3 MCFAKE, Example    25            3 McFake Examp~         3
## # ... with 11 more variables:
## #   redcap_survey_identifier <lgl>, form_1_timestamp <chr>,
## #   phone <chr>, address <chr>, tax_day <chr>,
## #   tax_day_wish <chr>, open_ended <chr>,
## #   form_1_complete <dbl>, name.y <chr>, age.y <dbl>,
## #   mood_score.y <dbl>
```

13.3 Extract some information

Often, a column in a spreadsheet holds many pieces of information… and we need just one specific piece. For example, a person's address has much information about their location, but the zipcode is typically most useful for statistical analysis because we can associate it withe median income.

The problem comes in when we need *just* that one piece of information in a new column. Again, R comes to the rescue, this time with the `str_extract()` function, which takes two arguments: the text we want to grab something from, and the "pattern" of things we want to extract. In this case, the pattern `'[0-9]+$'` tells R to grab any digit from 0-9. The `+` says that it needs to be an unbroken sequence of digits and the `$` says it must come at the end of the main text, which is where zipcodes are always located.

```
contact_df %>%
  mutate(
    zip = str_extract(address, '[0-9]+$')) %>%
  select(address, zip)
```

```
## # A tibble: 3 x 2
##   address                          zip
##   <chr>                            <chr>
## 1 123 4th St S  Appleton, MI 67890 67890
## 2 456 5th St  Appleton, MI 67890   67890
## 3 789 6th St  Appleton, MI 67890   67890
```

13.4 Phone numbers

When you ask a research subject, they can give you a valid phone number in a variety of different ways: (123) 456-7890, 123-456-7890, 1234567890, and so on. This can make working with them complicated. Technically, all we need from a phone number is the numbers. We could use `str_extract()` for that, but to demonstrate another approach, let's use `str_replace_all()` this time. Here, we tell R that we want to replace anything in `phone` that is NOT a digit (the ^ symbol means "not") with `''` (an empty string / nothing).

```
contact_df %>%
  mutate(
    formatted_phone = str_replace_all(phone, '[^0-9]', '')) %>%
  select(phone, formatted_phone)
```

```
## # A tibble: 3 x 2
##   phone          formatted_phone
##   <chr>          <chr>
## 1 314-159-1234   3141591234
## 2 (567) 123-5876 5671235876
## 3 4567890123     4567890123
```

They may not be pretty, but now all of our phone numbers are formatted in exactly the same way. This makes it easy to run subsequent manipulation tasks on them, like checking for duplicates.

13.4.1 What about using `str_extract_all()`

As we mentioned above, there are typically many ways to do the same process with strings, but one or the other often takes more work. In case you are interested, here is the process for extracting a phone number with `str_extract_all()`. The reason it is more complicated in this case is because that function returns a `list()` object, which means you need to wrap it in `map()` to play nice, then unlist and collapse it.

```
contact_df %>%
  mutate(
    formatted_phone = map_chr(
      .x = phone,
      .f = ~ str_extract_all(.x, '[0-9]+') %>%
        unlist() %>%
        paste(collapse = ''))) %>%
  select(phone, formatted_phone)
```

```
## # A tibble: 3 x 2
##   phone          formatted_phone
##   <chr>          <chr>
## 1 314-159-1234   3141591234
## 2 (567) 123-5876 5671235876
## 3 4567890123     4567890123
```

13.5 String interpolation

For basic tasks, the family of `str_...()` functions are great. But what if I want to do something more complicated, like compose a letter.

As a dmeonstration, our contact dataframe includes the date taxes were due, but also the days taxes were paid. What if we wanted to produce a message for everyone that included their last name, as well as the number of days late their taxes were.

Enter `glue` and **string interpolation**. This powerful tool lets us write out our template string, then use `{}` to refer to a variable that will be injected ("interpolated") into that string, like so.

```
library(glue)
```

```
## Warning: package 'glue' was built under R version 4.1.3
```

```
me <- 'Ian'
glue('My name is {me}')
```

```
## My name is Ian
```

We can use this knowledge to create individual messages for each person in our dataframe.

```
contact_df %>%
  mutate(
    tax_date = lubridate::mdy(tax_day),
    days_late = tax_date - as.Date('2022-04-18')) %>%
  filter(days_late > 0) %>%
  mutate(
    message = glue(
      'Mr. {lname}, your taxes are {days_late} days late.')) %>%
  select(lname, message)
```

```
## # A tibble: 2 x 2
##   lname  message
##   <chr>  <glue>
## 1 Cero   Mr. Cero, your taxes are 5 days late.
## 2 McFake Mr. McFake, your taxes are 4 days late.
```

You can also do it with as many variables as you like.

```
data.frame(id = 1:3) %>%
  mutate(
    name = c('Ian', 'Jen', 'Andy'),
    job = c('statistician', 'writer', 'researcher'),
    pronouns = c('He', 'She', 'They'),
    verb = ifelse(pronouns == 'They', 'are', 'is'),
    bio = glue('This is {name}. {pronouns} {verb} a {job}.')) %>%
  select(bio)
```

```
##                                    bio
## 1   This is Ian. He is a statistician.
## 2        This is Jen. She is a writer.
## 3 This is Andy. They are a researcher.
```

Chapter 14

Simulations

If strings are the main feature of R that people didn't realize they needed, simulation techniques are the 2nd. Many people start out saying they are a little intimidated by simulations, but end up feeling comfortable after a little bit of practice.

The initial discomfort usually comes from the fact that we are doing statistics in reverse. In statistical **analysis**, we feed the data to a fitting function (like `lm()`) in order to get parameter estimates. In contrast, in simulations we are studying how well those functions **recover** parameters, so we choose some parameter values ourselves, make data that are consistent with those parameters, then apply our fitting function to see how well it guessed the parameters we planted in our data.

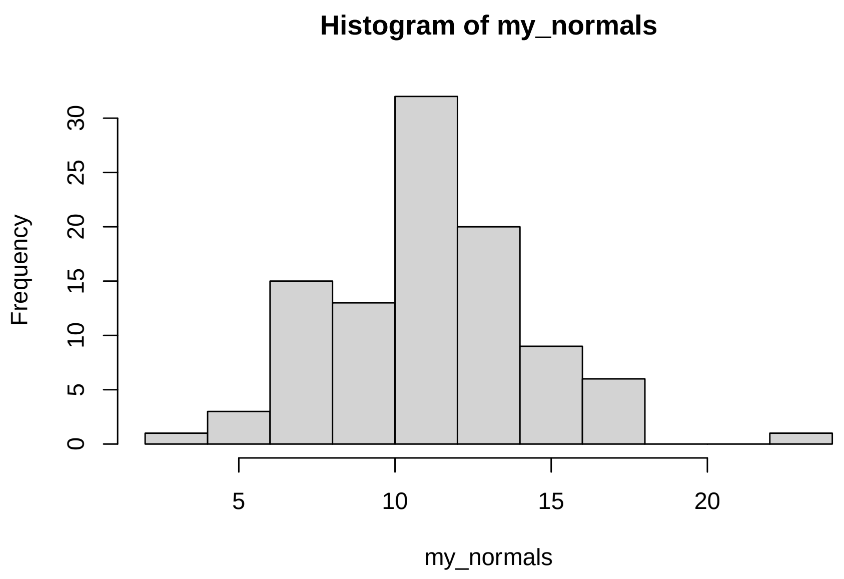
14.1 Generating fake data

To get started, the first thing you need to know is how to generate fake random data. Fortunately, R has many functions for this, which generally start with `r...` for "random".

For example, to make a random normal variable, you would call `rnorm`, which takes 3 arguments:

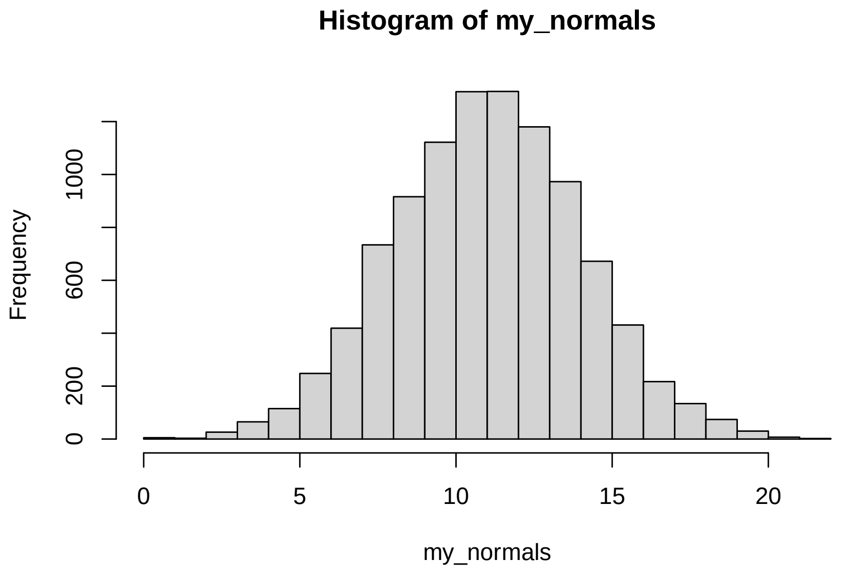
- `n`, the number of numbers you want to generate
- `mean`, the mean of the population you want to be drawing from
- `sd`, the standard deviation of the population you want to be drawing from

```r
my_normals <- rnorm(n = 100, mean = 11, sd = 3)

hist(my_normals)
```



Looking at the graph, we got just about what we would have expected. Will things get even more normal looking with a larger sample size? Definitely.

```
my_normals <- rnorm(n = 10000, mean = 11, sd = 3)

hist(my_normals)
```



14.2 How is this useful?

On their own, random numbers are useless. However, if we use them to make many fake datasets, we can apply the same analysis function to each dataset and keep track of the results. This allows us to ask questions like *how often was an effect significant?* (i.e., power). For example, let's look at the power of a t-test.

R tells us that if we use the formula, the power of a t-test with a true effect size of `d = .30` and a sample size of `n = 150` in each group, we have a power of about .73

```
pwr::pwr.t.test(n = 150, d = .30)
```

```
##
##      Two-sample t test power calculation
##
##               n = 150
##               d = 0.3
##       sig.level = 0.05
##           power = 0.7355674
##     alternative = two.sided
##
## NOTE: n is number in *each* group
```

Let's make our own data and see if that holds when we try to simulate it.

14.3 Re-testing the t-test

To simulate several datasets, we need just a few ingredients. First, we need a dataframe that is set up to represent many datasets. We can use `expand_grid()` for this. With the following commands, it will make 1000 samples, each with two groups (0, 1), and each of those groups will contain 150 people.

```
many_datasets <- expand_grid(
  sample = 1:1000,
  group = 0:1,
  person_id = 1:150)

head(many_datasets)
```

```
## # A tibble: 6 x 3
##   sample group person_id
##    <int> <int>     <int>
## 1      1     0         1
## 2      1     0         2
## 3      1     0         3
## 4      1     0         4
## 5      1     0         5
## 6      1     0         6
```

Now, let's put some information in those groups. We'll give group 0 a mean of 0 and group 1 a mean of .30. Assuming they both have a standard deviation of 1.0, this works out to a group difference of Cohen's d = .30.

```
many_datasets <- many_datasets %>%
  mutate(
    score = ifelse(
      test = group == 0,
      yes = rnorm(n(), mean = 0, sd = 1),
      no = rnorm(n(), mean = .30, sd = 1)))

head(many_datasets)
```

```
## # A tibble: 6 x 4
##   sample group person_id  score
##    <int> <int>     <int>  <dbl>
## 1      1     0         1 -0.986
## 2      1     0         2 -1.00
## 3      1     0         3 -1.77
## 4      1     0         4  0.156
## 5      1     0         5 -1.38
## 6      1     0         6  0.684
```

Now, let's conduct a t-test on each one and look at the results.

```
many_datasets %>%
  group_by(sample) %>%
  nest() %>%
  mutate(
    fit = map(data, ~ t.test(score ~ group, data = .x)),
    results = map(fit, ~ broom::tidy(.x))) %>%
  unnest(results) %>%
  ungroup() %>%
  summarise(power = sum(p.value < .05)/length(p.value))
```

```
## # A tibble: 1 x 1
##   power
##   <dbl>
## 1 0.756
```

Voila! Without doing any complicated calculus, we were able to calculate the power of a t-test - and pretty accurately too!