R for STATA Users

Chris Newton

Contents

1 Introduction **7**

1.1 Who is this book for? 8

1.2 Who isn't this book for? 8

1.3 Setting up R 8

2 Why switch to R **9**

2.1 R advantages 9

2.2 In conclusion. 11

3 Going Through a Project from Start to Finish **13**

3.1 Loading packages 13

3.2 Loading and prepping the data 14

3.3 Visualization 16

3.4 Modeling 17

3.5 Reporting 18

3.6 Exercises 21

4 Dealing with .dta **23**

4.1 Using haven to import STATA files 23

4.2 Dealing with errors 24

4.3 But there's still problems 24

4.4 Other data formats 25

4.5 Using multiple dataframes 26

5 Merging and Wrangling **27**

5.1 base R 27

5.2 dplyr 27

5.3 merge.stats 27

5.4 Into the tidyverse 28

6 One Stop Modeling: Zelig **29**

6.1 Simulation and Counterfactuals 33

7 Visualizing and Describing your Data **37**

7.1 ggplot2 37

7.2 base-R 38

8 Linear Regression **39**

8.1 Which way should you write your model? 40

8.2 Viewing the results 40

9 MLE **43**

9.1 Binary Dependent Variables 43

9.2 Counts 46

9.3 Rare-events and Zero-inflation 47

10 Bayesian Models **49**

10.1 Using STAN in R 49

10.2 McElreath and Rethinking 49

11 Panel Models **51**

11.1 Fixed-Effects 51

11.2 Random-Effects 51

11.3 Mixed Models 51

11.4 Multilevel/Hierarchical Models 51

12 Missing Data **53**

12.1 Amelia II 53

12.2 k-NN Imputation 53

12.3 Machine Learning for Missing Data 53

13 Matching **55**

13.1 The MarchIt Package 55

14 GIS **57**

15 A Brief Introduction to Coding **59**

15.1 Why Learn to Code? 59

15.2 Coding Basics 59

15.3 `ifelse()` 59

15.4 Writing a Function 59

15.5 Loops 59

15.6 Parallelization: What it is, Why it's Awesome, and How to Use it. 59

Chapter 1

Introduction

R for STATA Users is a book designed to take researchers that are already is profecient in STATA, and show them how to do the same analyses in R. The hope is that by starting with what the reader already finds comfrotable and showing how to replicate their code in R, the transition to open source software will be gentle. After covering the basics, the book moves on to more R specific lessons that may introduce techniques that aren't commonly seen in STATA.

This book will show snipets of STATA code, such as:

```
reg dep_var ind_var control_var1 control_var3
```

followed by the corresponding R code to produce the same result. In this case:

```
lm(dep_var ~ ind_var + control_var1 + control_var2,
    data = example.data)
```

There will also be additional answers and explanations. For example, the linear model above can be written differently in R.

```
lm(example.data$dep_var ~ example.data$ind_var +
    example.data$control_var1 + example.data$control_var2)
```

These models are exactly the same. They all run OLS regressions – you can either write the variables by themselves, and then specifly which dataframe they come from, or you can write the dataframe, and `$` symbol, and then the variable name, making sure there are no spaces between them. More on this later.

1.1 Who is this book for?

This book is for anyone that is already profecient in STATA that would like to learn how to conduct statistical analyses in R. It is primarily written for researchers, educations, and students in quatitative fields.

1.2 Who isn't this book for?

This books assumes that you are already comfortable with STATA and statistics. If you don't already know STATA, this book will not make a lot of sense as the underlying concepts will not be explained. If you are not already familiar with the math and intuition behind statistical models, this book will not be of much help eight. No pure math covered what so ever. THere may be some generall discussions on model selection or other topics, but the point of this book is not to explain why you should do something statistically, only to show you how to do something you're already familiar with in STATA using R. This book teaches you how to read in a new language – not how to read.

1.3 Setting up R

There are already a number of great tutorials on how install and setup R. As this book is unlikely to provide an even better tutorial, I recommend checking out one of these tutorials:

- *Hands-On Programming with R*, by Garrett Grolemund: **A Installing R and RStudio**
- *R for Data Science*, by Hadley Wickham & Garrett Grolemund: **1.4 Prerequisites**

Chapter 2

Why switch to R

If you've gotten this far, I'm assuming that you're already profeceint in STATA. Maybe you're a seasoned researcher with scores of publications on you CV. Maybe you're a grad student, recently emerged from the gauntlet of stats class after stats class, having learned STATA along the way. Perhaps you're asking yourself, Why would I re-learn how to do something I already know?

The answer for many of you is: You shouldn't. Some people don't need to learn R, especially considering that they've already learned how to do everything they need to do in STATA. For everyone else, here are some good reasons to make the switch (or at least learn *some* R).

2.1 R advantages

1. **R is free**

R is an open source programing language available for Windows, Mac, and Linux operating systems. This means that anyone can download it, use it, publish results, develop packages, and other fun stuff – without spending any money. Seriously, it's 100% free. 100%. All versions, updates, extra packages, even some books, free. No need for temporary lisences, shelling out for perpetual liscences, buyng new versions, getting your institution to buy it for you, or borrowing that sketchy thumb drive that one person has (not that any reader ever pirated anything). It's all free.

2. **R is free**

Seriously, though. It's free. Even if this doesn't matter to you, if you teach, it likely matters to your students. College is expensive, especially if you go for a really long time, like getting a Ph.D. Even for students with scholarships and funding, the financial burden can be tough. Not having to buy software, on top of buying an overpriced statistics textbook (the latest edition only!), can make a big difference.

3. **R is the preferred language of statisticians and methodologists in many fields**

If you want to be at the forefront of statistics or your chosen field, there's a good chance that the latest developments are going to come in the form of new R packages before they spread to other languages or software. Even before R packages are published, people often post their work on GitHub to be downloaded and used as you will.

4. **R is a programing language**

Getting comfortable in R means learning some fundanamentals of how to write basic code. While this can be extended to developing entire programs in R, learning a bit about functions and loops may mean suffering through a bit of a learning curve, but it will make you life easier down the road. Especially when it come to tedious repetitive tasks, learing a bit about coding can save you lots of time and energy. Learning R means getting comfortable with some of the more basic coding principals.

This also means that learning other programing languages will be easier. All languages employ the same basic logic, with some variation. Understading how one languages works, means it will be a lot easier to learn another. If you want to employ the latest machine learning algorithm, for example, you'll probably need to learn Python. If you're comfortable with the basics, learing a new language is mostly just changing a bit of syntax.

5. **Graphics**

R's libraies for visualization – ggplot2 in particular – can produce everything from publication ready graphs, to maps, to animated 3D simulations. The possibilites are vast with other programing languages building libraries to imitate R's. While a web developer creating visulizations may prefer something such as D3.js, for researchers, it's hard to beat R when it comes to visualizing you data and results.

6. **Boredom**

Sometimes we just want to do things a bit different. Tired of using STATA all the time, why not use R?

7. **To be condescending to your collegues**

- Oh, you use STATA? That's cute. I use R, like a *real* statistician.

2.2 In conclusion. . .

As you can see, some of these reasons are better than others. Maybe they all fit your situation, maybe none do. For those that are commited, lets write some R code.

Chapter 3

Going Through a Project from Start to Finish

To start off, we're going to look at an example analysis. This will go step-by-step through loading data, exploring the dataset, running a regression, and commuicating the results. You may not understand everything and this point, but you'll start to get familiar with R's syntax and won't have to wait 100 pages before trying out something useful. The following chapters will look at each step in detail to explain exactly what is happening, how it relates to STATA commands, and why we're doing it this way.

Note: A great way to learn a programming language is to start with something that works and then break it. Modify a part of the code and see what changes. Did it do what you expected? Did you get an error message? Playing with the code can help you figure out what each component is doing, why it does it that way, and how you can manipulate it to do what you want.

3.1 Loading packages

Once R is installed, you now have what is called base-R. Base-R comes ready to go with a number of statistical functions, visualizations, and even some sample datasets. There is, however, plenty more that can be done by loading additional packages. Packages are developed by the comunity of R users and typically hosted on CRAN (The **C**omprehensive **R** **A**rchive **N**etwork). For effeciency, additional packages have to be installed and then loaded when you want to use them. R doesn't automatically install or load additional packages as this would take up *a lot* of memory with packages that you'll never use.

If a package exists on CRAN, it can be installed by writing `install.packages("package.name")`.

Most packages that you'll want to use will be hosted on CRAN, but occasionally, new packages that are being developed are only on GitHub. If this is the case, the authors will include instructions on how to install the package in the README.md file of the GitHub repository.

You only have to intall a package once, but you have to call it everytime you open R. It's the norm to list all of the packages that you'll be using at the very top of you R script. You call a package with the command

`library(package.name)`

For this example, were going to use the package `modelsummary` for making regression tables, `dplyr` for data manipulation and pipes (`%>%`), which allow us to string commands together, `tidyr` for datat cleaning and wrangling, and `ggplot2` for making graphs. We can import `dplyr`. `tidyr`, and `ggplot2`, by calling `tidyverse`[1], which automatically loads a collection of packages. So starting out, our script should look like this:

```r
library(modelsummary)
library(tidyverse)
```

3.2 Loading and prepping the data

For this example, we are going to use one of the preloaded datasets that comes with R. While you'll never use one of these datasets for actual research, it's easier to use something that everyone already has to get started with an example. The next chapter will show you how to load data in STATA's .dta format, as well as other common formats.

For this example, we'll use the *Titanic* dataset. This will be obnoxiously familiary for anyone that has done some tutiorials on machine learning. For those unfamiliar, it contains variables on the age, gender, and ticket class for those that were on the titanic, as well as whether or not they survived. To access the data, we type `data("Titanic")`. You should now see `Titanic` in the `Enivronment` tab in RStudio. It's not quite ready yet, however, as it is not in a `data.frame` or `tibble` format. If we type `class(Titanic)`, we see that it's `table`. This can be converted with the commands `data.frame(Titanic)` or `as_tibble(Titanic)`

For this example, let's convert the table into a data frame. If you type `data.frame(Titanic)`, the table will be converted to a data frame, and then printed into the console. We don't want this. We want to store the data frame as an object that we can analyze. In R, you store an object by first typing a

[1]For more information on the tidyverse and how to use the various packages, see *R for Data Science*, by Hadley Wickham & Garrett Grolemund.

name of you choosing, followed by the assignment operator (`<-`) and then what you want to be stored as the object. It's best to choose descriptive names for objects, so it's easy to remember what they are. Let's use `titanic.data`. The code should look like this:

```
data("Titanic")
titanic.data <- data.frame(Titanic)
```

You should now see an object called `titanic.data` in you `Environment` with 32 observations or 5 variables. If we want to look at the entire dataset, we can type `View(titanic.data)`. If we only want to see the first few rows, we can type `head(titanic.data)` and the last few rows can be seen with `tail(titanic.data)`. A frequency table can be seen with `table(titanic.data)`. Let's check that out.

```
table(titanic.data)
```

As you can see, it's hard to glean any information from this as frequency is already a variable, with the other variables being collapsed. We can treate a subset without frequency by typing `titanic.subset <- titanic.data[(1:4)]`. This creates a new data frame called `titanic.subset` with only contains the first four columns of the `titanic.data` data frame. Now trying `table(titanic.subset)`, we see eveything has a ferequency of one. To use this data for a regression, let's expand it so that we have one observation for everyone that was on board. We can do this using the `tidyr` function `uncount()`. This function expands the data frame, based on a variable. The syntax is

```
uncount(data, weights, .remote = TRUE, .id = NULL)
```

where `data` is the data frame `weights` is the variable that has the count of rows to duplicate, `.remove` deletes the variable supplied to `weights` (TRUE by defalut) and `.id` creates a new ID for each row. For our data, let's type:

```
titanic.expanded <- uncount(titanic.data, Freq)
```

Now let's explore our expanded data. We already used `View()` to look at the full dataset, but with 2201 observations, it can be hard to tell much about what going on. Instead, we're going to generate summary statistics with `summary(titanic.expanded)`. This shows of the level of each variable, the number of ovservations at the level, and, implicitly, that there are no missing values. If there were missing values, the last row of each variable would read `NA's:` followed by the number of rows for which that variable didn't have a value.

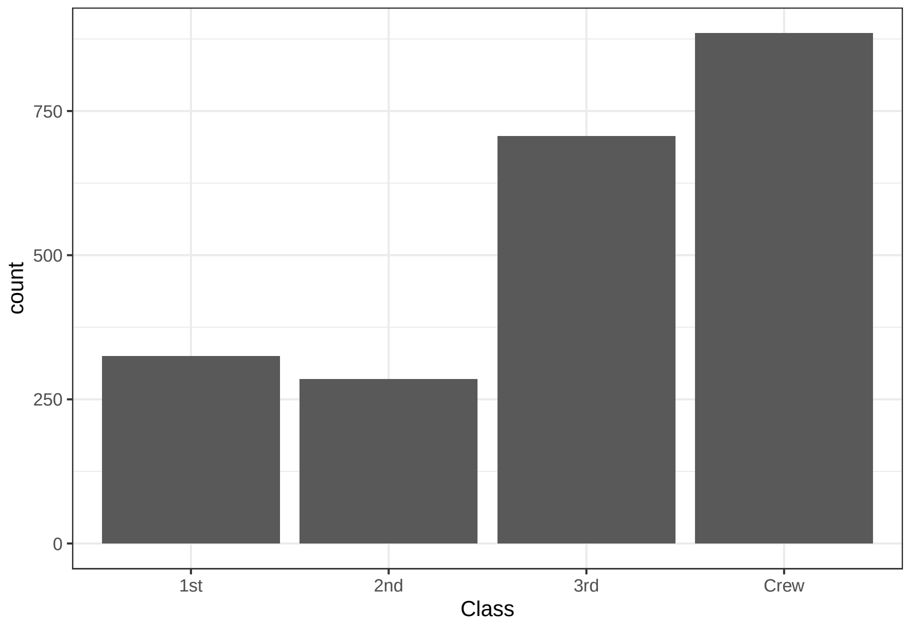
Now that we have a data frame that we can analyze, we no longer need the original data, or the subset we created. We can get rid of these with `rm(list = c('Titanic', 'titanic.data', 'titanic.subset'))`. R can use a lot of memory on your computer, so it's best to get rid of any objects that you're no longer using.

3.3 Visualization

Let's look at our data using some plots. First, we're going to check the distribution of our variables. Given that all of our variables are factors, a histogram is the was to go. Using `ggplot2`, we can do this:

```
ggplot() +
    geom_histogram(data = titanic.expanded,
        aes(x = Class), stat = 'count') +
    theme_bw()
```

```
## Warning: Ignoring unknown parameters: binwidth, bins, pad
```

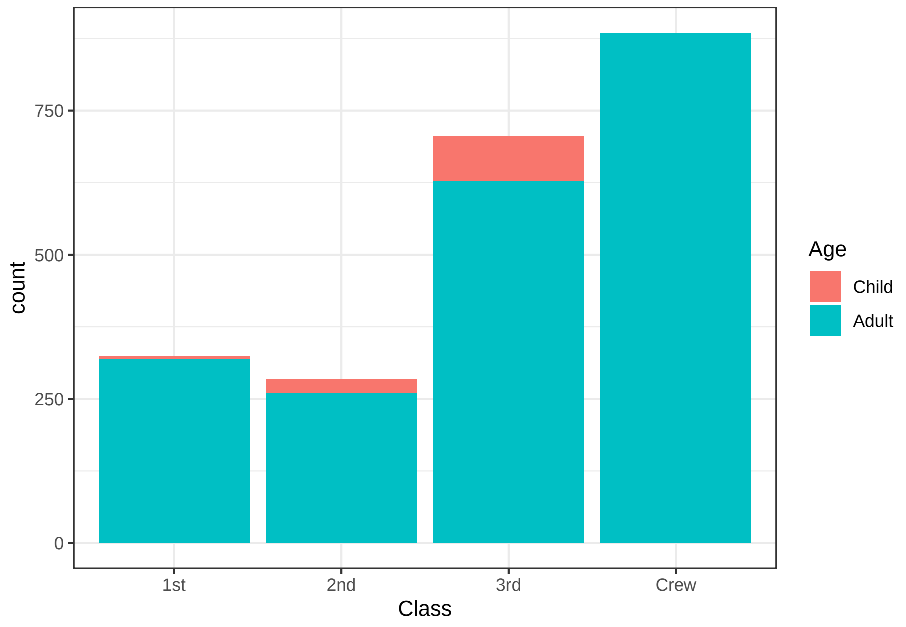


The above calls a plot (`ggplot()`) and then says that we're going to make a histogram (`geom_histogram()`). We're going to use the `titanic.expanded` data, and we want to see the variable `Class`. `aes()` is responsible for creating the mapping, in other words, with the variables that are being plotted. We include `stat = 'count'` as we're looking at the frequenqy of each level of the variable `Class`. Finally, `theme_bw()` styles the graph. This part is optional, and there are plenty of other themes you can choose from, including custom themes that you can make yourself. `ggplot2` uses the *grammar of graphics* which layers different aspects of a visualiztion on top of each other. Each layer is connected with a `+`. While you could keep everything on one line and the code will still run, it is best to end each line with a `+` and the start on the next line with an indent. This keeps the code organized and easy to read.

Now say you also wanted to show how many within each class were chilren and how many were adults. This could be done by changing the `fill`.

```
ggplot() +
    geom_histogram(data = titanic.expanded,
        aes(x = Class, fill = Age), stat = 'count') +
    theme_bw()
```

```
## Warning: Ignoring unknown parameters: binwidth, bins, pad
```



3.4 Modeling

We're going to build a model to predict whether or not someone would survive based on the variables we have. `Survived` is a binary variable, so we'll estimate a logit model.

```
titanic.logit <- glm(Survived ~ Class + Sex + Age,
    data = titanic.expanded, family = 'binomial')
summary(titanic.logit)
```

```
##
## Call:
## glm(formula = Survived ~ Class + Sex + Age, family = "binomial",
##     data = titanic.expanded)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.0812  -0.7149  -0.6656   0.6858   2.1278
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)   0.6853     0.2730   2.510   0.0121 *
## Class2nd     -1.0181     0.1960  -5.194 2.05e-07 ***
## Class3rd     -1.7778     0.1716 -10.362  < 2e-16 ***
## ClassCrew    -0.8577     0.1573  -5.451 5.00e-08 ***
## SexFemale     2.4201     0.1404  17.236  < 2e-16 ***
## AgeAdult     -1.0615     0.2440  -4.350 1.36e-05 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 2769.5  on 2200  degrees of freedom
## Residual deviance: 2210.1  on 2195  degrees of freedom
## AIC: 2222.1
##
## Number of Fisher Scoring iterations: 4
```

Unpacking the above command, `glm()` calls a generalized linear model, with `Survived` as the dependent variable, and `Class`, `Sex`, and `Age`, as inependent variables, using the `titanic.expanded` data frame. `family = 'binomial'` declares that the model is a logit, and we save this as an object called `titanic.logit`. The `summary()` command gives us the statistical information we want to know about the model.

3.5 Reporting

Now have results, we need to communicate them. Let's start with a nice table. Typing `modelsummary(titanic.logit, stars = TRUE)` gives us a basic table, but the variable names aren't formated nicely. We can change this by creating an object with new names, and adding `coef_map = independent.var.names` to `modelsummary()`:

```
independent.var.names = c(
    'Class2nd' = 'Second Class',
    'Class3rd' = 'Third Class',
    'ClassCrew' = 'Crew',
    'SexFemale' = 'Sex (Female)',
    'AgeAdult' = 'Age (Adult)'
)

modelsummary(titanic.logit, stars = TRUE,
    coef_map = independent.var.names)
```

	Model 1
Second Class	-1.018***
	(0.196)
Third Class	-1.778***
	(0.172)
Crew	-0.858***
	(0.157)
Sex (Female)	2.420***
	(0.140)
Age (Adult)	-1.062***
	(0.244)
Num.Obs.	2201
AIC	2222.1
BIC	2256.2
Log.Lik.	-1105.031

* p < 0.1, ** p < 0.05, *** p < 0.01

And say we have multiple models, such as one for each independent variable plus our original model, we can report all of them like this:

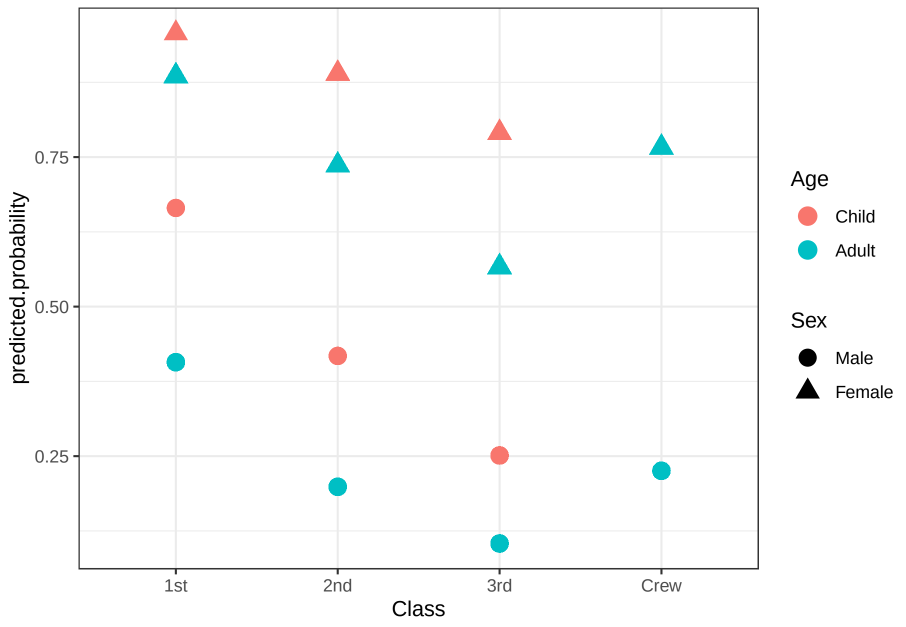
```
models = list(
    `Class` = glm(Survived ~ Class,
        data = titanic.expanded, family = 'binomial'),
    `Sex` = glm(Survived ~ Sex,
        data = titanic.expanded, family = 'binomial'),
    `Age` = glm(Survived ~ Age,
        data = titanic.expanded, family = 'binomial'),
    `All` = glm(Survived ~ Class + Sex + Age,
        data = titanic.expanded, family = 'binomial')
)

independent.var.names = c(
    'Class2nd' = 'Second Class',
    'Class3rd' = 'Third Class',
    'ClassCrew' = 'Crew',
    'SexFemale' = 'Sex (Female)',
    'AgeAdult' = 'Age (Adult)'
)

modelsummary(models, stars = TRUE,
    coef_map = independent.var.names)
```

	Class	Sex	Age	All
Second Class	-0.856***			-1.018***
	(0.166)			(0.196)
Third Class	-1.596***			-1.778***
	(0.144)			(0.172)
Crew	-1.664***			-0.858***
	(0.139)			(0.157)
Sex (Female)		2.317***		2.420***
		(0.120)		(0.140)
Age (Adult)			-0.880***	-1.062***
			(0.197)	(0.244)
Num.Obs.	2201	2201	2201	2201
AIC	2596.6	2339.0	2753.9	2222.1
BIC	2619.3	2350.4	2765.3	2256.2
Log.Lik.	-1294.278	-1167.494	-1374.948	-1105.031

* $p < 0.1$, ** $p < 0.05$, *** $p < 0.01$

We can graph our results using `ggplot2`, but first we need to calculate the predicted probailities.

```
titanic.predictions <- cbind(titanic.expanded,
    predict(titanic.logit, newdata = titanic.expanded,
        type = 'link', se = TRUE))

titanic.predictions <- within(titanic.predictions, {
    predicted.probability <- plogis(fit)
    }
)

ggplot(titanic.predictions, aes(Class, predicted.probability)) +
    geom_point(aes(color = Age, shape = Sex), size = 4) +
    theme_bw()
```



And there you have it. A complete project from start to finish in R. There are of course plently of other things we could have done, but this chapter is about getting a taste for R. In future chapters we'll go much futher in depth to each step and still only cover a portion of what's possible in R. Some of this code may not make sense yet – that's okay. As the book goes on, we'll get into the detail about why we do diffent things. The important part in this chapter is running the code and seeing what it produces.

3.6 Exercises

1. We used `geom_histogram()` to look as the frequency of `Class` as well as the break down of `Age` within `Class`. Try to make more graphs that similarly describe the other variables.

2. Using `fill =`, we're able to change the color of the bars. We specified this within `aes()`. What happens if you move `fill =` outside of `aes()`?

3. As mentioned above, one of the best ways to learn a programing language is to start with a script that works, and then play with the commands

until you break it. Once broken, figue out what went wrong, and try to understand both why it didn't work and what should be done instead. If you got this script to run from beginning to end, try to break it. Import your own data set, explore different variables, and run different models. When domething doesn't work, try to see how it's different from this script, why that won't work, and discover what will.

Chapter 4

Dealing with .dta

There are multiple packages that can read and write .dta files; we're going to use `haven`. Haven is not part of base-R, so it has to be installed if you haven't done so before. With it installed, put `library(haven)` at the top of you R script.

```r
library(haven)
```

4.1 Using haven to import STATA files

Now you import your data. This is done with the command `read_dta()`. As you probably want to actually load the data into the environment, and not just print the observations in the console, you'll have to assign the data a name. In R, names can contain uppercase and lowercase letters, numbers, underscores, and periods. An object name cannot contain spaces, begin with a number, or contain symbols such as $ or %. Also, names cannot be the same as a function in base-R or any of the packages you are using.

```
my.data <- read_dta('data.dta') # this works
MyDaTa <- read_dta('data.dta') # so does this

# this doesn't work, sum() is a function in base-R
 sum <- read_dta('data.dta')

data2 <- read_dta('data.dat') # this is fine
2data <- read_dta('data.dat') # this isn't
```

Names are assigned by using `<-`, typically with a space before and after (though this isn't necessary, it keeps the code clean and easy to read). To load your data, give it a name (that conforms with R's rules) followed by `<-` and then `read_dta()`. If your data is in your working directory, you can simply write the file name inside either single (") or double ("") quotes. To find you working directory, type `getwd()` into the console.

If your file is in a subdirectory of your working directory, you can simply specify the subdirectiory. For example, if you keep all of your datasets in a folder 'data', and you have a file `data.dta`, you would type `read_dta('data/data.dta')`. If your data is outside of your working directory, you can specify the complete file path. for example `read_dta('~/home/user/Desktop/datasets/data.dta')`.

4.2 Dealing with errors

The `read_dta()` function supports STATA versions 8-15. If you import your file and it doesn't look right, there may be an issue with interpreting the version. This can be fixed by adding a comma after the file name, followed by `version =` and then the version of STATA that wrote the file. For example, importing a file from STATA 10 would look like this:

```
stata.data <- read_dta('data/data.dta', version = 10)
```

If there's still an issue, it might be the econding. Before STATA 14, files relied on the default encoding of the system when writing a file. This means that a file written on Windows may not have the same encoding as one written on Mac or Linux. If you get the message `"Unable to convert string to the requested encoding"`, it's probably because STATA saved the default Windows encoding, windows-1252. To fix this, add `encoding = "latin1"` after the version (again seperated with a comma).

```
stata.data <- read_dta('data/data.dta', version = 10,
                        encoding = "latin1")
```

Of course, if you saved the file on you own computer or the file was save using STATA 14 or newer, this shouldn't be a problem.

4.3 But there's still problems

If you're still having errors at this point, the best option is probably to quit. Unistall R, throw your laptop into the sea, fish it our because you're worried about pollution, chuck it in rice because you realize that you started learning R to save money.

Or, start practicing th single most important programming skill there is: looking up the answer on the internet. When something doesn't run, R prints an error message. Copy and paste this error message into the search engine of your choice, and it's likely that someone has already had the same issue, posted about it on Stack Overflow or GitHub, and found a solution.

4.4 Other data formats

While `.dta` may be the most common format for data files if you use STATA, there are plenty of other formats out there that you'll run into.

4.4.1 .Rdata

`.Rdata` is the simplest format to load as it is R's native data format. You simply type `load()` with with file name if the data is in your working directory. Just as with `read_dta()`, if the data isn't in your working directory, you have to specify either the subdirectory or the complete file path. Note, `load()` will import you data with the file name preceding `.Rdata` being the name of the data frame, so `example_data.Rdata` will become an object named `example_data`. You can of course change the name, by writing:

```
new_name <- example_data
rm(example_data)
```

4.4.2 .csv and other delimited files

One of the most common ways of saving data is with delimited text files. The `readr` package, which is part of the `tidyverse`, comes with three functions for different seperators, as well as a generic delimited file importer. `read_csv()` imports comma seperated files, `read_csv2()` imports semicolon seperated files, and `read_tsv()` imports tab seperated files. `read_delim()` handles everying else as you can specify which the delimiter is. You do this with `delim = 'delimiter'`. For example:

```
# * delimited file
delimited.data <- read_delim(example_data.txt, delim = ’*’)
```

You can also read delimited files using base-R functions. For example, comma seperated files can be read with `read.csv()`. While using base-R eliminates the need to import a package, the readr functions run more quickly, and are therefore better if you have a large dataset.

4.4.3 Excel files

To read Excel files saved as `.xls` or `.xlsx`, use the package `readxl`. For `.xls` files, the command is `read_xls()`, and for `.xlsx` files, it's `read_xlsx()`.

4.4.4 .json

For `.json` files, the package `jsonlite` is used. To import a dataset, use `fromJSON()`.

4.5 Using multiple dataframes

One advantage of R over STATA is that you can have multiple dataframes loaded into your environment. If you have one main dataset, `data.dta`, you can import it, and then divide it into subsets or variations. If you are running four models, each of which is using a different sample or transformed data, you can create four dataframes (ex: `data1, data2, data3, data4`) and then use each dataframe from each model. If you then want to remove one that you're no longer using, you can do so with `rm()`.

This can be very usefull when merging. You can import your main dataset and the data you are going to merge in to compare. You can then save the merged data as a third dataframe to compare with the originals to make sure everything looks as it should.

Chapter 5

Merging and Wrangling

5.1 base R

```
merged_data <- merge(data1, data2, by = c("time", "space"), all = TRUE)
```

5.2 dplyr

```
joined_data <- inner_join(data1, data2, by ("time" = "time", "space" = "space"))
joined_data <- left_join(data1, data2, by ("time" = "time", "space" = "space"))
joined_data <- full_join(data1, data2, by ("time" = "time", "space" = "space"))
```

5.3 merge.stats

If you're used to merging in STATA, you'll probably miss the `_merge` column, which nicely summarizes how year observation merged (or didn't). To replicate this, I created the `merge.stats` package. This package is currently in devlopment, but it can be installed from GitHub and tried out by running

```
devtools::install_github("newton-c/merge_stats_R")
```

This package has two commands, `merge_stats()` and `join_stats()`. Both packages add a new column, `merge` to the merged dataframe, as well as printing statistics, such as how many observations from each dataframes did and did not successfully merge. `merge_stats()` is build on top of the base R `merge()` function and takes all of the same parameters. In addition, you can specify `show.stats = TRUE` to print the statics of the merge, or `show.stats = FALSE` if you want to cut down on how much is being printed to the console. `merge_join` is built on top of the various `_join()` functions from dyplr. This function has two additional arguements, `show_stats =` which says whether to print the statistics of the join, and `join =` which specifies wither the joint is `"inner"`, `"right"`, `"left"`, `"full"`, `"semi"`, or `"anti"`.

5.4 Into the tidyverse

5.4.1 `filter()`

5.4.2 `mutate()`

5.4.3 `group_by()`

5.4.4 `select()`

5.4.5 `%>%`

5.4.6 Stringing it Together

Chapter 6

One Stop Modeling: Zelig

Zelig is a statistical software originally created by Kosuke Imai, Gary King, and Olivia Lau. It attampts to unify different models, tests, vizualizations, and other statistical activities into a single framework. R is not a piece of software, it is an open source programing language. As such, as it has adapted and grown, different people have created different packages. This is a strength as it keeps R flexible and on the cutting-edge of statistics, but it means that there is often inconsistency. There are variaous packages in R that you can use for different models; Zelig was created to keep the syntax consistent and make using different models as simple as changind a single command. More information on Zelig can be found on its website.

In addition to running your typical statistical models, Zelig also replicates Clarify from STATA, integrates with multiple imputation methods (through the package `Amelia`) as well as matching methods (through the packages `cem` and `MatchIt`)

The generic syntax for statistical models in Zelig is as follows

```
zelig(y ~ x1 + x2 + x3, data = example.data, model = "model.type")
```

This syntax is very similar to most other models, but with the added arguement `model =` in order to specifiy what you're running. If we wanted to replicate the model we estimated in Chapter 3, this is how it'd look.

```r
library(tidyverse)
library(Zelig)

theme_set(theme_classic())

data("Titanic")
```

```r
titanic.data <- data.frame(Titanic)
titanic.expanded <- uncount(titanic.data, Freq)

titanic.logit <- zelig(Survived ~ Class + Sex + Age,
    data = titanic.expanded, model = "logit")
```

```
## How to cite this model in Zelig:
##   R Core Team. 2007.
##   logit: Logistic Regression for Dichotomous Dependent Variables
##   in Christine Choirat, Christopher Gandrud, James Honaker, Kosuke Imai, Gary King,
##   "Zelig: Everyone's Statistical Software," http://zeligproject.org/
```

As you can see, Zelig automatically prints the citation for the model your using. This can be very useful when preparing your references for a project, but very annoying when you're running a bunch of models. Adding `cite = FALSE` (or `cite = F`) will stop this.

```r
summary(titanic.logit)
```

```
## Model:
##
## Call:
## z5$zelig(formula = Survived ~ Class + Sex + Age, data = titanic.expanded)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.0812  -0.7149  -0.6656   0.6858   2.1278
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)   0.6853     0.2730   2.510   0.0121
## Class2nd     -1.0181     0.1960  -5.194 2.05e-07
## Class3rd     -1.7778     0.1716 -10.362  < 2e-16
## ClassCrew    -0.8577     0.1573  -5.451 5.00e-08
## SexFemale     2.4201     0.1404  17.236  < 2e-16
## AgeAdult     -1.0615     0.2440  -4.350 1.36e-05
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 2769.5  on 2200  degrees of freedom
## Residual deviance: 2210.1  on 2195  degrees of freedom
## AIC: 2222.1
##
## Number of Fisher Scoring iterations: 4
```

```
##
## Next step: Use 'setx' method
```

Zelig models can be easily exported to a table for publication-ready results, but the require an additional command, `from_zelig_model()`. We can compare the original logit from Chapter 3 to the one above to varify that they're the same.[1]

```
library(modelsummary)

models = list(
    "Ch 3 model" = glm(Survived ~ Class + Sex + Age,
        data = titanic.expanded, family = 'binomial'),

    # using titanic.logit we just estimated above
    "Zelig model" = zelig(Survived ~ Class + Sex + Age,
    data = titanic.expanded, model = "logit", cite = FALSE) %>%
        from_zelig_model()
)

modelsummary(models, stars = TRUE)
```

[1]There are two options for converting Zelig's output to a format that's readable for `modelsummary()`. As seen above you can add a pipe (`%>%`) followed by `from_zelig_model()`. Alternatively, you can wrap the code inside of `from_zelig_model()`. For example `from_zelig_model(zelig(Survived ~ Class + Sex + Age, data = titanic.expanded, model = "logit"))`. Both methods do the same thing, the difference is ultimately asthetic. Choose the method you find most readable.

	Ch 3 model	Zelig model
(Intercept)	0.685**	0.685**
	(0.273)	(0.273)
Class2nd	-1.018***	-1.018***
	(0.196)	(0.196)
Class3rd	-1.778***	-1.778***
	(0.172)	(0.172)
ClassCrew	-0.858***	-0.858***
	(0.157)	(0.157)
SexFemale	2.420***	2.420***
	(0.140)	(0.140)
AgeAdult	-1.062***	-1.062***
	(0.244)	(0.244)
Num.Obs.	2201	2201
AIC	2222.1	2222.1
BIC	2256.2	2256.2
Log.Lik.	-1105.031	-1105.031

* $p < 0.1$, ** $p < 0.05$, *** $p < 0.01$

As we can see, the output is identical, proving that either method will work. Now we can look at various different models by simply changing the `model =` parpameter. Here's a compariason of a logit, and probit.

```r
models = list(
    "Logit" = zelig(Survived ~ Class + Sex + Age,
                    data = titanic.expanded,
                    model = "logit", cite = F) %>%
        from_zelig_model(),
    "Probit" = zelig(Survived ~ Class + Sex + Age,
                     data = titanic.expanded,
                     model = "probit", cite = F) %>%
        from_zelig_model()
)

modelsummary(models, stars = TRUE)
```

	Logit	Probit
(Intercept)	0.685**	0.367**
	(0.273)	(0.161)
Class2nd	-1.018***	-0.630***
	(0.196)	(0.114)
Class3rd	-1.778***	-1.027***
	(0.172)	(0.098)
ClassCrew	-0.858***	-0.540***
	(0.157)	(0.094)
SexFemale	2.420***	1.450***
	(0.140)	(0.080)
AgeAdult	-1.062***	-0.580***
	(0.244)	(0.141)
Num.Obs.	2201	2201
AIC	2222.1	2224.6
BIC	2256.2	2258.8
Log.Lik.	-1105.031	-1106.314

* $p < 0.1$, ** $p < 0.05$, *** $p < 0.01$

You can see all of the supported models, and their specific syntax in the Zelig documentation.

6.1 Simulation and Counterfactuals

Zelig can take a model and generate simulated values. This can be particularly useful for plotting counterfactuals. We'll use the `mtcars` dataset. This is a dataset of various car features for 32 models, from a Motor Trend magazine's 1974 issue.

```
data("mtcars")
head(mtcars)
```

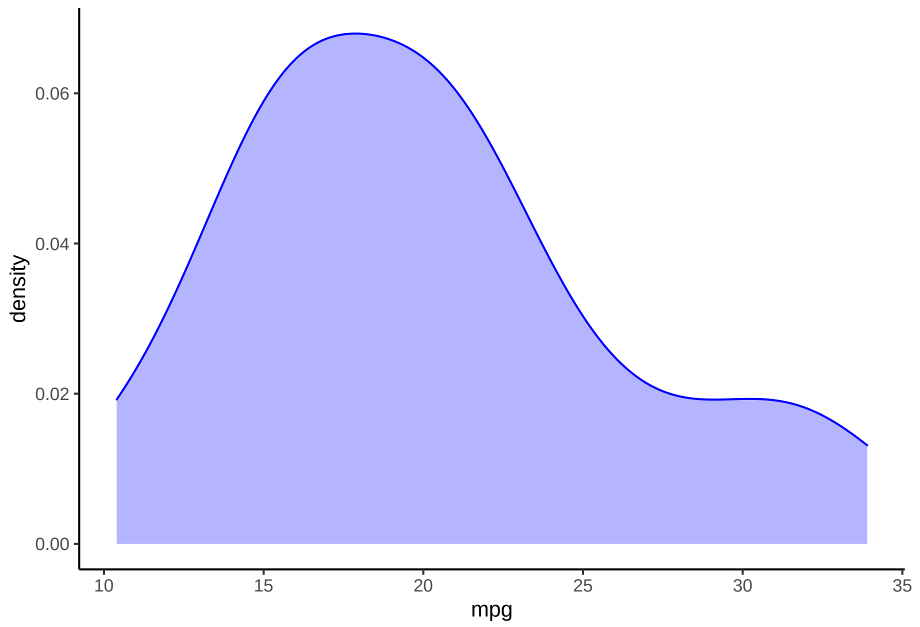
```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet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

Let's say we're interested in the effect the that number of cylinders has on a car's fuel effeciency. We'll start by looking at the data.

```
table(mtcars$cyl)
```

```
##
##  4  6  8
## 11  7 14
```

```
ggplot(mtcars, aes(x = mpg)) +
    geom_density(fill = "blue", color = "blue", alpha = .3)
```



To look at the effect of cylinders *all else equal*, we need to estimate a model. Miles per gallon is roughly countious and regularly distributed, so we'll use OLS. In addition to the number of cylinders lets adjust for the weight, horsepower, and the number of gears a car has.

```
m1 <- zelig(mpg ~ cyl + wt + hp + gear, data = mtcars,
            model = "ls", cite = FALSE)
summary(m1)
```

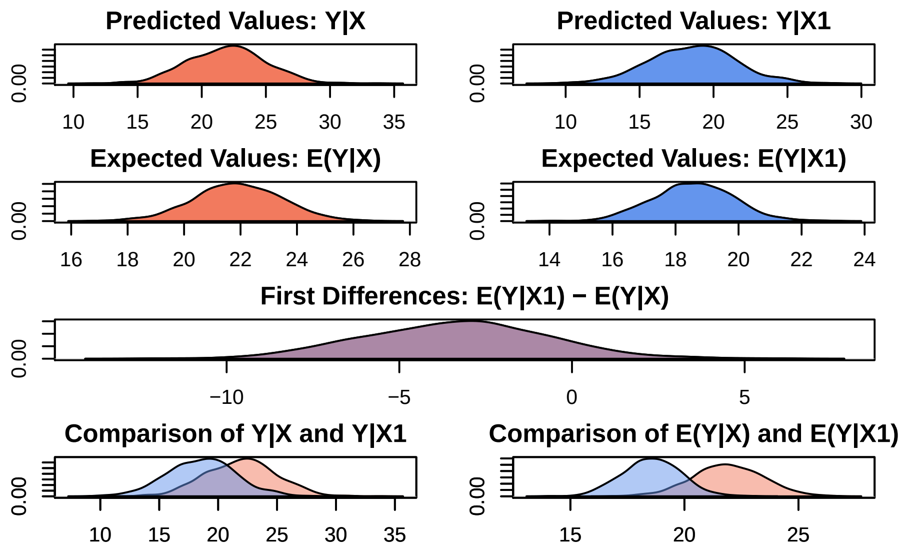
```
## Model:
##
## Call:
## z5$zelig(formula = mpg ~ cyl + wt + hp + gear, data = mtcars)
##
```

```
## Residuals:
##     Min      1Q  Median      3Q     Max 
## -3.4710 -1.7876 -0.6517  1.2362  5.9677 
## 
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 36.68953    5.97025   6.145 1.44e-06
## cyl         -0.81260    0.66320  -1.225  0.23106
## wt          -3.02263    0.85116  -3.551  0.00143
## hp          -0.02170    0.01574  -1.379  0.17922
## gear         0.36259    1.00000   0.363  0.71974
## 
## Residual standard error: 2.551 on 27 degrees of freedom
## Multiple R-squared:  0.8439, Adjusted R-squared:  0.8208 
## F-statistic: 36.49 on 4 and 27 DF,  p-value: 1.599e-10
## 
## Next step: Use 'setx' method
```

The table shows that adding a cylinder to the engine decreases the average miles per gallon by 0.8. Now we can simulate some values and visualize the relationship.

First we have to set the values of interest. Let's see a 4 cylinder vs 8 cylinder car. This is done with the `setx()` and `setx1()` commands. We then simulate values with `sim()`, and finally plot the results with `plot()`. This can all be chained together with `%>%`.

```r
par(mar = rep(2, 4))

zelig(mpg ~ cyl + wt + hp + gear, data = mtcars,
            model = "ls", cite = FALSE) %>%
    setx(cyl = 4) %>%
    setx1(cyl = 8) %>%
    sim() %>%
    plot()
```



We can see from the graphs that a car with 8 cylinders (in blue) is expected to have a lower mgp when compared to a car with 4 cylinders. That said, there is some overlap in the comparison.

Chapter 7

Visualizing and Describing your Data

7.1 ggplot2

7.1.1 Destriptive visualitation

7.1.1.1 Box and Whisker Plots

7.1.1.2 Histograms

7.1.1.3 Density Plots

7.1.1.3.1 Stacking Plots: Histogram and Density in one Frame

7.1.1.4 Missing Data

7.1.2 Communicating Results

7.1.2.1 Dot Plots

7.1.2.1.1 Plotting Regression over the Data

7.1.3 Other Vizualizations

7.1.3.1 GIS Maps

7.1.3.2 Animation

7.2 base-R

Chapter 8

Linear Regression

Now we'll look at plain old OLS. In STATA this is done using the command `reg`. In R, OLS is run using the command `lm()`. This is part of base-R, so there are no packages to install, you just start R and you're ready to go.

While STATA seperates each part of the regression with a space, R wraps everything in parentheses. You don't have to include any spaces between the elements inside the pathenses, but it best to do so, for readability. You do have to add `~`, `+`, and `:`. `~` goes between the dependent variabale, and the rest of the equation, `+` seperates the rest of the variables in an additive model, and `:` indicates a multiplitcative interation. So

```
lm(y ~ x1 + x2 + x1:x2, data = stata.data)
```

in R is the same as

```
gen x1x2 = x1*x2
reg y x1 x2 x1x2
```

in STATA.

This R code could also be written:

```
lm(stata.data$y ~ stata.data$x1 + stata.data$x2 +
    stata.data$x1:stata.data$x2)
```

And just to add even more variety, one could also write:

```
attach(stata.data)
lm(y ~ x1 + x2 + x1:x2)
```

8.1 Which way should you write your model?

Starting with `attach(stata.data)` is likely to be most comfortable for STATA users. This method loads a single dataframe into the environment (in this case stata.data) and now any variable you reference is assumed to belong to that dataframe. If you call a variable that doesn't exist in the dataframe, you see the message `Error in eval(predvars, data, env) : object 'variable' not found` where `'variable'` is that name of the non-existant variable you tried to call.

If you want to switch to another dataframe, you simply write `detach(stata.data)` and then attach another dataframe (i.e. `attach(stata.data2)`). Note that `detach()` will not remove the data from the environment, it only removes it from being the default for calling variables. If you want to remove the data completly, type `rm(stata.data)`.

While this may be the most similar to the way you're used to working with dataframes in STATA, one of the advantages of R is that you can work with various different dataframes at the same time. You can alsways indicate which dataframe a variable belongs to by writeing `dataframe$variable`. This is seen in the second example above and is quite specific, but requires specifying the dataframe for every variable. This can be tedious. Adding `data = dataframe` is a nice balance where you only have to specify the data once per model, but you can still access different dataframes without constantly typing `detach()` and `attach()`.

8.2 Viewing the results

If you simply run the `lm()` command without assigning it to an object, the results will print in the console. If you do assign you model to an object, you can access the results with the `summary()` command. Here's an example using the `ToothGrowth` dataset, which examines the role of vitimins in the rate of guinea pigs' tooth growth.

```r
data("ToothGrowth")
dat <- data.frame(ToothGrowth)

model <- lm(len ~ supp + dose, data = dat)
summary(model)
```

```
##
## Call:
## lm(formula = len ~ supp + dose, data = dat)
##
```

```
## Residuals:
##    Min     1Q Median     3Q    Max
## -6.600 -3.700  0.373  2.116  8.800
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   9.2725     1.2824   7.231 1.31e-09 ***
## suppVC       -3.7000     1.0936  -3.383   0.0013 **
## dose          9.7636     0.8768  11.135 6.31e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 4.236 on 57 degrees of freedom
## Multiple R-squared:  0.7038, Adjusted R-squared:  0.6934
## F-statistic: 67.72 on 2 and 57 DF,  p-value: 8.716e-16
```

You can combine the results of numerous models into a single (publication ready) table as we've seen before.

```r
library(modelsummary)

cn = c(
    "suppVC" = "Supplement Type",
    "dose" = "Dose (mg/day)"
)

modelsummary(models = model, coef_map = cn, stars = TRUE)
```

	Model 1
Supplement Type	-3.700***
	(1.094)
Dose (mg/day)	9.764***
	(0.877)
Num.Obs.	60
R2	0.704
R2 Adj.	0.693
AIC	348.4
BIC	356.8
Log.Lik.	-170.208
F	67.718

* $p < 0.1$, ** $p < 0.05$, *** $p < 0.01$

Chapter 9

MLE

9.1 Binary Dependent Variables

While STATA has seperate commands for different MLE models (`logit`, `nbreg`, etc.), R combines some models into single commands. We can use `zelig()`, the command we learned earlier, and just change the `model =` portion. Alternatively, there are commands such as `glm()`, which do the same thing outside of the Zeligverse. We'll loog at some examples with the `iris` dataset.

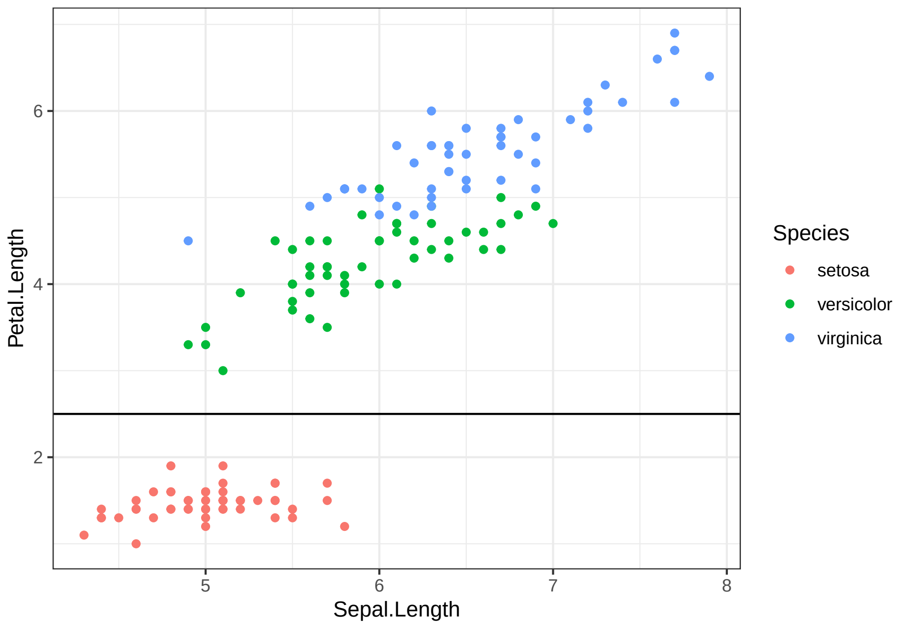
```
library(modelsummary)
library(tidyverse)
library(Zelig)

head(iris)
```

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

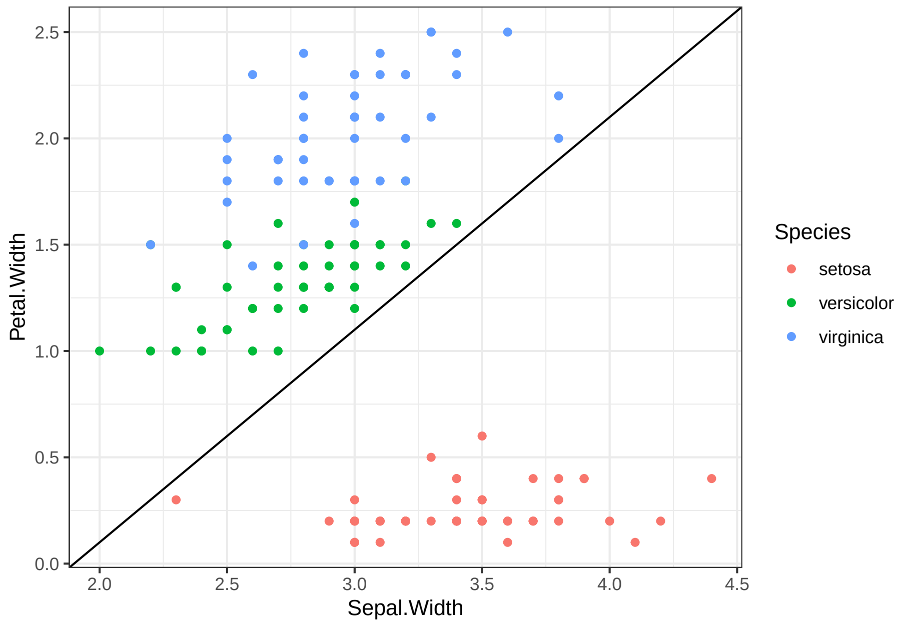
```
summary(iris)
```

```
##   Sepal.Length    Sepal.Width     Petal.Length    Petal.Width   
##  Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100  
##  1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300  
##  Median :5.800   Median :3.000   Median :4.350   Median :1.300
##  Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
##  3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
##  Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
##        Species
##  setosa    :50
##  versicolor:50
##  virginica :50
##
##
##
```

```
ggplot(iris) +
    geom_point(aes(x = Sepal.Length, y = Petal.Length, color = Species)) +
    geom_hline(yintercept = 2.5) +
    theme_bw()
```



```
ggplot(iris) +
    geom_point(aes(x = Sepal.Width, y = Petal.Width, color = Species)) +
    geom_abline(intercept = -1.9) +
    theme_bw()
```



This dataset, originally collected by Ronald Fisher, looks at three different species of iris: setosa, versicolor, and virginica. It provides information on the length and width of flowers' petal and sepals. Looking at the data, we can see that setosas are rather distinct, and easy to seperate graphically. Veriscolor and virginia are more similar, so we'll examine them statistically. We can create a new dataframe with the `filter()` command we learned in *Modeling and Wrangling*.

```
iris.binary <- filter(iris, Species != "setosa")
```

To predict whether a flower is setosa or virginica, we could use a logit model.

```
logit Species Sepal.Length Sepal.Width Petal.Length Petal.Width
```

In R, we estimate a logit by specifying `model = "logit"` in `zelig()`.

```
iris.logit <- zelig(Species ~ Sepal.Length + Sepal.Width + Petal.Length +
                    Petal.Width, data = iris.binary,
                    model = "logit", cite = FALSE)
```

IF we want to instead estimate a probit model, in STATA, we change the command.

```
probit Species Sepal.Length Sepal.Width Petal.Length Petal.Width
```

In R, we change `model =`.

```
iris.probit <- zelig(Species ~ Sepal.Length + Sepal.Width + Petal.Length +
                     Petal.Width, data = iris.binary,
                     model = "probit", cite = FALSE)
```

```
models = list(
    `Logit` = from_zelig_model(iris.logit),

    `Probit` = from_zelig_model(iris.probit)
)

modelsummary(models = models, stars = TRUE)
```

	Logit	Probit
(Intercept)	-42.638*	-23.985*
	(25.707)	(13.843)
Sepal.Length	-2.465	-1.440
	(2.394)	(1.272)
Sepal.Width	-6.681	-3.778
	(4.480)	(2.556)
Petal.Length	9.429**	5.316**
	(4.737)	(2.435)
Petal.Width	18.286*	10.486*
	(9.743)	(5.614)
Num.Obs.	100	100
AIC	21.9	21.8
BIC	34.9	34.8
Log.Lik.	-5.949	-5.876

* $p < 0.1$, ** $p < 0.05$, *** $p < 0.01$

9.2 Counts

Count models tend to fall into two categories: Poisson and negative binomial. Poisson models assume an even dispersion, with the mean equal to the variance, while negative binomial accout for overdispersed data.

```
summary(diamonds)
```

```
##      carat               cut          color        clarity          depth
##  Min.   :0.2000   Fair     : 1610   D: 6775   SI1    :13065   Min.   :43.00
##  1st Qu.:0.4000   Good     : 4906   E: 9797   VS2    :12258   1st Qu.:61.00
##  Median :0.7000   Very Good:12082   F: 9542   SI2    : 9194   Median :61.80
##  Mean   :0.7979   Premium  :13791   G:11292   VS1    : 8171   Mean   :61.75
##  3rd Qu.:1.0400   Ideal    :21551   H: 8304   VVS2   : 5066   3rd Qu.:62.50
##  Max.   :5.0100                     I: 5422   VVS1   : 3655   Max.   :79.00
##                                     J: 2808   (Other): 2531
##      table           price             x                 y
##  Min.   :43.00   Min.   :  326   Min.   : 0.000   Min.   : 0.000
##  1st Qu.:56.00   1st Qu.:  950   1st Qu.: 4.710   1st Qu.: 4.720
##  Median :57.00   Median : 2401   Median : 5.700   Median : 5.710
##  Mean   :57.46   Mean   : 3933   Mean   : 5.731   Mean   : 5.735
##  3rd Qu.:59.00   3rd Qu.: 5324   3rd Qu.: 6.540   3rd Qu.: 6.540
##  Max.   :95.00   Max.   :18823   Max.   :10.740   Max.   :58.900
##
##        z
##  Min.   : 0.000
##  1st Qu.: 2.910
##  Median : 3.530
##  Mean   : 3.539
##  3rd Qu.: 4.040
##  Max.   :31.800
##
```

9.3 Rare-events and Zero-inflation

Chapter 10

Bayesian Models

Bayesian models may seem like a whole new world, with different definitions of probability, new interprestations, and a suspicious lack of *p-values* (how do I know if it's significant? What about the stars?) Bayesian methods are very similary to maximumlikehihood, however. If fact, under two conditions, Bayesian and ML models will give you the exact same answer. The first is as N approaches infinity. Of course this never actually occurs, but if you have enough observations, Bayesian and ML models will be the same. One advantage of Bayesian analysis, is that it often outperforms ML when your dataset is small. The second condition under which Bayesian and ML models give the same answer, is when the Bayesian priors are flat.

10.1 Using STAN in R

10.2 McElreath and Rethinking

Chapter 11

Panel Models

11.1 Fixed-Effects

11.1.1 Adjusting Standard Errors

11.2 Random-Effects

11.2.1 PML

11.3 Mixed Models

11.4 Multilevel/Hierarchical Models

Chapter 12

Missing Data

12.1 Amelia II

12.2 k-NN Imputation

12.3 Machine Learning for Missing Data

Chapter 13

Matching

13.1 The MarchIt Package

```
library(MatchIt)
library(modelsummary)
library(tidyverse)
library(Zelig)

m.cem <- matchit(treatment ~ x1 + x2 + x3,
                 data = dat, method = "cem")
m.gen <- matchit(treatment ~ x1 + x2 + x3,
                 data = dat, method = "genetic")

m.cem <- match.data(m.cem)
m.gen <- match.data(m.gen)

models = list(
    ‘CEM‘ = zelig(y ~ treatment,
                  data = m.cem, weights = weights) %>%
        from_zelig_model(),

    ‘Genetic‘ = zelig(y ~ treatment,
                      data = m.gen, weights = weights) %>%
        from_zelig_model()
)

modelsummary(models = models, stars = TRUE)
```

Chapter 14

GIS

Chapter 15

A Brief Introduction to Coding

15.1 Why Learn to Code?

15.2 Coding Basics

15.2.1 The Internet is your Friend

15.3 `ifelse()`

15.4 Writing a Function

15.5 Loops

15.5.1 Loops in Loops

15.6 Parallelization: What it is, Why it's Awesome, and How to Use it.