A brief guide to using the **R** statistical programming language[1]

Danielle A. Presgraves

August 12, 2021

[1]This entirely informal book was created in the same way that most scientists learn and use **R**: by painstakingly googling error terms present in log files and seeing how other *smarter* individuals solved that particular problem. StackOverFlow and Google were utilized prodigiously. This was tedious; you're welcome. Most data sets come from Whitlock and Schluter, the textbook used for Bio/Stt 214.

Contents

Chapter 1

Downloading R and the RStudio IDE and getting started with RStudio

[1]

[1]This may be the chapter that you struggle with the most if you are new to programming. It is also the longest and most dense chapter since getting up and running in RStudio and R tends to require a lot of disjointed pieces of initial knowledge. Learning all of this will be worth it and most students assure me that the course gets much more straightforward - and less time consuming! - after the first few chapters.

1.1 Goals

- Introductory words about **R** and **RStudio**
- The Basics of **R**
- The Basics of **RStudio**
  - Structure of the program
  - Data entry/access
  - Data manipulation
- Basic data visualization (covered in the appendix Initial_Data_Exploration)
  - mosaicplot with built-in data
  - scatterplots with uploaded data set

1.2 Introductory words about R and RStudio

R is powerful but it can also be quite intimidating. Luckily, there exists **RStudio** which couples the power of **R** in a much more user friendly package. Of course, the underlying engine used in **RStudio** is still **R** which means that we will need to download both files. **RStudio home** can be found here: `https://www.rstudio.com/products/rstudio/download/`. After you have followed the instructions to install **RStudio**, which provides the environment, you will also need to install **R**. In order to do that, you may need to use the command > *install.packages("rmarkdown")* in the console quadrant in order to get **R** working in the **RStudio** environment (note: the newest version of RStudio may not require this since which libraries RStudio includes with various updates changes. However, if RStudio won't display Chapter1.rmd file then you will need to install the library using the install.packages command in the **Console** quadrant at the > icon). *Instructions in downloading R/RStudio have been included in the same files as your syllabus and, of course, you can always look up a youtube video on how to install these programs (hurray for open-source software!).* I have also included the general outline of installing R below for completeness only - you will find the instructions uploaded to your syllabus folder on Blackboard easier to follow and I will update them occasionally unlike the generic instructions given below:

1.2.1 Instructions for Installing R on your very own computer

In case you ever have an assignment due and have forgotten/dropped/spilled water on your laptop you should know that many computers on campus, including the ones in Carlson library, already have **R** or even **RStudio** already installed on them. However, say that for some bizarre reason, you really really hated **RStudio** and only wanted to use naked, ol' **R**?

- Go to **R homepage**: `https://www.r-project.org`
- Pick a mirror site: In the left sidebar menu, click "CRAN". Scroll through the list and click on an appropriate mirror site.
- If you have questions about how to download **R** for your particular platform (Windows or Mac) and you don?t understand this document, click on the FAQ section, followed by the **R** FAQ link and click on "How can **R** be installed"
- Download **R**. Clock on the appropriate PRECOMPILED BINARY distribution of **R** for your operating system (Linux, (Mac) OS, Windows)
- Install **R**. Install the software by double clicking on the installer (if it doesn't automatically start up) and going through all the steps. Keep all the default options.

1.2.2 Instructions for Installing RStudio

1. Go to www.rstudio.com and click on the "Download RStudio" button.
2. Click on "Download RStudio Desktop."
3. Click on the version recommended for your system, or the latest Windows version, and save the executable file. Run the .exe file and follow the installation instructions

RStudio has features that can make life easier for those undertaking data exploration. There are specific features which ensure that your **R** coding experience is improved such as automatic bracket highlighting so you remember to close brackets before compiling and commands have code completion, so you can often get away with not remembering exact command for a function. This is an amazing feature if you toggle between languages and can't remember the different commands for the same behaviour among various programming languages. More generally, **RStudio** has four quandrants which allow you to see the graphics you produce on the same screen as the console so you tweak your graph in real time without having to scroll between screens. We investigate those four quadrants and give an example of uploading a dataframe in the accompanying Chapter1.rmd file.

Additionally, there is quite a bit of support for the **RStudio** environment. For instance, there are numerous "cheat" sheets that will help you with common commands here: `https://www.rstudio.com/resources/cheatsheets/`. There is a lot of help at websites such as StackOverFlow and even Google - the point of this course is to ensure that you develop fearlessness and autonomy with the R programming language so you will also become familiar with these websites. I have been using R for years and I still, occasionally, have to spend a couple of hours trying to track down a specific package or function when developing a pipeline and I use these websites A LOT! It can feel like wasted time in hindsight (once you have figured out the issue, of course) but it is how you end up learning these languages so don't get discouraged too easily!

Another exciting feature of **RStudio** is the ability to create ***integrated documents*** for your data analysis by using "R Markdown language". Instead of having to cut-and-paste constantly, you can include specific sections of your **R** code and graphics directly *in* your text document. How do you accomplish this magical feature? The basic steps are outlined below (and are presented in more detail in Chapter1.rmd):

1. under the **RStudio**"File" pull down menu –>"New File' and choose the option "R Markdown"

2. this will open a new window for you to save your R Markdown file. The default, at the top, is "document" but you could choose "Presentation", "Shiny" and "From Template". We won't worry about those other options. Name your R Markdown File and choose the "HTML" option since you can save it as a pdf later (it is a more flexible option).

3. Now in the upper left hand quadrant of **RStudio** is your new R Markdown file. Which we will now fill up with commands!

1.3 The basics of R

R is a very powerful tool for statistical (and other types of) computing, but it takes a bit of getting used to since, before **RStudio** came along, it had only a command-line interface rather than a drop-down menu.This has advantages and disadvantages. Lucky for us, **RStudio** has allowed us to have the best of both worlds: namely, we have the big advantage of extreme flexibility and expandability– instead of being limited to only the options the application writers thought were appropriate to include you can, basically, do anything you want with data in R but it is now easier with the **RStudio** interface to install packages and scripts for specific projects that aren't included in the standard download of R. The old fashioned R scripting can still be accessed in the **Console** quadrant of the **RStudio** environment. You can also write entire R scripts in the quadrant where we will write markdown scripts by choosing the R script option in the pull down menu (the same pull down menu where we will always choose the R Markdown option for this course).

1.3.1 Nuts & Bolts of R

1. **Getting help with R:**

   - There are a lot of websites that will help you. This is one of the benefits of **R**; a large community of online support!
   - official R website and a couple of REALLY useful ones: `http://www.cyclismo.org/tutorial/R/` and `http://www.r-tutor.com/`.

- There is even a beginner level and clear website written by Dolph Schluter (coauthor of our fine textbook for BioStt 214): `https://www.zoology.ubc.ca/~schluter/R/`.

2. **Structure of R**

   - **Basic Math Calculation** Some people are surprised to learn that they can do basic math calculations in R. Try a few:
     >1+1 [return]
     >10/4 [return]
   - **Command line** Like many other command-line systems, you can scroll back up through past commands by using the "up" arrow on the keyboard. This is very very helpful when you are editing or repeatedly calling a complicated function that you don't want to type out multiple times. Keep this in mind as we go on...
   - **More Complicated Calculations** Most commands in R have the syntax >function(arguments). We can start exploring this with more complicated mathematical functions such as square root. Type >sqrt(9) [return]. Notice that, as you typed "sqrt(", R did two things for you. First, it completed the parentheses for you (very helpful when you are doing more complicated functions), and second, at the gray bar at the bottom of the console, it gave you a hint as to what it expected inside the parentheses. Try another function, >log(), to see how this works. Notice that for this function, there are two arguments– the number to be "logged", and another optional argument, which is set as a default "base=exp(1)". What this means is that you can specify the base of the logarithm, and that **R**'s default is to set it at base e. Try the following two commands: log(100) [return], and log(100, base=10) [return], to see how this works.
   - **Help function**: Sometimes, you need a bit more information to parse out what's going on with a new function than is in that bottom bar. Type help(log) for an example. This opens up a new window with the detailed help file for the function. This file will include all the required and optional arguments of the function. Here, we learn that we could have used a shortcut to get the base 10 log– there is a built-in function, log10(). Try it in the console window.
   - **Basic Data Management** R is more than a big calculator, and it is smart enough to be able to do its work on things other than numbers typed directly onto the command line. R is an object oriented language. A single number can be an object (such as the numbers we added above), output is an object, and variables can be objects. Most of the time, we want it to work on data that has been stored into *variables.*
     - **Assigning values to a variable name.** The assignment operator is "<-" (less than sign followed by a dash). Type a<-2 [return]. This puts the

value "2" into the variable name "a", which will then behave just like the value it contains. Type a [return]. Type a+5 [return]. You can also assign expressions (commands): Type b<-sqrt(9) [return], then b [return] and a+b [return]. Did you get what you'd expect? What about VAR1<-2+3 (also try: 2+3-<VAR2)
There are some set rules about names (e.g. they must begin with a letter rather then a number of an operator and they cannot contain certain character such as space, #, % , ˜, [], {}, ()) and some conventions that should be followed (e.g. avoid names that are predefined functions; remember case sensitivity; give unique names that are as short as possible while still being informative; separate words in names by "." such as head.mean.length)

  - **Vectors.** A vector is a collection of objects of the same type (e.g. numeric). More specifically, vectors can be in two flavours: atomic vectors (where the data is all in one form: integer, logical, character etc) and lists, which can be made up of a bunch of different data types. Note that vectors of characters use double quotes. We will often be using sets of numbers since it is difficult to convey any interesting information with one dot. For example, we might want to measure several individual's diastolic blood pressures and analyze b.p. for that population. If we took 10 such measurements and got (in mmHg) 82, 61, 90, 86, 78, 76, 72, 72, 70, and 60, we can save those as a vector of values in R by using the **c()** function. The "c()" function is the combine function. Type dbp<-c(82, 61, 90, 86, 78, 76, 72, 72, 70, 60) [return], then type dbp [return] to see how it got stored.
  - **Dataframes.** You can think of a vector as a column of data in Excel, with the name you assigned to the vector (such as "dbp" above) as the name of the column. Most of the time, we'll have several such columns we want to deal with simultaneously– for example, with the blood pressure data we may have more sets of 10 numbers indicating other measurements on the same subjects. The equivalent to an entire Excel file with many adjacent columns and their associated names is a structure called a dataframe– we'll be using dataframes a lot since they are flexible in the type of data they can contain. names() gives you the name of each of the columns. Another type of data structure in R is called a table and there are arrays etc but we won't cover these . . . at least not today.

- **Programming in R** Remember that **R** can be used for programming too. In fact, we will focus on programming during Chapter 11 (remember the "SWIRL" package is great as an introduction to **R**, including programming) and you will have some encounters with small programs throughout most of the chapters. Because combining statements is so ubiquitous, here is a mild warning: If you have had some experience with other programming logic, this may confuse you (depending on the your prior language experience and we will re-visit this difference when it is appropriate in the future): simple Boolean expressions using the & ,the logical expression “AND” apply across all elements of a vector, whereas &&, “AND” applies across JUST the first element of a vector, | , the logical expression “OR” across all elements of a vector and || ,“OR” across just the first element of a vector, can be evaluated to create straightforward programming expressions. It is also important to realize that all “AND” expressions are evaluated before the “OR” expressions.

1.4 Getting Started: Appendix Initial_Data_Exploration

The best way to get used to how **R** and **RStudio** work is to jump right in. With that in mind, we will work through a couple of examples that are found in the appendix of this book called Initial_Data_exploration. We won’t work through the entire document right now, some of the functions will be re-visited as we work through different chapters of this course and some parts of the appendix will be used to illustrate different features as we gain more experience with **R**. We will start by working through the **BP.csv** data set to explore how data is brought into **R** (hint: file.choose() is a useful function) and to conduct some basic data manipulation such as plotting the data as a scatterplot and as a histogram. We will also use the built-in data set Titanic to look at some sophisticated functions, like apply() which will be re-visit later, and mosaicplot().

1.5 Importing data into R:

We can get data into R through several means. We can type it in directly (only recommended for small data sets) by using the **c()** function.
Example: the function **vector.name<-c(1,3,5,7,9,11)** would result in a vector called 'vectorname' in memory that contains 6 numbers: 1,3,5,7 and 9.
For more complicated datasets with multiple categories we can import it from a spreadsheet program such as Excel. There are lots of ways to format data files and read them in, but in this class we’ll keep it simple– all data files should be in “comma-separated” format (.csv) (it’s easy to save an Excel file in this format), and read in using a function called *read.csv()*. Let’s try this for a file containing blood pressure data from patients in the Dominican Republic. The file is called “BP.csv”.
Type >bp<-read.csv(file.choose()) [return]. The assignment vector is “<-”. Locate BP.csv and select it for import. We’ve now loaded the file into our workspace, and assigned it to a dataframe named bp.
Did it work? Type >bp [return]. You should see the dataset printed out in the R console. If you want to just see the names of the variables, you can type >names(bp) [return].
Can we work with the individual columns of the dataset (the variables)? Type DPB [return].

Hmmm. R doesn't (yet) know the individual variables (column names). But you can easily make R integrate all the information in the bp dataframe by "attaching" the dataframe. Type **attach(bp)** [return]. Now type DBP [return]. R should print out the column of diastolic blood pressure readings– it now knows the data by the column names, and we can work with those names as regular variables.
Try a couple of basic operations on the variables you saw using names(). For example, you can get the average diastolic blood pressure using mean(DBP) [return], the minimum age by min(Age) [return], the maximum systolic blood pressure by max(SBP) [return], etc. You can find out the levels (different categories) of categorical variables using levels(variableName)[return].
R When you're done with a dataframe, it's good practice to detach it– this prevents needless conflict with new variable names. Type **detach(bp)** [return]. Also, since we made up some stand-alone variables earlier, we should get rid of them, too (we almost already had a conflict with two vectors for diastolic blood pressure with very similar names. Type remove(a, b, dbp) [return]. We can also remove the entire dataframe after we've detached it, using remove(bp) [return]. Your workspace should be nice and clear now. Check by typing ls().

IF YOU ARE LAZY OR "IN A HURRY" (as I often am) **AND YOU KNOW THAT THERE WILL NOT BE A CONFLICT WITH NAMES**, YOU CAN ALSO JUST USE $ TO INDICATE THE COLUMN OF INTEREST.

Proper Way:
>attach(dataframeName). Remember to >detach(dataframeName) when finished
Fast Way:
Use "$".
Example 1: dataframeName$colName (or in our previous example, >bp$Age)

1.6 SWIRL

I am not sure that I extolled the virtues of SWIRL *enough* in the first module .rmd file. Hadley Wickham `https://en.wikipedia.org/wiki/Hadley_Wickham` HIMSELF believes that it is an excellent resource for a beginner to become acquainted with the full spectrum of functionality that R possesses which is the strongest recommendation that it can be given. We will use the SWIRL() program directly in Chapter 11 to introduce ourselves to programming but you might want to play around with it now to expose yourself to the basics of the R programming language.

SWIRL() is an interactive way to pick up the basic of the R programming language in a hurry. This program works best in **RStudio**. You open the program and it takes you through various commands interactively so that you are able to learn **R** *while you are using*

R. The link to Swirl is **here**. We will learn about other packages that are especially useful for biologists' in Chapter 2 but until then, in order to get the SWIRL() package up and running, you will type the following into your **R** console quadrant:

- install.package("swirl")
- library("swirl")
- swirl()
- install_from_swirl("R Programming")

The **swirl** program will then take over and allow you to work through a series of interactive modules to learn some basic programming in **R**. If you can't download the **R Programming** module then go **here** for more instructions on how to download the modules manually.

1.6.1 An illustrative-but-outdated example of the install function.

The `https://cran.r-project.org/web/packages/foreign/index.html` is useful; it allows you to import data into R which is in the format of another program, such as STATA or minitab or SPSS. To install this outside package, you simply type >install.packages("foreign") into the console quadrant and then type >library("foreign") to run it! (N.B.: if you are using version 3 or higher of R - and everyone at this point should be! - than this library is already installed in the standard package so you won't need to install it.). Installing packages is even easier in **RStudio** - there is button under the "Packages" heading that you simply click on! In Chapter 2, we will download the ggplot2 package for displaying data so you will use the install package pipeline very soon.

This is yet another aspect of R that makes it so appealing to scientists who often need to share scripts or data files among themselves even if they can't all afford (or prefer to use) the same statistics software. The big disadvantage of R is that it takes a little longer to learn how to use it, but once you are comfortable with the basics, it's pretty smooth. R is also FREE! Which means that it is highly portable - even if you end up at an institution/company (or collaborate with colleagues who do) after graduation that can't afford a fancier statistical software package, you can still use R.

In the appendix is a finished document produced from RMarkdown called, strangely, Initial_Data_Exploration.

Chapter 2

Graphics and basic Statistics

2.1 Goals

- Learn to graph data as the first step in data analysis
- Make graphs, such as histograms, scatter plots
- Learn how to access packages to extend the functions that you can use
- Learn how to make graphs in **ggplot2**

2.2 Data Visualization

Human beings are visual creatures; much of the hardware in our brains is dedicated to processing and interpreting visual signals. Effectively (and efficiently) graphing out complex data to uncover trends is a highly coveted skill and one that is well represented in foundational statistics and the post-graduation job market! If you are interested in learning more about displaying information, Edward Tufte (referred to as the Guru of visualization by many...) has the best-known and foundational book on that topic called "The Visual Display of Quantitative Data".

R took off as a programming language partially because of its effective graphing functionality. In module 2 we will continue to explore the flexible graphing abilities included in base R by building on the examples that we worked through in the appendix and in Chapter 1. We will also investigate the philosophical shift in data visualization and added sophistication with the ggplot2 package.

2.3 More Practice importing data sets

We will continue where we left off in Chapter 1 by importing a handful of data sets - one of them, BP.csv, we imported last week - which are up on Blackboard, Blowflies.csv, and the last one includes a string of numeric values that will give you a chance to practice inputting data directly into an R Chunk. There is a function called **c()** that allows you to input a vector into R memory so you will need to use that to directly input your numeric values.

- Blowflies.csv
- BP.csv (already imported from first part of lab)
- Male fireflies of the species *Photinus ignitus* attract females with pulses of light. During mating, the male transfers a spermatophore to the female. The following are the sizes of transferred spermatophore (data is from Whitlock and Schluter, Q19, chapter 2): 0.047, 0.037, 0.041, 0.045, 0.039, 0.064, 0.064, 0.065, 0.079,0.07, 0.077, 0.059, 0.075,0.079, 0.09, 0.069, 0.066, 0.078, 0.066, 0.078, 0.066,0.066,0.055, 0.046,

0.056, 0.067, 0.075, 0.048, 0.077, 0.081, 0.066, 0.172, 0.08, 0.078, 0.048, 0.096, 0.097. **You will need to use the function c() to do this**.

2.4 Basic Plotting of Data in R

Plotting Data The most important first step is to identify variable type: In order to determine what type of plot would be the most appropriate, you first need to identify the types of variables involved in your data set/question. You can also ask **R** to do that for you, where x =column in your dataframe:
>is.vector(x)
>is.numeric(x) etc.
Example 2: Identify the variable types from above .csv files and plot them appropriately.
Commands for different plots: After you identify the types of variables involved in a data set, it is useful to inspect the data visually. Lucky for you, **R** is particularly adept at data visualization! There are 'high level' functions that govern the distribution graphed and there are 'low level' adjustments that can be made to each graph to tweak the visual output.

2.4.1 Scatterplot

The most straightforward plot is the scatterplot and this is the likely starting point for initial data exploration. The plot() function is a generic overloaded function (which is scary programming lingo that means it results in different types of plots depending upon the class of objects passed to it). To produce a scatterplot of y against x, both must be numeric. The function will simply be: ***plot(x,y)***. The plot() function can be modified by tweaking parameters. For instance, you might be interested in adding lines to the graph via the type parameter:

ie. plot(x,y, type="o")

Type	Description
p	Points
l	Lines
o	Overplot
b,c	Points joined by lines
n	No lines or points

Of course, in **R** there are always multiple ways to achieve a goal. The function lines() can be used after graphing a scatterplot and will, strangely enough, add lines attaching the points. The lines( ) function adds information to a graph. It can not produce a graph on its own. Usually it follows a plot(x, y) command that produces a graph. This function can be further tweaked via the lty parameter:

plot(x,y)
lines(x,y, lty=6)

lty can take a value between 1 (solid line) and 6 (a dotted line). There are a number of other values but they start to become a bit complicated. From the files which you have imported, you can use the scatterplot as demonstrated: >plot(blowflies$ day,blowflies$ number_emerging). This is the type of command, maybe with some variation (giving it a title, specifying a color etc) that you should have used with your homework.

2.4.2 Histogram

Histograms show the frequency of the occurrence of all the values of a variable. Histograms can be a bit tricky since there is no set rule about the number of classes to use. Remember that you want to use enough classes in constructing your histogram for it to be informative for an audience without being overly complicated. You also want to play around with the intervals to ensure that you are not creating artifacts by making your categories too small and that you are not destroying real features of the distribution by collapsing two many of the categories. For instance if you use the command: **hist(x,breaks=18)**, you can the results to the following: **hist(x,breaks=5)**. Now try to create more breaks and see the effect that has on the graph: **hist(x,breaks=20)**. Which of these two graphs, graphing the same data, is more informative to the audience?

2.5 the ggplot2 package

In today's lecture, we will download the package (mentioned below) "ggplot2". The ggplot2 package, developed by the R guru Hadley Wickham (his personal website is here:http://hadley.nz; a website that shows you how to obtain ggplot2 is here: http://ggplot2.org; finally, a website that contains basic documentation for ggplot2 is here: http://docs.ggplot2.org/current/), that should be easier to use on complex data than the base packages of R. There are a number of very smart people who argue that you should teach the graphics packages created by Hadley Wickham first. He has a bit of a cult following.... You can see at least one opinion about that **here**: `http://varianceexplained.org/r/teach_ggplot2_to_beginners/`. I have updated the appendix to include an example of data visualization using the ggplot2 package called **And now for some ggplot2 package graphs**. ggplot2 can result in even more beautiful graphics than **R** and, in general, is easier to use (but not for everything).

Once **ggplot2** is installed, every time that you want to use it, you will need to check the box in your packages list AND you will also need to load the library in an **R chunk**. This is how packages/libraries work in general; you download them once to your RStudio memory but you need to call them in every document that requires them. This stops you having to utilize memory on libraries/packages that you may only use in certain files. Your chunks will individually evaluate and you will produce plots using ggplot2 simply by checking your package box but you won't be able to knit together your final document without including the following command in one your first chunks: **library(ggplot2)**. You can now use the **ggplot()**function. Remember when we quickly visited the generic plot function that is present in base R? **ggplot** is the generic plot function that is part of the ggplot2 package. You always start a plot by using the **ggplot()** function since that creates a coordinate system and then you add 'layers'.

ggplot uses geometrical objects, called *geoms*, to visually represent the dataset. Bar charts use *geom_bar*, histograms use *geom_histogram*, boxplots use *geom_boxplot* and scatterplots use *geom_point*. All of these geoms have a wide range of arguments and aesthetics that can be used with them. There is a cheat sheet here: https://www.rstudio.com/wp-content/uploads/2015/03/ggplot2-cheatsheet.pdf. We worked through two data sets that are built into the **ggplot2 package** in class and in the **ggplot2** appendix in this book: mpg and diamonds. Like any package (or even **R** itself), **ggplot** has a bit of a learning curve where you figure out, usually through examples, how to produce the graphs and features that you want from the data set that you have. However, the result of your efforts will be such beautiful graphs that you will be happy that invested the time to learn ggplot.

2.5.1 Some other useful packages for biologists

- pegas: contains functions that are crucial to population genetics such as F_{st} and Tajima's D measurements.
- plyr - dplyr is the newest version: One unique aspect of dplyr is that the same set of tools allow you to work with tabular data from a variety of sources, including data frames, data tables, databases and multidimensional arrays. The first step of working with data in dplyr is to load the data into what the package authors call a "data frame tbl" or 'tbl_df'. dplyr has five functions that are particularly suited to data manipulation: elect(), filter(), arrange(), mutate(), and summarize()
- reshape
- For bioinformatics and genomics, the following two programs provide important functionality:
  - bioconductor set of **R** pages, found `http://bioconductor.org` contains a set of **R** packages that can be used to analyze microarray data amongst other data types
  - seqinr is a library for analyzing DNA and protein sequences.
- These are the Hadley Wickham Packages for data visualization (there are more):
  - ggplot2
  - lattice

Chapter 3

Describing Uncertainty

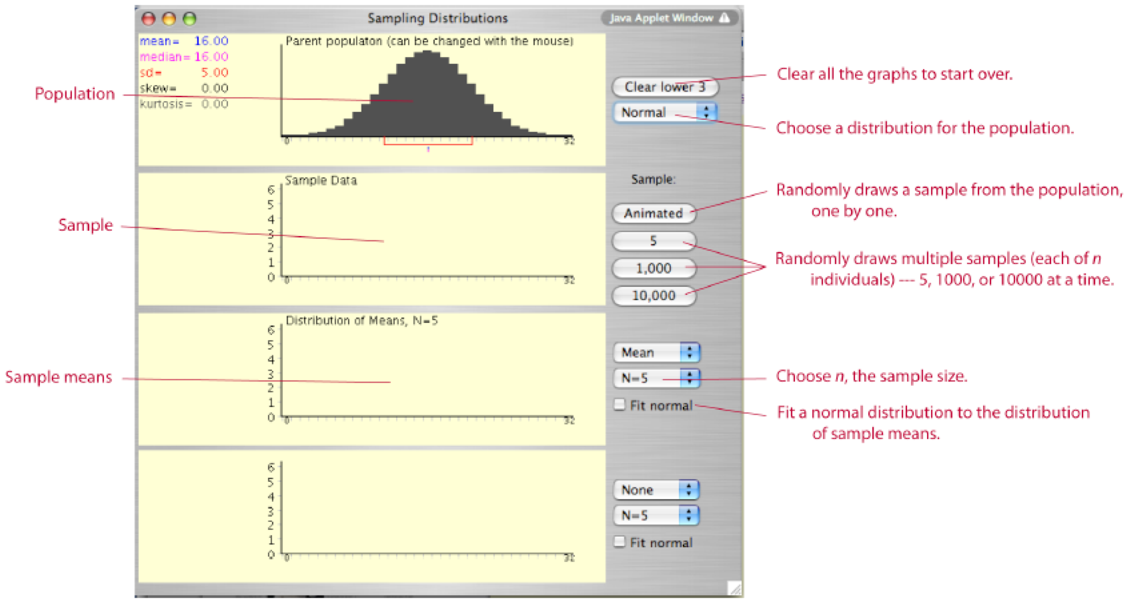
3.1 Goals

- Review the concept of uncertainty: Sampling distribution of the mean and confidence intervals using online simulations
- Calculate basic summary statistics using **R**
- Calculate confidence intervals for the mean using **R**
- Understand box plots.
- Visualize confidence intervals using directly inputted data sets and ggplot2
- Bootstrap simulations (for confidence intervals when assumptions, like normality of your data, are not met).

3.2 Review Activities:

3.2.1 Distribution of sample means

Open the website found `http://onlinestatbook.com/stat_sim/sampling_dist/index.html`. This site contains a java applet that allows you to play around with sampling and to see the distribution of sampling means.



Sampling Distributions
Java Applet Window
mean= 16.00
median= 16.00
sd= 5.00
skew= 0.00
kurtosis= 0.00
Parent populaton (can be changed with the mouse)
Population
Clear lower 3
Clear all the graphs to start over.
Normal
Choose a distribution for the population.
Sample Data
Sample
Sample:
Animated
Randomly draws a sample from the population, one by one.
5
1,000
10,000
Randomly draws multiple samples (each of n individuals) --- 5, 1000, or 10000 at a time.
Distribution of Means, N=5
Sample means
Mean
N=5
Choose n, the sample size.
Fit normal
Fit a normal distribution to the distribution of sample means.
None
N=5
Fit normal

The box shown contains a distribution that represents the true distribution of a variable in the population. This is the true universe that the program is using to sample individual data points. In the example pictured here, the mean of the population is 16, with standard deviation 5, and the individuals in the population occur according to a normal (bell-shaped) distribution. Next click the **"Animated"** button on the right side. This will draw five individuals at random from the population, and plot a histogram of the sample data in the second graph. It will also calculate the mean of the sample, and start to create a histogram in the third graph. The third graph shows the distribution not of individuals, but of sample means. Notice how the bar added in the distribution of sample means corresponds to the mean of the sample in the second graph.

If you click **"Animated"** again, it will draw an entirely new sample from the population, and then add the mean of that new sample to the distribution of sample means. Do this a few times, and watch what's happening. After you get bored of that, click the button that says "5". This button will draw 5 different samples, and add the five new sample means to the histogram of sample means. The "1,000" and "10,000" buttons do the same thing, except for more samples.

After adding a large number of samples, look at the distribution of sample means. What kind of shape does it have? It ought to look normal as well. Also look at the width of the distribution of sample means - is it more variable, less variable or about as variable as the population distribution? What do you expect? Next, hit the **"Clear lower 3"** button at the top. This will erase all the distributions that you just drew. Use the "N = 5" button to choose a new sample size, starting with N = 25. Repeat the steps above to generate a distribution of sample means. Now the sample means are based on 25 individuals instead of just 5 individuals. Are the sample means more or less variable with N = 25 than with N = 5?

3.2.2 Confidence Intervals

Go back to the web, and open the java applet `http://onlinestatbook.com/stat_sim/conf_interval/`. This applet draws 100 different confidence intervals for the mean of a known population. Hit "Sample", and look at the results. Each horizontal line on the graph represents a different sample of 10 individuals (or more, if you change the sample size). Each line shows the 95% confidence interval for the mean (the inside part of the line in red or yellow) as well as the 99% confidence interval (the wider line in blue or white). The 95% confidence interval is shown in yellow if the interval contains the true value of the mean, represented by the vertical line down the screen. The program changes the 95% confidence interval line to red if the 95% confidence interval fails to enclose to true mean. Out of the 100 samples, how many 95% confidence intervals contain the true mean? How many missed?

Click "Sample" a few more times. Each time you click, the program will draw 100 more samples and draw them. It also keeps track of all the samples you have drawn in the table at the bottom. After drawing a lot of samples (say 1000), what proportion of samples have 95% confidence intervals that include the true mean? What proportion fail to enclose the true mean? What about the 99% confidence interval – what fraction of 99% confidence intervals enclose the true mean?

A 95% confidence interval ought to enclose the true value of the parameter for 95 samples out of 100, on average, and the 99% confidence interval ought to succeed 99% of samples. In fact, of course, that is the definition of a 95% (or 99%) confidence interval; the percentage tells us the expected probability that the true mean is captured within the confidence interval.

3.3 Illustrative Examples:

A straightforward way to visually scrutinize the general features of the distribution of a *categorical* variable is to graph a **boxplot**. This graphing method captures the median and the spread of the data and makes it easy to quickly see any outliers and asymmetries. (N.B.: a histogram, *for numerical data*, also helps visualize spread and distribution, including possible outliers.) The easiest function to use to create a boxplot happens to be >boxplot(y∼x, data=name). Where x = factor , **factor** is the category, and y = dv (dependent variable). Remember: if you have attached your data, then you won't need to also specify the data=name; if you haven't attached your data set, you will need to include it. You can also create a boxplot rather than a scatterplot by 'overloading' the plot() function (using the plot() function with different arguments) in the following ways:
plot(fact, dv) or plot(dv∼fact) when fact is a factor vector (ie. high shade, medium shade, low shade or no shade when measuring plant growth) and dv is a numeric vector (ie. temperature)

Using the following two data sets: caffeine.csv and caffeine_Starbucks.csv (both of which have *categorical data*) and a directly inputted data set with *numerical data*, we will explore the following:

- Calculating summary statistics
- Calculating confidence intervals of means
- Plotting these data sets with boxplots

3.3.1 Boxplots: Caffeine Examples

The file "caffeine.csv" contains data on the amount of caffeine in a 16 oz. cup of coffee obtained from various vendors. For context, doses of caffeine over 25 mg are enough to increase anxiety in some people, and doses over 300 to 360 mg are enough to significantly increase heart rate in most people. Red Bull contains 80mg of caffeine per serving.
a. What is the mean amount of caffeine in 16 oz. coffees? Use the function >mean()
b. What is the 95% confidence interval for the mean? In **R**, the easiest function to use to get the confidence interval of a mean is >t.test(). Don't worry too much about what the

function does right now, we will be spending time on t-tests in Chapter 7 and the action of this function will become obvious then! The t.test() function has an argument, conf.level that allows you to specify the confidence level calculated (= 0.95 or = 0.99) (As usual, use >help(t.test()), or pull up the help menu, for more information. The t.test() function will return some information that we are not interested in for the present time but it will also return two numbers that form the boundaries of the specific confidence interval specified and the mean of the data. It is a good idea to confirm that the mean from the t.test() is the same value for the data as was produced by mean(data).
c. Plot the distribution of caffeine level for these data. Is the amount of caffeine relatively predictable in a cup of coffee? What is the standard deviation of caffeine level? What is the coefficient of variation (there is no package in **R** for coefficient of variation so just calculate it using the formula $100\% \times \frac{s}{\bar{x}}$)?
d. The function quantile() will give you the values for the percentiles specified as arguments. Ex. quantile(data, c(0.05,0.95)) will give the value of the cut-off for the bottom 5% of values and the value for the cut-off of the bottom 95% of values. If the function is used without specifying the percentiles, then it gives the standard 0%, 25%, 50%,75% and 100% values.
e. The file "caffeine-Starbucks.csv" has data on six 16 oz. cups of coffee sampled on six different days from a Starbucks location, for the Breakfast Blend variety. Calculate the mean (and its 95% confidence interval) for these data. Compare these results to the data taken on the broader sample in the first file, and describe the difference.
f. We are going to cheat a little bit and use the caffeine-Starbucks.csv to create a boxplot of one. Since there is only ONE factor (starbucks is the only brand being tested), we will avoid specifying an x. Type >boxplot(Caffeine..mg..in.16.oz..coffee). This should result in graph with ONE boxplot. Lets examine the information that the boxplot displays. What do the limits of the central box correspond to? What does the thick line within the box represent? What are "the whiskers"?

3.3.2 Directly inputted numerical example

Suppose we've collected a random sample of 10 recently graduated seniors and we have asked them to report their starting salaries:
>x<-c(44617,7066,17594,2726,1178,18898,5033,37151,4514,4000)

What is the mean salary of all recent graduates from the program? Give the 90% and 95% confidence interval for this estimate of the population parameter.

First, calculate the mean and the standard deviation of the data set:
>mean(x)
>sd(x)

Now, to calculate a confidence interval of 90%, we need an α=0.1. We have estimated **sd** from our data thus using up a degree of freedom. Due to that, we can't use a normal distribution but we can use the *t distribution* with **9** degrees of freedom as we have **10** independent data points and have use up one to estimate sd. Since we know that this is a

two-sided distribution (there is no reason to believe that there aren't as many graduates earning low salaries as those earning high salaries), we can use:
>qt(0.95,9)
Remember that we have a sample of n=10:
>StError<-qt(0.95,9)*(sd(x)/$sqrt(n)$)
The Upper Boundary of the 90% C.I. is: >mean(x) +StError
The Lower Boundary of the 90% C.I. is: >mean(x) -StError

You can then repeat this process for the 95% confidence interval with the only difference being that we need to use α=0.05 which results in >qt(0.975).
What is the 95% Confidence Interval for the given data set?

Note that to obtain the summary statistics of a particular dataset, you can use the **summary()**. Depending on the type of data you use as an argument for the summary command, it will return different information to you.

You can use **ggplot2** to graph a confidence interval once the upper and lower boundaries are specified using either **geom_linerange** or **geom_errorbar**. For the above data set, the **geom_linerange** is probably the better choice but we will briefly look at both options.

3.3.3 Bootstrap estimate

When you draw a histogram of this data set (you should do that now. I'll wait), you find that it isn't really even close to normally distributed, which means that it isn't really appropriate to use a t-distribution since normally distributed data is one of the important assumptions of using the t-test. In fact, you could use bootstrap to estimate confidence intervals. Not all introductory statistics course cover bootstrapping but, for the purposes of this R course, it is sufficient to know that it is an important (and widely used) **resampling technique** that allows you to calculate confidence interval end points on non-normally distributed datasets (or, more crucially, data sets that can't be described with a known distribution - like phylogenies in biology!). This is a reasonable wikipedia article that explains bootstrapping:
`https://en.wikipedia.org/wiki/Bootstrapping_(statistics)`

Below is a reasonable example of how to use bootstrap to determine confidence intervals.

1. from our sample of size 10, we draw a new sample of size 10 **WITH REPLACEMENT**
2. calculate sample average, called bootstrap estimate
3. store it
4. repeat many times - we will use 1000 times
5. For 90% CI, we will use the 5% sample quantile as the lower bound and the 95% sample quantile as the upper bound

```
> bstrap <- c()
> for (i in 1:1000){
+ # First take the sample
+ bsample <- sample(x,10,replace=T)
+ #now calculate the bootstrap estimate
+ bestimate <- mean(bsample)
+ bstrap <- c(bstrap,bestimate)}
> #lower bound
> quantile(bstrap,.05)
5%
7413.795
> #upper bound
> quantile(bstrap,.95)
95%
21906.49
>
We use the same output to get the 95\% confidence interval:
> #lower bound for 95\% CI is the 2.5th quantile:
> quantile(bstrap,.025)
2.5%
6357.615
> quantile(bstrap,.975)
97.5%
23736.75
So the 90% CI is (7414,21906) and the 95% is (6358,23737).
```

As a note, you could make the above code a bit more streamlined (okay, A LOT more streamlined) by the following:

```
> for (i in 1:1000){
+ bstrap <- c(bstrap, mean(sample(x,10,replace=T)))}
and then you find the quantiles as before.
```

Chapter 4

Hypothesis Testing

4.1 Goals

- **Review: Better understand the logic of hypothesis testing, especially Type I and Type II errors with simulations**
- Learn about the binomial distribution (with **on line simulation**,**a little programming simulation** and with **R** commands)
- Apply the binomial test to test for proportions using all four steps of hypothesis testing
- Learn and apply the **tapply** function
- Fit frequency data to a model
- Receiver Operator Curves (ROC) and Associated Area Under Curve (AUC)

4.2 Hypothesis testing: significance and errors

Go to the applet in the lower right hand corner of this link `http://www.intuitor.com/statistics/T1T2Errors.html`. We will explore the effect of sample size, stated α and the distribution shapes on the power of the test.

1. Begin with a sample size of "1". Move the red dotted line to declare the type I error rate to be equal to 0.05, or 5% (a very common value). Leave the mean positions alone. What is the type II error rate that results? What is the value of the power of this test?
2. Change the Type I error rate to 0.01, or 1%. What happens to the Type II error rate and the Power? Why does this happen?
3. Change the sample size to 5 and again move the red dotted line to correspond to a Type I value of 0.05. What is the Type II error rate and the Power?
4. Now repeat the above steps above but go all the way to a sample size of 11. What happens to the shape of the two distributions and why? What is the value of the Type II error and the Power of the test?

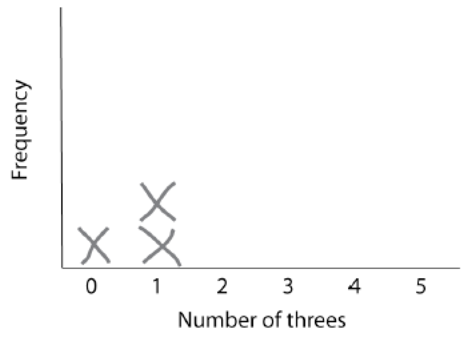
4.3 The Binomial Distribution:

This is a simple, quick exercise meant to help you visualize the meaning of a binomial distribution. The binomial distribution applies to cases where we do a set number of trials, and for each trial there is an equal and independent probability of a success. Let's say there are n trials and the probability of success is p. Of those n trials, anywhere between 0

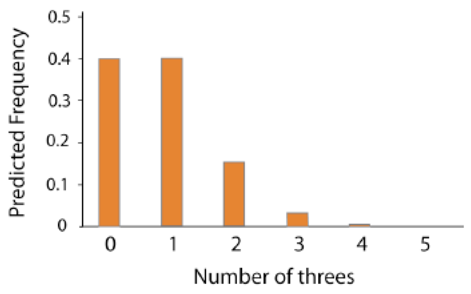
and n of them could be a "success" and the rest will be "failures." Let's call the number of successes X.
As our example, let's roll five regular dice, and keep track of how many come up as "three." In this case, there are five trials, so **n** = 5. The probability of rolling a "three" is 1/6, so p = 0.1666. Let's repeat the sampling process a lot of times. For each sample, roll five dice, and record the number of threes. If you don't happen to have five dice available, there is an on-line simulator `http://www.random.org/dice/`. Be sure to specify 5 dice. Now let's see how to run this simulation with an **R** command.

You can also use the **sample()** function in **R** that we saw in the previous chapter. For instance, to specify five dice, use the following command: **sample(1:6,5, replace=TRUE)** which will create five six-sided dice and sample them with replacement. What do you think happens if you run **sample(1:6,5, replace=FALSE)**?

For this exercise, let's draw a histogram of the results by hand. First draw the axes, and then for each result draw an X stacked up in the appropriate place. For example, after three trials that got 0, 1, and 1 threes, respectively, the histogram will look like this: Keep



doing this until you have 20 or more samples. Here is the frequency distribution for this process as calculated from the binomial distribution Is this similar to what you found?



(Remember that with only a few trials, you don't necessarily expect it to exactly match the predicted distribution. If you increase your number of samples, the fit should improve.) The expected mean number of threes is 5/6 which is approximately 0.83. (For the binomial distribution, mean is **np**) Calculate the mean number of threes in your samples.

4.3.1 Generating the binomial distribution in R

Among its many uses, **R** has a number of built in functions that calculate the probability for discrete and probability densities for continuous variables. In future recitations, the Normal distribution and the Poisson distribution will be examined. For now, we will demonstrate how to calculate the probability of obtaining X successes in “n” random trials, for a binomial variable with the probability of individual success a constant “p”.

>dbinom(X, size = n, prob = p)

If, for instance, you flipped a coin 100 times and it came up heads 80 times, you could use the command above to determine the probability of this outcome. For the binomial distribution (and the Normal and Poisson distributions) there are four classes of command which have slightly different meanings. We can write these in a generic case as: **dfunction**, **pfunction**, **qfunction** and **rfunction**. In the case above, **d**binom(appropriate arguments go here) (or **d**norm or **d**pois), you are calculating the probability of obtaining a probability with the arguments given. For **p**binom(appropriate arguments go here), you are actually calculating out **F(X)=P(X<=x)** $=\int_{-\infty}^{x} f(x)\,\mathrm{d}x$

4.3.2 Sneaking in a little unnecessary programming for practice:

This is taken from a fantastic blog called “Count Bayesie”: `https://www.countbayesie.com/blog/2015/3/3/6-amazing-trick-with-monte-carlo-simulations` We flip a coin 10 times and we want to know the probability of getting more than 3 heads. We could also plug this into the binomial distribution but, instead, we are going to use it to give us a bit more programming experience in R. In fact, we are going to use a **FANCY** method called “Monte Carlo simulation” related to, but not the same as, an even **FANCIER** process called “Markov Chain Monte Carlo” (or MCMC) which you may have heard of. In Monte Carlo simulation we are simply modeling our problem and then repeatedly simulating it until you get an answer.

We will treat heads as ’1’ and tails as ’0’ and simulate 10 coin tosses 100,000 times to see how often we get more than 3 heads in 10 tosses.
First, we decide how many times to run the simulation:
>runs <- 100000
Next, we define a single round of simulating 10 coin tosses and returning “TRUE” if there are more than 3 heads in a given simulation:
>one.trial <- function(){
sum(sample(c(0,1),10,replace=TRUE))>3
}
Lastly, we repeat the trial ‘runs’ times:
>mc.binom<-sum(replicate(runs,one.trial()))/runs
We can now compare the result of our simulation to the following built-in function which

captures the cumulative distribution of the binomial for 3 or fewer heads:
>pbinom(3,10,0.5,lower.tail=FALSE)
In fact, if you don't specify the lower.tail argument, you will get 1-the value produced by your created simulation (so it will be 1-mc.binom).

4.4 The Binomial Test and Testing for Proportions:

A binomial test compares the proportion of individuals in a sample that have some quality to a hypothesis about that proportion. The data set called "bumpus.csv" contains data on house sparrow (*Passer domesticus*) population caught in a wind storm in the 1880's. One hundred-thirty-six dead or moribund were brought to Professor Herman Bumpus. He divided the birds into those that were dead and those that were alive when he received them. He then took various data points from the skeleton of each bird (weight, wing span length etc). Yes, this means that he killed the ones that were alive when he received them and skeletonized them. He found differences in shape and size between the survivors and the non-survivors and he published all of his data. These data were among the first to demonstrate natural selection, in fact, stabilizing selection, in a wild population.[1]

As always, lets start with a tiny bit of data exploration and add to our ever-growing list of useful **R** commands. The bumpus data will be re-visited in recitation 8 when we investigate non-parametric statistical methods so it will be particularly useful to have a sense of what the data look like overall. Famously, the Bumpus dataset was used to demonstrate stabilizing selection: selection which operates on the extreme values of both tails on the curve of variation is called stabilizing, or normalizing selection, because under this mode of selection, the average value of the trait does not change and too small or too large individuals have a lower survival rate. Use a **boxplot** (remember that the commands for this can be found in Recitation #3) to see if any of the characteristics measured by Herman Bumpus have different distributions between the survivors and the dead birds.

Since birds are sexually dimorphic, and will therefore have different expectations for size etc, pick one sex at a time to analysis. **R** allows you to choose a subset of your data to analyze at a given time; you can use this argument to only choose the males or the females for any particular analysis. For example, let's say you decided to focus on the skull width of each bird and you wanted to see if there was a difference in the tot skull widths of birds who survived (at least until Bumpus got his hands on them!) and those who died. You might use the command:

>boxplot(skull.width.in.∼survival,col = "Dark Red")

And then you would to overlay a title onto this graph with the following function:

[1]In fact, the data has been re-analyzed in a more sophisticated manner (using various 'modern' methods such as principle component analysis) and stabilizing selection was found to act on female size and directional selection acted on male size.

```
>title(main="Are skull width and survival related?")
```

You realized that you would like to pull out the boxplots for the males and for the females. You might use the subset argument of the >boxplot function and specify that the column "sex", and a particular value ("m" or "f"), should be used as a method of sorting the data:

```
>boxplot(Total.length.mm. survival, subset=sex=="m")
>boxplot(Total.length.mm. survival, subset=sex=="f")
```

You could then compare, visually, the boxplots of the two sexes for whatever trait(s) you are particularly interested in. The "==" stands for equivalency, or more colloquially, equals. Since using only one "=" assigns a value to a vector, we need to differentiate between testing whether or not two conditions are equivalent and assignment which is why we use two equal signs for testing equivalence. For other comparisons, you can use the other boolean operators such as ">" or "<" on other, non-string (string in this context means words or letters) variables. There are two more important and ubiquitous Boolean operators: '&' which stands for AND and "|" which stands for OR.

For comparison sakes', you may wish to put multiple boxplots on the same page/graph by using the following style of command (once you have these arguments in your data set):

```
>boxplot(death,female;survival, female;death,male;survival,male)
```

or

```
>boxplot(skull.width.in interaction(survival, sex),data=bumpus,main="Survival by sex and skull width")
```

4.4.1 The "tapply" function

If the trait you have chosen to examine produces boxplots that are only visually subtly different, you will want to use a second very useful command: **>tapply()**. **tapply** is an important command which allows you to apply a specific criteria, for instance female or male, to a vector or data frame (you can think of it as sorting a column in excel based on specific criteria) that you are calculating a function on. It is another method of teasing out the data in column into distinct categories. But that is a little abstract, isn't it? Here is an example to make it easier to understand. If visual inspection of your trait isn't satisfactory to answer whether or not there is a difference between the two sexes, you can look at the summary data (median, mean, variance). However, you still have the same problem as before: the males and the female trait measurements are all jumbled together in the same column.

If you wanted the median total length of all of the birds, you would use the command:

>median(Total.length.mm.)

BUT if you wanted the median total length of the birds sorted out by the category of sex, you would use this command:

>tapply(Total.length.mm., sex, median)

This will give you two medians for the total length of the bird: the median length for male birds and the median length for female birds. You can use **tapply** for mean and variance as well (as well as many other functions). Using any one trait (length of femur, skull or whatever trait strikes your fancy) is there any indication of stabilizing selection among the house sparrows? Do the survivors of the storm have less variance than the non-surviving? Was this true equally for either sex?

The **tapply** function is a member of the so called 'loop functions' which are based on the **apply** function which allows you to, funnily enough, apply a specific function - like mean etc - across data points such as ones in a particular column of a dataframe. Other members of this group include **lapply**, **vapply** and **sapply**. There are subtle differences between these commands but, mostly, they use different types of data as input (vectors or matrices etc).

Now that we have spent some time doing data exploration with this data let's try to run some tests on the data. For proportions, we have two major tools at our disposal: **the binomial test** and **the proportion test**. But first, we need to obtain counts of the four categories of interest, female survivors, female non-survivors, male survivors and male non-survivors. For this we will use the **>sum()** function along with two conditional statements.
To count how many females survived: **>sum(survival==TRUE&sex==“f”)**
To count how many females did not survive: **>sum(survival==FALSE&sex==“f”)**
To count total survivors, you remove the condition that the survivors are a particular sex: **>sum(survival==TRUE)**

4.5 Activity:

Write out the 4 steps of hypothesis testing to answer the question: Were male and female birds equally likely to have died in the storm?

1. **Step 1: Formulate the Null hypothesis**. In this case, you would re-frame the question to be the H_O: are 50% of survivors female? To unpack why we use p = 50% note that you can, instead, ask:H_O: are 50% of birds sampled by Dr. Bumpus (brought to him by others) female? We expect that in the entire population of house sparrows, approximately 50% will be female so why were only 36% of the sample female? Was it due to variation in one of the measured characteristics between male and female house sparrows?

2. **Step 2: What is the test statistic distribution that reflects the null hypothesis? Use this to conduct the test of the H_O**: for this H_O, the test statistic distribution is the binomial distribution, you can then use the function **binom.test()**. In this case, the successes, x, are the female survivors - 21- and the number of trials, n, are the total survivors, both male and female - which is 72. It is a bit of a strange way of seeing it, I admit, but we could phrase the question differently and interpret the results in a similar fashion.

3. **Step 3: What is the p-value or other significance value/critical value of the results of the test?**

4. **Step 4: Conclude and, usually, include a confidence interval with your conclusion from the test that you have run.**

>binom.test(21,72,p=0.5,alternative="two.sided",conf.level=0.95) or
>binom.test(51,72,p=0.5,alternative="two.sided",conf.level=0.95)

There is (as always with **R**) another way of testing for proportions. For larger data sets, you can use the **>prop.test()** whose command is give below. The **prop.test()** uses a estimation procedure instead of calculating values exactly, so, if your data set is reasonably small, it is usually best to use **binom.test()** instead.

>prop.test(21,72,p=0.5,conf.level=0.95)

What is the difference in p-values between these two methods?

4.6 ROC and AUC:

ROC (Receiver Operator Curves) and AUC (Area Under the Curve):

ROC are efficient ways to assess * the fit of many models simultaneously under specific criteria where the key summaries of the model are visualized in one place.* Basically, ROC should help you determine which models fit your data better and allow you to choose the best model. ROC can be used for ANY **binary classifier**, where there are TWO outcomes (logistic regression, Odds Ratio or other model where there are TWO outcomes). Accompanying the ROC is usually the AUC (area under the curve) value which provides a single number summarizing the performance of a particular model. You can graph the ROC and determine the AUC ? and the corresponding confidence interval ? for each model under consideration to decide which model is superior to another model.

The drawback of ROC curves is, of course, that some of these key summaries can be challenging to understand initially. Take a moment to understand the following important terms, TP (True Positive), TN (True Negative), FP (False Positive), FN (False Negative). Wikipedia has clear resources that summaries what each of these means and gives several example tables of them. Since ROC and AUC (Area under Curve) are used in fields ranging from engineering, precision medicine, and genomics, I highly recommend that you understand crucial basic terms such as sensitivity, specificity, type I, type II errors, and Power and, of course, the relationships between them all! `https://en.wikipedia.org/wiki/Receiver_operating_characteristic`.

	Disease	No Disease
Positive Test	True Positive(TP)	False Positive (FP)
Negative Test	False Negative (FN)	True Negative (TN)

Sensitivity (same as Power): $TruePositiveRate = P(+test|Disease) = countTP/(countTP+countFN)$

Type I error: $FalsePositiveRate = P(+test|NODisease) = countFP/(countFP+countTN)$

Specificity: $TrueNegativeRate = P(-test|NoDisease) = countTN/(countFP+countTN)$

Accuracy: $countTP+countTN/(totalcountinentirepopulation) = (countTP+countTN)/(countTP+countFP+countTN+countFN)$

Prevalence: True # of individuals with condition in a population (usually estimated from population surveys)

There are several libraries that you might want to become familiar with (including G-Wiz which appears to focus on genomic information), but we will currently focus on an older and established library called **pROC** . This includes a data set that investigates 6 factors that contribute to the outcome, good or poor, six months after 113 patients experienced a Aneurysmal subarachnoid Hemorrhage (aSAH) data. If you are interested, you can find the paper here: `https://link.springer.com/article/10.1007/s00134-009-1641-y`. The six predictors of outcome are:

1. **gos6** -Glasgow score at 6 months
2. **WFNS**- World Federation of Neurological Surgeons score when patient was admitted
3. **Gender**
4. **Age**
5. **s100b**-calcium binding protein which is a biomarker the blood brain barrier (since it is glial specific) and Central Nervous System. Elevated levels suggest Nervous System damage.
6. **NDKA** -Nucleoside Diphosphate Kinase A. This is another brain specific biomarker/enzyme
7. **outcome**- good or poor at 6 months after event

Chapter 5

Goodness of Fit Tests

5.1 Goals

- Test for the independence of two categorical variables using χ^2 **contingency test**.
- Learn how to easily use **>mosaicplot()**

5.2 χ^2 Goodness-of-fit test Activity

(Adapted from Whitlock and Schluter lab exercises) We'll do an experiment on ourselves. The point of the experiment needs to remain obscure until after the data is collected, so as to not bias the data collection process, so please don't peek at the end of the lab just yet! The information that you provide (for yourself) will be the basis of the first question on your problem set.

After this paragraph, we reprint the last paragraph of Darwin's Origin of Species (which has the most beautiful final sentence of any book). Please read through this paragraph, and circle every letter "t". Please proceed at anormal reading speed. If you ever realize that you missed a "t" in a previous word, do not retrace your steps to encircle the "t". You are not expected to get every "t", so don't slow down your reading to get the "t"s.

It is interesting to contemplate an entangled bank, clothed with many plants of many kinds, with birds singing on the bushes, with various insects flitting about, and with worms crawling through the damp earth, and to reflect that these elaborately constructed forms, so different from each other, and dependent on each other in so complex a manner, have all been produced by laws acting around us. These laws, taken in the largest sense, being Growth with Reproduction; inheritance which is almost implied by reproduction; Variability from the indirect and direct action of the external conditions of life, and from use and disuse; a Ratio of Increase so high as to lead to a Struggle for Life, and as a consequence to Natural Selection, entailing Divergence of Character and the Extinction of less-improved forms. Thus, from the war of nature, from famine and death, the most exalted object which we are capable of conceiving, namely, the production of the higher animals, directly follows. There is grandeur in this view of life, with its several powers, having been originally breathed into a few forms or into one; and that, whilst this planet has gone cycling on according to the fixed law of gravity, from so simple a beginning endless forms most beautiful and most wonderful have been, and are being, evolved.

5.3 χ^2 Contingency Test

Now we are going to use **R** to run contingency analysis. Below find lifestyle data on individuals' smoking and exercise habits. We want to investigate the following question: *Are smoking and a commitment to frequent exercise independent events?*

The R code for conducting contingency tests has the following main steps:

1. **Feed in simple data (four categories of smokers and two categories of exercisers)**

Smoking Level	Frequency Exerciser	Seldom Exerciser
Heavy Smoker	7	4
Frequent Smoker	9	8
Occasional Smoker	12	7
Never Smoke	87	102

There are a number of ways you can produce this table in **R**. Below are two methods: in the first, you read in the above table as a matrix (which is simply a two-dimensional vector of numbers) and in the second, you convert it from a matrix into a table (which allows you to add names to the columns and rows). It is probably not clear to you with this simple example but a table is particularly useful when you have categorical data. What exactly is table format? So far we have mostly dealt with **data.frames** since they are very user-friendly. We haven't really seen tables yet (except when we looked at the built in data sets Titanic and HairEyeColor) because they are less forgiving; the columns need to have exactly the same length, for instance. As another aside: so far we have used the function **read.csv()** exclusively to import data.frames. There is a very similar function for tables called, oddly enough, **read.table()**. The real difference between these two functions is that **read.csv()** has the default argument for header set to TRUE whereas **read.table()** does not.
For this example, you are ultimately going to use the **>chisq.test** function, which is robust so you can use either method (I have included both for completeness). However you choose to coerce your data into a table format, you will want to have your data in table format so that you can graph a mosaic plot eventually (outlined in the next section).

   (a) **Simplest method (creates a matrix):**
   - Enter the data as a matrix: **>variable_name = c(7,9,12,87,4,8,7,102)**
   - Impose an order onto the data. In this case, you want the first four entries to be in the first column and the next four to be a second column (there are 4 rows and 2 columns) such as: **>dim(variable_name)=c(4,2)**

- Then, check data: **>variable_name**. This should have produced a matrix of the data. You can also use the built in function **>class** which will confirm that this is a matrix.

(b) **Method which creates a contingency table of counts:**

- Read in data as a matrix which has two columns. Note the order that the data is entered column 1 followed by column 2 like so:
  >variable_name <-matrix(c(7,9,12,87,4,8,7,102), ncol=2)
- Name Columns and Rows:
  >colnames(variable_name)<-c("Frequent Exercise","Seldom Exercise")
  >rownames(variable_name)<-c("Heavy","Frequent","Occasional", "Never")
- Tell **R** it is a table: **>variable_name <-as.table(variable_name)**
- Check your input: **>variable_name**
- Check the class of your input to ensure that it is a matrix: **>class(variable_name)**

(c) **you can also use the rbind function! to add rows to a matrix**

(d) **Other useful functions - remember there are always multiple ways of doing things in R - include: as.matrix(), read.table(), attributes()**

2. **Run the χ^2 function:** Now that you have created your table, you can run the χ^2function with the **chisq.test** function. Remember to practice good programming habits by assigning the results of chi-square test to a variable so you will be able to call that variable later. For example, **>variable_name.chi=chisq.test(variable_name)**
You should always check the results produced by the chisq.test function:
>names(variable_name.chi)
Or, if you forgot to park your chisq.test result into a variable, you can also just type: **>names (chisq.test(variable_name)** But it's messier right? It is also more difficult to keep track of when you are conducting multiple tests. So get into the good programming habit of parking of putting your results into a new result variable. Both of the above commands will give you the same thing:

```
[1] "statistic" "parameter" "p.value" "method" "data.name" "observed"
"expected" "residuals"
```

Now you can access other components of χ^2 test and see how much confidence about whether or not the data was extremely different from our expectations.
First let's check that our observed was the table that we inputted:
>variable_name.chi$observed
Does that give us our original table of observations? Good. Now what are the

expected values of this table under out null hypothesis that there is no relationship between the amount of smoking and exercising. **>variable_name.chi$expected** We can see what the p-value for the test is by typing: **>variable_name.chi$p.value**. Now that we already know what the answer is (since we know the p-value), let's see if a visual/graphical answer supports the p-value.

3. **To graphically display our results, we can use the mosaicplot function:**

   >mosaicplot(variable_name)

   A quick aside about Mosaic plots:
   It is usually quick and easy to tell from a mosaic plot whether the two variables are independent. Note that the mosaic treats the row and column variables differently. The column variable is treated as a conditioning variable, a "treatment", and the row variable is treated as a "response". Each column is a histogram, with bins stacked on top of each other instead of side by side. Please note that in order to take advantage of **mosaicplot()**, you need to ensure that your data is in table format. For the example given above: **>mosaicplot(variable_name, main="Exercising and Smoking", col = c("Pink","Grey"))**

5.3.1 More Examples of how to draw a mosaic plot

1. Pre-loaded Titanic data set.

   (a) load the Titanic data from the **R** data library (this dataset comes pre-installed with **R**; it is Titanic with a capital "T"). **>data(Titanic)**

   (b) Check to ensure that you have loaded (and in a table format) it with the command: **>Titanic**

   (c) You can try the **mosaicplot(Titanic)** command now ... but the plot with contain more information than you are probably interested in and will look quite crowded: **>mosaicplot(Titanic, main="Titanic proportions")**

   (d) In order to effectively use the **mosaicplot** function, we have to organize the data within Titanic first based on sex and then on survival. Since "Titanic" sorts data based on the class of the passengers (information that we aren't interested in right now), we have to corral all the females (who survived or not and regardless of their class) and all the males (who survived or not and regardless of their class) into female and male bins/categories. This is a slightly different process than we saw in Recitation 4 with the bumpus data which wasn't in a table format and on which we used the function **tapply()** to sort by sex of the birds. The **tapply()** function expects an array (or dataframes) as input. We can use a related function, **apply**, which expects our data to be in a table format, to summarize the data

in our table. For instance, it may be useful to know how many men or women were present on the titanic: **>apply(Titanic, 2, sum)**. You can also ask for summaries of more than one of the dimensions of this table, for instance, try a command which should break down the survival of men and women: **>apply(Titanic, c(2,4), sum)**. Tables also have their very own commands; in order to sort based on a specified condition in a table, we use the function **margin.table** (remember you can use google or **help(margin.table)** to find out more about this function): **>mosaicplot(margin.table(Titanic,2))**
We have specified "2" in this table to indicate that we are summing over the second dimension in this four-dimension table (1 class; 2 sex; 3 age; 4 - survival) to get the total female and male passengers. You can see by the plot that there were more male passengers than female passengers.

(e) Now that we have a plot of the females and the males, we want to further break down which females survived and which did not and which males survived and which did not. Again we use **margin.table()**: **>mosaicplot(margin.table(Titanic, c(2,4)), main="Titanic Survival by Gender")**

(f) If you want to also create a plot which includes "class" as a category that will cut through sex then you can do the following: **>mosaicplot(margin.table(Titanic, c(2,4,1)), main="Titanic Survival by Gender and Class", Shade=TRUE)**

2. **Direct input data**
Input the following directly by using the matrix function. As before, you will need to specify how many columns the data that you are inputting will have so that they are broken up correctly. You also have to specify the fact that you are organizing the data by row. The numbers are placed into the matrix with every 4th item placed in a new row. (This is a bit tricky to understand so compare your input line to the resultant matrix: 51, 43, 22 are in the first row and 92,28,21 are in the second row.): **>smoke <-matrix(c(51,43,22,92,28,21,68,22,9),ncol=3,byrow=TRUE)**.
specify column names **>colnames(smoke) <- c("High","Low","Middle")**
specify row names: **>rownames(smoke) <- c("current","former","never")**
and specify that the table format should be imposed on the matrix data. **>smoke <- as.table(smoke)**
confirm that the data now is in table format:
>smoke which should give the following output:

```
        High Low Middle
current   51  43     22
former    92  28     21
never     68  22      9
```

Sometimes, it is useful to be able to combine the measurements from different factors or categories. There are a number of ways to get the marginal distributions using the **>margin.table** command. If you just give the command the table it calculates the total number of observations. You can also calculate the marginal distributions across the rows or columns based on the one optional argument: **>margin.table(smoke)** which should result in

[1] 356

or **>margin.table(smoke,1)** which should result in

```
current  former   never
    116     141      99
```

or **>margin.table(smoke,2)** which will result in

```
  High    Low Middle
   211     93     52
```

5.4 Fisher's Exact Test of Independence

Fisher's exact test is another way of testing whether or not two categorical variables and there outcomes are independent. It is more flexible that the χ^2 contingency test since it not burdened by assumptions about sample size and, in fact, it provides more accuracy when sample sizes are small.

One of the most common tests of speciation in biology is called the **McDonald-Kreitman test**. It aligns sequences of multiple species of an organism and counts the number of two categories of mutation between them. The first category is *synonymous sites:* meaning that due to the degeneracy of the genetic code the nucleotide substitution does not result in a different amino acid. The second category is *non-synonymous sites:* a change in nucleotide does result in the specification of a different amino acid.

Nucleotide sites that varied were also categorized as **polymorphic** - they varied within a species - and **fixed differences** - they did not vary within a species but did vary between species. The underlying hypothesis is that: *" In the absence of natural selection, the ratio of synonymous to nonsynonymous sites should be the same for polymorphisms and fixed differences."*

You will need to input your data as a matrix as you do for χ^2 tests. Here is an example of McDonald and Kreitman's original data that used three species of *Drosophila*:

	Synonymous	**Non-synonymous**
polymorphisms	43	2
Fixed	17	7

Now that we know how to input Matrix data in R, try to input the above data. The $H_o : \frac{2}{43} = \frac{7}{17}$.

When you are finished, you may use the command **>fisher.test(matrix_name)** to see the **pvalue** of the test. When I typed this into R and ran it, I got:

```
Fisher's Exact Test for Count Data

data:  mk
p-value = 0.006653
alternative hypothesis: true odds ratio is not equal to 1
95 percent confidence interval:
  1.437432 92.388001
sample estimates:
odds ratio
  8.540913
```

This suggests that the null hypothesis of independence between mutations that changed amino acids or those that didn't and sites that varied within a population or were fixed between species can be rejected; there is significant difference in synonymous/nonsynonymous ratio between polymorphisms and fixed differences. Be careful when interpreting the **odds ratio** that fishers exact test returns; it is based on a conditional maximum likelihood estimate and isn't the same thing as the odds ratio that we can calculate by hand - it should be somewhat close, though.

In this case, the H_0 is $(43/2) = (17/7)$. However, fisher's exact test allows us to reject that null hypothesis and returns the ratio $(43/2)/(17/7)$. This means that there are 8.5 more synonymous mutations within a species than between two species. We could just test the ratios of $D_n/D_s = 7/17 = 0.41$ and compare that to $P_n/P_s = 2/43 = 0.047$. Since $D_n/D_s > P_n/P_s$, this suggests that positive natural selection is exerting influence. If $D_n/D_s < P_n/P_s$ would have suggested purifying selection.

5.5 Answer key to χ^2Goodness-of-fit test Activity

It is interesting to contemplate an entangled bank, clothed with many plants of many kinds, with birds singing on the bushes, with various insects flitting about, and with worms crawling through the damp earth, and to reflect that these elaborately constructed forms, so different from each other, and dependent on each other in so complex a manner, have all been produced by laws acting around us. These laws, taken in the largest sense, being Growth with Reproduction; inheritance which is almost implied by reproduction; Variability from the indirect and direct action of the external conditions of life, and from use and disuse; a Ratio of Increase so high as to lead to a Struggle for Life, and as a consequence to Natural Selection, entailing Divergence of Character and the Extinction of less-improved forms. Thus, from the war of nature, from famine and death, the most exalted object which we are capable of conceiving, namely, the production of the higher animals, directly follows. There is grandeur in this view of life, with its several powers, having been originally breathed into a few forms or into one; and that, whilst this planet has gone cycling on according to the fixed law of gravity, from so simple a beginning endless forms most beautiful and most wonderful have been, and are being, evolved.

Chapter 6

The Normal Distribution

6.1 Goals

- See the Central Limit Theorem in action.
- Visualize properties of the normal distribution.
- Calculate sampling properties of means.

6.2 Exploring the implications of the Central Limit Theorem

- **Distribution of sample means** The applet for this lab is found `http://onlinestatbook.com/stat_sim/sampling_dist/index.html:` We want to investigate three claims about the Central Limit Theorem since these properties are often invoked to ensure that we have a normal-enough distribution (normal-enough meaning that the distribution approximates a normal distribution enough to allow us to apply our statistics tests, like the t test):

- ***Claim 1: The distribution of sample means is normal, if the variable itself has a normal distribution.***

  First, hit "Animated" a few times, to remind yourself of what this applet does. It makes a sample from the distribution shown in the top panel. The second panel shows a histogram of that sample, and the third and fourth panels show the distribution of sample statistics from all the previous samples. You can control the sample size (N = 2, 5, 10, 16, 20 or 25) and which statistics you are interested in examining (Mean, median, sd, variance etc). You can examine, for instance, what happens if you set the third panel to be mean with sample size of 2 and the fourth panel to be mean with sample of 25.

  Next, hit the "10,000" button. This button makes 10,000 separate samples at one go, to save you from making the samples one by one. Look at the distribution of sample means. Does it seem to have a normal distribution? Click the checkbox by "Fit normal" off to the right, which will draw the curve for a normal distribution with the same mean and variance as this distribution of sample means.

- ***Claim 2: The standard error of the distribution of sample means is predicted by the standard deviation of the variable, divided by the square root of the sample size***

  The standard deviation of the population distribution in the top panel is given in the top left corner of the window, along with some other parameters. The sample size is set by the pull-down menu on the right of the sample mean distribution. (The default when it opens is set to N=5.) For N=5, have the applet calculate 10,000 sample means as you did in the previous exercise. If the sample size is 5 and the standard deviation is 5.0 (as in the default), what do you predict the standard deviation of the sample means to be? How does this match the simulated value?

Change the sample size to N=25, and recalculate 10,000 samples. Calculate the predicted standard deviation of sample means, and compare it to what you observed.

- ***Claim 3: The distribution of sample means is approximately normal no matter what the distribution of the variable, as long as the sample size is large enough. (The Central Limit Theorem)***

  Let's change the distribution of the variable in the population. At the top right of the window, change "Normal" to "Skewed." This will cause the program to sample from a very skewed distribution. Set N=2 for the sample size, and simulate 10,000 samples. Does the distribution of sample means look normal? Is it closer to normal than the distribution of individuals in the population?

  Now set N=25 and simulate 10,000 samples. How does the distribution of sample means look now? It should look much more like a normal distribution, because of the Central Limit Theorem. (Explain in your own words why this is so.)

  Finally, look at the standard deviations of the distribution of sample means for these last few cases, and compare them to the expectation from the standard deviation of individuals and the sample size. If you want to play around with this applet, it will let you draw in your own distribution at the top. Just hold down the mouse button over the top distribution, and it will let you paint in a new distribution to use. Try to make a distribution as non-normal as possible, and then draw samples from it.

6.3 Testing for Normality

One of the best ways to know whether a population is approximately Normally distributed is to plot a **histogram** of the data. If the sample size is large and the data look even approximately normally distributed, then that will be close enough for most statistical and exploratory purposes. This is because many of our commonly used statistical tests are robust to small violations of the assumptions of each test. In a couple of weeks, in lab #8, we will investigate non-parametric statistics; these are statistics that we use when our data is not normal. Many of these statistical methods (and certainly any future publication of your own!) will require a more formal and exhaustive look at normality (or lack of normality).

A quantile plot is an excellent starting place to investigate normality. A **Q-Q** plot (which stands for Quantile-Quantile plot) compares two distributions by plotting their quantiles against each other; if the two distributions are similar, the comparison points should result in a straight line through **y=x**. If the two distributions are not the same but are related in a linear manner then they should lie on a different line. A normal quantile plot is a cumulative frequency plot that has a Y axis scaled so that a normal distribution will appear as a straight, diagonal line. A Normal Q-Q plot works in the same way but it compares the distribution of your data to a general normal distribution. This means that if your data is normally distributed (or close), than the two distributions are equal and so the plotted points should lie on a straight line. These curves are useful in assessing how and where a non-normal distribution differs from the ideal. Exponential relationships will appear as curves, while skews will cause the quantile plot to fall above or below the ideal straight line. If your data isnt normally distributed, the Q-Q normal plot should give some insight into whether or not it is a linear combination of normal distributions or something more sinister altogether. In other words, the Q-Q normal plot will give you some insight into whether parametric methods could possibly be appropriate or if other tools (transformation or non-parametric methods) will need to be used.

To demonstrate how we use a quantile plot, we will do the following:

- First, we will simulate two distinct distributions in **R** so that you are able to visualize the difference between the normal distribution and a non-normal distribution. In order to see what normally distributed look like, you should attempt to randomly generate normally distributed data by using the function **>rnorm(n,mean,standard deviation).** To simulate a normal distribution use the command: **>variableName <-rnorm(100,2,5)**. This will create a normal distribution with 100 observations whose mean is 2 and whose standard deviation is 5. As always, double check: Use the command **>hist(variableName)** to ensure that you have in fact simulated what you think that you have simulated.

- Now create a non-normally distributed variable. A quick and easy example would be a **Poisson** distributed variable with a small sample size. Create a second variable using the following command: **>variableName_Pois <- rpois(100, 1)**. This will create 100 observations from a poisson distribution with a $\lambda = 1$. Once again, use the command **hist(variableName_Pois)** to ensure that you really have plotted the distribution that you think you created.

  Okay, so now we have two distinct samples: one from a normal distribution and one from a non-normal distribution. In this artificial case, we know that one is normal and one isnt but we will proceed to pretend that we dont know. So, how can tell? That is where the quantile plot comes in

- Start with the normally distributed variable. The command to generate a normal quantile plot is **qqnorm**. You can give it one argument, the univariate data set of interest: **>qqnorm(variableName for normal data)**. You can, of course, annotate this graphing function in the same way as the other graphing functions you have used so far: you can add a title, and clean up the axes, add colors etc. See **help(qqnorm)** to determine what can be customized within the function.

- After you create the normal quantile plot you can also add the theoretical line that the data should fall on if they were normally distributed: **>qqline(variableName)**.

This helps you to visualize just how far away from normality your data lay (which, in turn, allows you to determine if you just need a minor transformation of the data or if you need to take more drastic measures.). As noted in the above example, I find it particularly useful to distinguish this line from the data by assigning it a different color than the qqnorm plot points.

- Now do the same thing for the non-normal distributed variable: **>qqnorm(variableName_Pois)** and **>qqline(variableName_Pois)**.

  How do these two graphs compare?

Chapter 7

t Tests

7.1 Goals

- Compare the means of two groups
- Understand paired vs. two-sample designs

7.2 One Sample t tests

For an example, let's use data for a particular moose population, a northern population, where the average weight is 423 kg. A random sample of 9 moose was taken from an adjacent geographical area, a western population, and the following weights were recorded: 401, 380,393,450,420,435,426,397 and 415 kg.
We want to answer the following question: *Is the average weight of moose from the two different geographical locations the same?* When answering this type of question, as always, start by explicitly stating the **null** and **alternative** hypotheses:

$$H_0\text{: } \mu_0 = 423$$
$$H_A\text{: } \mu_A \neq 423$$

To do a one-sample t-test, you will use the function **t.test()**. This function contains a variety of options and will be used for two-sample t-tests as well. To call it, use the function: **>t.test(x,y=NULL, alternative=c("two.sided", "less","greater"), mu=0, paired=FALSE, var.equal=FALSE, conf.level=0.95)**. If y is excluded by not including it or assigning it the value NULL, the function performs a one-sided t-test on the data in x; if y is included it performs a two-sample t-test. **mu** is the true mean (or difference in means if you are performing a two sample test) under the null hypothesis. Remember that the basic assumptions of a **t-test** include a normally distributed random variable and no outliers. It is almost always a good idea to start off by testing your data to ensure that they fulfill these basic criteria (usually a graphical method, such as a histogram or qqnorm plot will suffice, followed by a more rigorous investigation if necessary).
You begin by inputting the moose weight data into a vector such as:
>moose_weight <-c(401, 380,393,450,420,435,426,397,415).
You can plot your data to ensure that it is generally 'mound shaped' and symmetric in a couple of different ways:
hist(moose_weight)
plot(density(moose_weight), main="Moose Weights")

Remember to continue to use the good programming habit of parking the results of the test into a new variable as in the following example:
>moose.t<-t.test(moose_weight, mu=423,var.equal=TRUE). In lab 8, we will investigate what happens when variances between two populations being compared are not equal but for now please ensure that you write var.equal=TRUE since the default of the **t.test()** function is var.equal=FALSE. You can specify other components of the test such as

a confidence level of 0.99 instead of 0.95 by changing the default setting of conf.level=0.95. To refresh your memory about the default settings of a parameter function, you can always use the command: **>??t.test**

7.3 Two-sample *t*-tests (pooled t-test)

6 subjects are given a drug (treatment group) and an additional 6 subjects a placebo (control group). Their reaction time to a stimulus was measured in ms. We want to know if there is a difference in reaction times between the two groups.

```
Drug: 101,110,103,93,99,104
Placebo: 91,87,99,77,88,91
```

Let μ_1 be the mean reaction time of the treated group and μ_2 be the mean reaction time of the placebo group.

$$H_0\text{: } \mu_1 - \mu_2 = 0$$
$$H_A\text{: } \mu_1 - \mu_2 \neq 0$$

You are asking the question: *Is there a difference in reaction times?* The procedure to answer this question is the same as for the one sample t-test but you need to specify the y variable as well. So, for instance, you will need to create two vectors: one for the values of the Drug and one for the values of the Placebo as in the following:

- **>Drug<-c(101,11,103,93,99,104)**
- **>Placebo<-c(91,87,99.77,88,91)**
- **>drug_treatment.t<-t.test(Drug,Placebo,var.equal=TRUE)**
- **>drug_treatment.t**

Remember: When you include a y variable into the **t.test()** function, the default test is for whether or not the means of the two groups are equal. In the above example, we assumed that the two samples have equal variances since if we use the default var.equal=FALSE , the result is a slightly different version of the **t-test** being evaluated called the **Welch two sample test**. However, we haven't actually tested whether that is a legitimate assumption or not. We need to explore the two distributions.
To put both graphs side by side in the same window we use **par** and we specific that there is 1 row and 2 columns:
>par(mfrow=c(1,2))
>plot(density(Drug)), main="Drug")
>plot(density(Placebo)), main="Placebo")
Of course, we can also do a lovely boxplot for the two populations, too. Since boxplots

show the median value but **t-tests** compare the means of the two populations, we probably want to add the mean value on top of the boxplot.
>boxplot(Drug,Placebo),ylab="reaction times to stimulus", names=c("Drug","Placebo")
>means <- tapply(Drug,Placebo,mean)
>points(means,col="Red",pch=15)

7.4 power.t.test()

How big of a sample size do you need to pick up the difference in means between two populations that you would like? This difference is called, oddly enough, "Delta". There is a function to help you to determine that! However, you do need to calculate (or get R to summarize for you) the **pooled standard deviation** of your samples. Pooled standard deviation is calculated by summing the multiplication of the sd of each group by its particular degrees of freedom and then dividing this sum by the total degrees of freedom of all groups minus 2. You will also need to specify the power that you want to have in your test.
>power.t.test(delta=, sd=, sig.level=.05, power=, type="two.sample", alternative="two.sided")

7.5 Non-Parametric Alternative to two-sample t-test

If Normality is violated and the sample sizes are too small to leverage the central limit theorem then you will need to use the nonparametric alternative. For the Two-sample t-test, this is called the wilcox.text and it is a rank sum test.

7.6 Paired *t* tests

There are many experimental settings where each subject in the study is in both the treatment and the control group (ie. before and after). We can't use two-sample t-tests with such data because the assumption of independence of data points is violated. For a paired t-test, we use the **t.test()** function and include the option **paired=TRUE**.

A study was performed to test whether cars get better mileage on premium gas than on regular gas. Each of ten cars was first filled with either regular or premium gas, and the mileage for that tank was recorded. This scenario was repeated for the second type of gas for the next tank. Do cars get significantly better mileage with premium gas?
Regular: 16,20,21,22,23,22,27,25,27,28
Premium: 19,22,24,24,25,25,26,26,28,32

Chapter 8

Non-Parametric Tests

8.1 Goals

- **On-line simulation exercises: Explore the effects of violation of assumptions**
- **Simulate and compare normally distributed data and Poisson distributed data sets and data sets with different variances**
- **The often fruitless effort of transformation**
- **Non-parametric tests: Welch's approx. t-test and Wilcoxon test**
- **A common healthcare non-parametric test: Kaplan-Meier estimator and plot**
- **Simulating a p-value: an example**

8.2 Investigating robustness:

The **t-test** is fairly robust to its assumptions. This means that even when the assumptions are not correct, the t-test often performs quite well. Remember that the two-sample t-test assumes that the variables have a normal distribution in the populations, that the variance of those distributions is (approximately) the same in the two populations, and that the two samples are random samples.
Let's start with the following exercise: Open a browser and load the applet by clicking "Begin" `http://onlinestatbook.com/stat_sim/robustness/index.html`. This applet will simulate t-tests from random samples from two populations, and it lets you determine the true nature of those populations.
Start with a scenario that matches the assumptions of the t-test. Use the default parameters of Mean =0, standard deviation (sd) = 1, and no skew for both populations. (These values can be changed by the controls on the left hand side of the window.) You can leave the sample sizes at 5 (controls for the sample sizes are in the right side of the window.) Now click "Simulate" at the bottom. The computer will artificially create 2000 samples of 5 individuals each from both populations, and calculate the results of a two-sample t-test comparing the means. It tallies the numbers of significant and non-significant results at the bottom of the window. In this case, both distributions are normal and the variances are equal, just as assumed by the t-test. Therefore we expect it to work quite well in this case. (Take a moment to think about what "working well" means. If the null hypothesis is true, then the ***actual Type I error rate should match the stated Type I error rate, and if the null hypothesis is false it should be rejected as often as possible***.)
Now, look at the following cases, and describe the effects on the performance of the test.

1. **Unequal standard deviation, equal sample size** Make the standard deviations of the two populations unequal, with the first equal to one and the second equal to 5, with everything else like the defaults. (In particular, keep the sample sizes equal.)

2. **Unequal standard deviation, unequal sample size** Make the standard deviations unequal as in part a. (that is, 1 and 5), but make the sample size of the first population be 25, leaving the second sample with 5 individuals.

3. **Skew, equal sample size** Return to the defaults, except make both populations have "Severe" skew.

Play around with other parameter combinations.

8.2.1 Is it normal and are the variances equal?

When faced with real data, you will need to explore the data to see if you will be able to rely on the power of parametric methods (remember: if you are able to use a parametric method rather than a non-parametric method, it is almost always better to do so). Your particular strategy may be idiosyncratic but it is a good idea, in the interest of being somewhat efficient, to adopt the general approach of visual assessment (usually the easiest method and normally using a simple scatter plot or boxplot or histogram etc) followed by a narrowing down to formal tests (if your visual inspection suggests particular non-parametric pathologies of the data). To start off, let's remember that you can create a normal distribution and a non-normal distribution (such as a Poisson distribution) using the function: **>rnorm(n,mu,var)** and **>rpois(n,lamba)**. So, for example:

```
>Norm_dist<-rnorm(100,2,5)
>Poiss_dist<-rpois(100,1)
```

You now have two distinct vectors to test using the methods outlined below.

1. visual assessment (some useful choices):

   - Histogram
   - Boxplot
   - Normal Quantile Plot

2. formal tests: *is it normal?*. The Shapiro-wilk test for normality. The Shapiro-Wilk test uses the null hypothesis that the data is normally distributed. If it is not normally distributed, the null hypothesis is rejected. **>shapiro.text(x)**.

3. formal tests: *are the variances the same/similar enough?*. We will apply this test again next week when we cover ANOVA but, for now, we have two tests to choose from:

(a) **Bartlett test** this works even if the two distributions being compared are not normally distributed.

(b) **variance test**. **var.test(x,y,ratio, alternative, conf.level, data)**. This test will give you a ratio of the variances from the two samples BUT ONLY WORKS IF EACH POPULATION IS NORMALLY DISTRIBUTED!

You can see how **var.test** works by simulating two data sets that are normally distributed but have different variances:
>x<-rnorm(100, 0, 2)
>y<-rnomr(100, 0, 6)
>var.test(x,y)
If data are normal but the variances are not equal: .
Welchs t-test: >t.test(x,y) performs **Welch's** t-test automatically (by default). Since Welch's t.test works by adjusting degrees of freedom you don't want to use it by default if you know that your variables are normally distributed and have (approximately) equal variances. The **t.test** is fairly robust to violations of equal variance allowing up to 3 times the variance for one variable as the other (usually you need to have sample sizes of >30 for each population for robustness to be true). So if, you KNOW that your variables meet the pre-requisites for the parametric **t.test**, be sure to set the argument **var.equal = TRUE**.
If data are NOT normal:.
Transformation: In some cases a distribution will be more normal if the data are transformed before analysis, for example the most common transformation takes the log of each data point. To do so, simply take the **>log(yourdata)** within the function of interest. Almost certainly, you will want to run the **qqnorm()** function on the newly transformed data to see if it has become, via the transformation, into normally distributed data. Be careful, though - transformation is often not worth the effort! To explore the effects of transformation in use the applet at `http://onlinestatbook.com/stat_sim/transformations/index.html`. to explore the effects of transformations on the various data sets provided there. Click the transformation buttons on the side and bottom of each graph, and select alternative data sets from the menu button at the top.
Wilcox-signed-Rank test:
If you cannot find a transformation that allows you to use a parametric test, you will need to use a non-parametric alternative test. The most common function is: **>wilcox.test(x, y = NULL, alternative = c("two.sided", "less", "greater"), mu = 0, paired = FALSE, exact = NULL, correct = TRUE, conf.int = FALSE, conf.level = 0.95, ...etc)**. If only x is given, a Wilcoxon rank test is performed on the distribution of x. If both x and y are given and paired=TRUE, a Wilcoxon signed rank test of x-y is performed. If x and y are both given and paired=FALSE, a Mann-Whitney test is performed.

8.2.2 Estimating Survival Frequencies with Kaplan-Meier Curves:

There are a number of useful distributions that describe "time until a successful event", including common distributions such as the geometric distribution and the exponential distribution. These distributions are effective at explaining many biological problems and processes as long as their assumptions are met.

In health data, we often are interested in estimating the **time until death**. It is, admittedly, a bit morbid to label **death** as a **success**, but it is unambiguous state and so it is the convention. With many circumstances, researchers end up with so-called **right censored** datasets. That is, many individuals join a study, but then some drop out or, for leave the study for other reasons, and their information is no longer collected (that is why these are called "right" censored). There is a particular challenge that arises when individual data points in a dataset are incomplete. If these censored data points are excluded in the data analysis, the resulting estimator is biased, and that is a HUGE problem. However, our conventional parametric distributions and estimators rely on assumptions that are not met when data is incomplete.

So: what do we do?

Enter the Kaplan-Meier estimator. It is widely used (including in medicine) to describe the frequency of survival in particular time frames after a treatment is applied. That is, after chemotherapy to treat a particular type of cancer, a patient might have an X % of survival after Y months. The KM estimator is used more widely in biology than that small example, but survival after medical treatment is often where individuals first encounter the concept! You can get an overview of KM on Wikipedia: `https://en.wikipedia.org/wiki/Kaplan?Meier_estimator`.

The KM estimator has some things are a bit different from previously examined estimators. The two most important features are:

1. First: time frames are not (necessarily) uniform. So the survival proportion might be 7 days after treatment for the first time unit, but the second time chunk might be 3 weeks later (corresponding to a month after treatment).

2. Second: the survival probability in the first time unit is p_0. The probability of survival in the second time unit is conditional on surviving the first time unit, $p_1|p_0$. So the probabilities are not independent.

8.2.3 SIMULATING a p-value for data that violates normality:

This is another example from count bayesie: Let's suppose we're comparing two webpages to see which one convinces our customers to "sign up" at a higher rate. For page design A we have seen 20 convert and 100 not convert, for page design B we have 38 converting and 110 not converting. We'll model this as two Beta distributions (beta distributions are governed by two parameters and give us a lot of flexibility so they are used a lot when simulating random variables).

We could of course run a single tailed t-test, that would require that we assume that these are Normal distributions (which isn't a terrible approximation in this case). However we can also solve this via a Monte Carlo simulation! We're going to take 100,000 samples from A and 100,000 samples from B and see how often A ends up being larger than B.

```
>runs <- 100000
>a.samples <- rbeta(runs,20,100)
>b.samples <- rbeta(runs,38,110)
>mc.p.value <- sum(a.samples >b.samples)/runs
```

And we have mc.p.value = 0.0348 which means "Awesome our results are **statistically significant**"! But wait, there's more! We can also plot out a histogram for of the differences to see how big a difference there might be between our two tests!

```
>hist(b.samples/a.samples)
```

Chapter 9

ANOVA

9.1 Goals:

- Compare the means of multiple groups using Analysis of Variance.
- Make post-hoc comparisons with the Tukey-Kramer test

9.2 ANOVA

To perform ANOVA in R, there are a few major steps. First I will define the following terms that I use throughout this document:

```
DV = Dependent Variable; it is the response variable
Factor = what we are testing, ie. different treatments, medications etc.
```

The pollen of the corn (maize) plant is known to be a source of food to larval mosquitoes of the species *Anopheles arabiensis*, the main vector of malaria in Ethiopia. In 2010, malaria caused an estimated 660 000 deaths (with an uncertainty range of 490 000 to 836 000), mostly among African children. The production of maize has increased substantially in certain areas of Ethiopia recently, and over the same time malaria has entered in to new areas where it was previously rare. This raises the question, is the increase of maize cultivation partly responsible for the increase in malaria?

In this case, the (alleged) Dependent Variable is the rate of malaria infection and the Factor will be the low, medium and high cultivation of maize.

One line of evidence is to look for a geographical association on a smaller spatial scale between maize production and malaria incidence. The data set **"malariaMaize.csv"** contains information on several high-altitude sites in Ethiopia, with information about the level of cultivation of maize (low, medium or high) and the rate of malaria per 10,000 people.

9.2.1 Data exploration - checking assumptions of the test

First, we need to test our parametric assumptions:

1. Check normality of response variable at each level of the categorical variable:
   The easiest way to check for normality is to visually assess our data with a couple of straight-forward functions. In order to plot the relationship between maize production and the incidence of malaria, we can use
   - Boxplots
   - Histograms
   - shapiro-wilk test for each group or category in the ANOVA

Example: Remember to attach your data set!
>plot(IncidenceRate.10000~Maize_yield,data=maizeMalaria). You'll notice on this boxplot that that there is an outlier in the low cultivation of maize. Moving on.
You will want to use the shapiro wilk test that we saw in an earlier chapter but we want to apply it to each category that we are testing instead of all of the data pooled together. ANOVA requires that each category or group has data that is normally distributed. Instead of using the command **>shapiro.test(IncidenceRate.10000)** which would test the normality of the pooled data set, we break it down into each category via the following extremely useful command:
>by(IncidenceRate.10000, Maize_yield,shapiro.test).
Sadly, the data does not look normally distributed and ... there is a pesky outlier! Now what?

2. Check homogeneity of variance:
   In general, a straightforward test for equality of variance is if normality is demonstrated: **>bartlett.test(DV~Factor, data=name_of_file)**. Unfortunately in this case, this function isn't useful to us because it is sensitive to violations of normality; the data are not normally distributed.
   Another method is to draw a scatterplot of mean versus variance. It may not be intuitive why this is a useful graph but there are a number of statistical tests that allow for violations in homogeneity of variance *but only if the variance does not increases with increasing sample mean.* Basically, you don't want a 'funnel' shape. So even if the samples variances are not equal (heteroscedastic), if the variances don't increase with increasing mean than, depending on the test, that might be okay. We'll use our old buddy **tapply** to demonstrate:

   >plot(tapply(DV,Factor, mean), tapply(DV,Factor, var),main="Interesting title here!")

   This doesn't look particularly conclusive (there are only three points, after all) and the variance appears to increase with the mean. Sadly, I believe that we will need to move on to other options ...

3. If parametric assumptions not met:

   Can we use a transformation to get to parametric data? Try the two most common ones on the data: **log(variable)** or **sqrt(X +0.5)** and test to see if the transformations work to make the data more normal and homoscedastic.
   Example:
   >by(sqrt(IncidenceRate.10000+0.5), Maize_yield,shapiro.test).

   What about another quick boxplot using a slightly different command format (due to the transformation)?

>boxplot(sqrt(IncidenceRate.10000+0.5)~Maize_yield,data=maizeMalaria)

The data is starting to behave isn't it? But this transformation isn't quite enough. Let's try another transformation, we shall try log (which is the log function)?

Example:
>by(log(IncidenceRate.10000), Maize_yield,shapiro.test).

>boxplot(log(IncidenceRate.10000)~Maize_yield,data=maizeMalaria)

Okay! Now we have data that looks normal-ish enough. Maybe? Does it? That is the artistry of statistics; data is messy and sometimes you continue onwards with an analysis with the potential flaws of your assumptions in the back of your mind so that you may confess them to your audience in your conclusions and your discussion sections.

4. No, seriously: If parametric assumptions are really not met even by our transformation: If we can't transform the non-normality and variance heteroscedasticity away, we need to employ (*deep sigh*) the **Kruskal-Wallis nonparametric test**. Like all non-parametric tests, we give up power when we use it. We will try this on our data at the end of our analysis to compare it to what we got from our transformed data.

9.2.2 ANOVA test in R

As we use these functions, just remember: One-way ANOVA is, fundamentally, done the same way as linear regression (under the hood). Another piece of advice is that stylistically, as has been mentioned in previous chapters, it is best to name a new object to receive the results of the your ANOVA test. It just makes keeping track of everything much, much easier. So make sure that you put the suffix ".aov" onto our the receiving object of the function.

For a one-way ANOVA, you have various function choices but the most straight-forward option is **>aov(DV~Factor, data=name_of_file)**. This function is a 'wrap' function of **lm()**. In fact, it is often more useful to use **lm()** rather than **aov()** for one-way ANOVA as long as your design has balanced numbers of individuals. **lm()** refers to "Linear Model" function. In the case of our data set, which has required transformation, we can obtain an ANOVA table in the following manner:
>maizeMalaria_TRANS_log.aov<-aov(log(IncidenceRate.10000)~Maize_yield,data=maizeMalaria)

The summary function will spit out ANOVA table:
>summary(maizeMalaria_TRANS_log.aov)

9.3 Post hoc Test

If you are able to reject your null hypothesis (and in this case, to get practice, even if you aren't able to reject your null hypothesis) that all of your populations have equal means, in a so-called fixed effects model of ANOVA, you are now able to use the **Tukey honestly significant test** to determine *which* means are different from each other. The input for the Tukey function is the fitted model - so it is the results object created above (which has the suffix .aov).The function is **>TukeyHSD(result.aov)** and it works by taking pairwise comparisons of each of the factors (for instance, in the case of the malaria and the maize production, Tukey would compare the malaria deaths between the High-Low yields, Low-Medium yields and High-Medium yields and it would put upper and lower bounds on these treatments).

For example:
>TukeyHSD(maizeMalaria_TRANS_log.aov)

9.4 Nonparametric test:

We can now compare our results to what would have been produced if we were forced to rely on the non-parametric alternative to ANOVA, **kruskal.test(DV~Factor, data=name_of_file)**.

Example:
>kruskal.test(log(IncidenceRate.10000)~Maize_yield,data=maizeMalaria)

Finally, remember: we (ALWAYS!) conclude!

Chapter 10

Regression and Correlation

10.1 Goals:

- Test Assumptions about data set and, maybe, think about what these assumptions are important
- Calculate regression lines and correlation coefficients
- Test null hypothesis about slope using parametric tests
- Explore the non-parametric version of Pearson's test

This week we will look at two more powerful analyses that build on ANOVA: simple linear regression and simple linear correlation. Both of these are parametric tests and have some fairly rigorous assumptions that must be met. If we can meet these assumptions, correlation and regression can provide us with detailed information about the relationship between two variables. As usual, we will discover that our data does not meet the necessary assumptions but we will use it to demonstrate how the tests work in R and to compare the results to the non-parametric versions of the test (so we can see what happens when you use parametric tests on data that actually requires non-parametric versions of the tests).

Regression is used to describe a relationship (not necessarily linear but we will only investigate linear regression; see STT 216 for more) in which changes in an independent variable (X) cause predictable numerical changes in a dependent variable (Y). Regression allows us to predict Y if we know X by estimating a line of best fit. In contrast, correlation is a way to describe the strength of two variables that are associated with each other, that co-vary linearly together. It has no direct predictive ability and does not imply causation. Both of these methods, share a number of calculations but they, fundamentally, interested in showing different types of relationships. **Warning:** carrying out a regression analysis and finding a significant relationship does not necessarily mean that you have found a cause and effect relationship. Demonstrating a cause and effect relationship also requires proper experimental design, with appropriate controls to rule out other possibilities.

10.2 Linear Correlation

We'll illustrate the steps of using the program to do regression and linear correlation with the same data set: Example 17.1 from Whitlock and Schluter, which involves predicting the age of a lion from the amount of black on its nose pad so that trophy hunters can target lions of specific ages thereby reducing disruptions to a given pride of lions. In the data file, the age of lion is called **age**, given in years, and the proportion of black on its nose pad is called **proportion.black**. Load the data from the file "Nose_age.csv". Remember to **attach** it so you can call the columns independently! Make a scatter plot of the relationship between age and proportion.black. Remember that age is one variable (V1) and proportion.black (V2) is the second variable in the functions given below since there is not necessarily any dependency between the variables (ie. there is no clear dependent variable and independent variable).

To calculate the regression line (the line of best fit) and test the null hypothesis that the slope is zero, we follow a similar procedure as with ANOVA. In other words, we need to check that all of the assumptions of the Pearson correlation test are fulfilled.

1. **Check assumptions:**

   (a) **Bivariate normality of the two variables:**
   The joint XY population distribution should be bivariate normal. This situation only occurs when both individual populations (X and Y) are each normally distributed. We can use the graphical methods that we have already seen in previous chapters such as **boxplots** or **hist** to visualize any serious violations of bivariate normality (ie. if the distributions are highly skewed or have outliers present). Of course, remember that you can modify the **boxplots** to be whatever color you wish (it doesn't have to be "Blue" as it is in the example below). In some cases, a continuous variable (such as the ones in the lion example), for instance, a simple scatterplot will work better. For example, **>boxplot(V1~V2,pch=16,col="Blue")**. You can also use **qqplots** to plot the quantiles of two variables against each other. In order to ensure that each variable is normally distributed, you can plot each of them against a randomly generated normal distribution like so:
   >qqplot(V1,rnorm(32,mean(V1),sd(V1)))
   >qqplot(V2,rnorm(32,mean(V2),sd(V2)))
   It is even better to plot the two variables against each other since we don't just care if they are individually normally distributed, we want to ensure that they are bivariately distributed: **>qqplot(V1, V2)**. And then, of course, there is the **qqnorm** function that we can use on just one of the variables: **>qqnorm(V1)** **>qqline(V1,col="Dark Red")**.
   For a more formal test, consider also using the **shapiro.test(V1)** on each of the two variables. If the resulting pvalue is smaller than your specified α, you can reject the null hypothesis that your data set is normally distributed. Additionally, you can use the MVN package which contains an omnibus test that tests for bivariate normality and graphs a resultant contour plot (shown in rmd file). You can find it here: `https://cran.r-project.org/web/packages/MVN/MVN.pdf`

   (b) **Linearity of data points can be demonstrated via a simple scatterplot:**
   >plot(V1~V2) If the assumptions are met (or approximately met) then we can move on to actually testing the null hypothesis: H0: ρ=0 (the population correlation coefficient is equal to 0; there is no linear relationship between the two variables)

2. **The actual correlation test**

- If Parametric assumptions met (use Pearson's correlation): **>cor.test(∼V1+V2,data=dataset)**. The lion dataset will not meet the assumptions but you may wish to conduct a correlation test anyway for practice.
- If Parametric assumptions are not met and could not be met by transformation (that's right! More transformation!) then one of the non-parametric versions of correlation need to be used instead. Luckily, **R** differentiates between these methods simply by their names:
  (a) *Sample size: 7 <n <30 −>Spearman method*
  >cor.test(∼V1+V2,data=dataset, method = "spearman")
  (b) *Sample size: n >30 −>Kendall method*
  >cor.test(∼V1+V2,data=dataset, method = "kendall"). The Lion dataset has 32 datapoints so you should use Kendall. There are some values which are tied which will throw a warning but the results are still valid.

3. **Conclusion** You might want to superimpose the "best fit" line onto your scatter correlation plot. In order to do that, you need to fit a "general linear model". You will use the same methodology as outlined in the linear regression section (next up!) to do this. Once you have the general linear model, use **>abline()** command to superimpose the line onto your graph.

10.3 Linear Regression

The difference between correlation and regression is that in regression either one of the variables has been set (not measured) or there is an implied causality between variables (one variable could influence the other but the reverse is unlikely). In regression analysis there are two variables: one is the Independent variable (IV) and the other is the dependent variable (DV) whose value is hypothesized to be predicted by the independent variable. $DV = Y_{intercept} + \beta * IV + \epsilon_{ij}$ With regression $H_0 : \beta = 0$ (this means that we are testing that the true population slope is equal to zero). The first two steps are the same as linear correlation; the third step, homogeneity of variance, is new:

10.3.1 Check Assumptions:

1. Linearity
2. Normality of response variable
3. Homogeneity of variance: This term translates into variance that is equally scattered at any given value of the independent variable. You can visualize this via a residual plot, once you have fitted a general linear model to your data, where the residuals versus X (Independent Variable) should be a flat line. To produce such a plot, follow

the commands given below: **>Your_Chosen_Name.lm <-lm(DV~IV,dataset)**
>Your_Chosen_Name.resid<-resid(chosen_name.lm)
>plot(Your_Chosen_Name.resid)
>text(Your_Chosen_Name.resid, labels=Independent_variable_name, cex=0.4, pos=3)

If there is no obvious "wedge" pattern apparent in the residual plot (confirming that the assumption of homogeneity of variance is likely met) than we can move on. Remember that the residual plot = observed-predicted. This means that there should be a reasonable scatter above and below the "0" line.

10.3.2 Single response value for each level of the predictor variable

>Your_Chosen_Name.lm <- lm (DV~IV, dataset)

>plot (Your_Chosen_Name.lm)

>summary(Your_Chosen_Name.lm)

The **>summary()** gives you the intercept and slope of your linear model line among many other useful numbers.

10.3.3 conclusions

Compute confidence interval:

In order to calculate the 95% confidence intervals of the intercept of the regression equation and of the independent variable you use the **>confint** function:
>confint(Your_Chosen_Name.lm, level = 0.95).

The relationship of the independent variable and the dependent variable is nicely summarized by graphing the original scatterplot along with a superimposed **>abline** of the regression line and confidence bands:
>plot(DV~IV,data, pch=16, xlab = "Whatever", ylab ="Blah", main="Confidence Bands")
>abline(Your_Chosen_Name.lm, col="Red")
Producing confidence bands can be a little tricky. In an effort to keep this recitation to a reasonable length, I will skip over some of the underlying complications and just give basic instructions (if you are interested in understanding how, exactly, some of these commands work, I will refer you to google! Or the library to track down an **R** book! Or STT 278 that will be offered starting in Spring 2019!).
First we need the fitted values and their standard errors for all the observations. We use the **>predict()** function to obtain these and we will save them by placing them ever so carefully into an object. We can call this object whatever we wish but using a

clear name makes it easier to keep track of all the pieces. Let's name it "CI":
>CI <- predict(Your_Chosen_Name.lm, se.fit = TRUE)
We also need the Working-Hotelling multiplier (these bands are sometimes called the Working-Hotelling (1-α)100 Confidence bands). For a 95% confidence band, we use:
>W <- sqrt(2 * qf(0.95, 2, n-2)). In the case of the lion dataset, n-2 will be 30 since there are 32 data points.
There are just a few more steps. We need to have **R** calculate the upper and the lower bounds of the confidence band for each observation and we will save these calculations in another object. Let's call this one "Band":
>Band <- cbind(CI$fit - W * CI$se.fit, CI$fit + W * CI$se.fit).
To superimpose these lines on your scatterplot, do the following: If your estimated regression line has a positive slope, use the following commands, with the variable names modified as needed:
>points(sort(IV), sort(Band[,1]), type="l", lty=2)
>points(sort(IV), sort(Band[,2]), type="l", lty=2).

You get the confidence band enclosing the estimated regression line, as shown on the next page. You can save this plot, just as previously done with the original scatterplot.

If your estimated regression line has a negative slope, you need to sort the columns of Band in reverse order. So change the above **R** commands to:
>points(sort(IV), sort(Band[,1], decreasing=TRUE), type="l", lty=2)
>points(sort(IV), sort(Band[,2], decreasing=TRUE), type="l", lty=2)

Chapter 11

General Linear Models

11.1 Goals:

- ANOVA, Correlation and Regression all rely on minimizing and partitioning SS (Sum of Squares). Since they all feature similar calculations and have a similar goal, they can all be considered specific subsets, dependent on the variable data types, of the general linear model
- We usually take multiple measurements of many variables; we need to be able to summarize and display important patterns and relationships in the variables.
- I have included a general overview of each method and there is an accompanying .rmd file which works through an example. However, for more specific information about each method, I have included links that I believe clearly demonstrate particular concepts with specific examples.

11.2 General Assumptions of General Linear Models

We will assume that our data is balanced (the same number of data points in each factor), normally distributed, has equal(ish) variances and that the dependent variable (**DV**) enjoys a linear relationship with the independent variable(s) (**IV**). Earlier recitations, especially #9 and #10, went through how to test for normality and equal variances so we won't focus on that in this chapter (although you should always test these assumptions when you begin to explore your own data sets).

If you have data that violates any of the above assumptions then you will need to track down a more advanced guide on how to handle "mixed models". This chapter is a guide that is intended to provide a foundational knowledge for tackling these types of analysis and it is not exhaustive! The following link provides a reasonable guide for more complex modeling (note: there are MANY more such guides available at your googling fingertips). `http://ase.tufts.edu/gsc/gradresources/guidetomixedmodelsinr/mixed%20model%20guide.html`.

11.3 Blocking

On top of parsing out the effects of the variable of interest and the response variable, good experimental design often demands the presence of blocks. Blocks are often used to account for likely sources of confounding variables (maybe differences in response to a treatment by men and women or between hospitals that are well-funded and those that aren't, for example) thereby reducing the undesired effects on variation of response by extraneous elements. Including a block should account for some of the variance in the experiment and has the effect of turning down the 'noise' to better see the true treatment 'signal'. You can read a bit about blocks at wikipedia: `https://en.wikipedia.org/wiki/Blocking_(statistics)`.

Blocking forces you to include an additional variable, the block, which must be accounted for in analysis. There are a number of ways of handling blocks but the ideal way is with a randomized block design where each block contains all of the treatments. There are other ways that often require any interactions between the variables and the block to also be included in the analysis. We'll focus on an example of a randomized block design, where every treatment is present once in a particular block, since there is no need to include an interaction term in this scenario and it will make our analysis a little more straight forward (we'll see interaction terms in the next couple of sections). The basic command for a randomized block design (remember: no interaction term needs to be included in the formula in that case) will look like this:
>aov(DV~IV1+Block, data)

We'll work through the ice cream example in the .rmd file that is associated with chapter 11. Just to remind you of ideas that we have seen in previous chapters:

- **aov** is a wrapper function of **lm**. **aov** fits a model and produces regression, coefficients, residuals etc. which means that sometimes you will see **aov** used or sometimes **lm** used interchangeably. For a beginner user of **R**, you can get by with thinking that these two functions are very similar (if you want to tackle slightly more sophisticated analysis with nested designs etc, then you should appreciate that there are some subtle differences between these two functions)

- **anova** is a function that analysizes the results of either **aov** or **lm** to produce more streamlined output such as an ANOVA table.

11.4 More than one factor: Multivariate

When you are interested in measuring the effect of two independent variables on a dependent variable, you can still use the **aov()** function. However, you will need to account for any potential interactions between the two independent variables by including a specific interaction term in your analysis. You can do this in two functionally equivalent ways using the **aov**:

>aov(DV~IV1*IV2, data) OR
>aov(DV~IV1+IV2+IV1:IV2, data)

Personally, I think the second format is more clear; since this format requires us to include an explicit value for the interaction between IV1 and IV2 in our analysis. For your own sake, it is always better to be as clear as possible about your analysis if, for no other reason, than your future self might better understand what you were trying to do with your analysis (don't ask me how I know this). You can then call the **summary()** on the resultant aov object. Keep in mind that the **aov()** function fits a very particular design type: type I, sequential sum of squares for balanced design. This jargon means that the IV1 factor is tested first, followed by the IV2 next (conditioned on IV1) and then, lastly, the interaction term, between IV1 and IV2, is tested. Due to the fact that the factors are tested in a particular order, if the design if the experiment is unbalanced, i.e. there are a different number of data points being tested under different circumstances, then the order of IV testing - whether IV1 or IV2 is tested first - will influence the results.

You can get a quick visual confirmation of equal slopes (and therefore no interactions between the independent variables) with the following command (we'll see this function deployed in the .rmd file):

>interaction.plot(IV1, IV2, DV)

You will want to reverse which independent variable is plotted against which by also running the command:

>interaction.plot(IV2, IV1, DV)

We're going to cheat a bit - we don't this module to be an entire course unto itself (see STT 216, STT226W, STT 221W for more detailed information about the specifics of these types of models) - and incorporate multivariate analysis into our covariate analysis in the accompanying .rmd file. However, in case you are interested in more resources for working through a multivariate example, there is an excellent example at the link below (the r cookbook is a wonderful resource, in general):
`http://www.cookbook-r.com/Statistical_analysis/ANOVA/`.

11.5 Covariates (ANCOVA):

Analysis of covariance (ANCOVA) combines features of ANOVA and regression. It augments the ANOVA model with one or more quantitative variables, called covariates, which are related to the response variable. Covariates are a type of confounding variable included in the initial round of analysis. A standard example of a variable that qualifies as a covariate is S.E.S. (Social Economic Status), a factor that tends to influence the outcome of many other variables but is not independent (otherwise, if it was an independent variable, the multivariate analysis would be most appropriate). The covariates are included to reduce the 'noise' and allow for more precise measurement of the treatment effect.

Since ANCOVA models include numeric and categorical variables, the linear model uses a separate linear regression for each group of the categorical variable. Interaction terms between these two types of variables fit different linear regression slopes; otherwise the same slope is forced upon every group.

For example, if you have a dependent variable (**DV**), an independent variable (**IV**) and a covariate **A**, you need to include a term which allows for an interaction between the **IV** and the covariate **A**. This would be written in **R** as:
>z <-lm(DV~IV+A+IV:A, data=mydata)

There are **two rounds** of model fitting in ANCOVA. In the first round, the covariate is included to see if it's inclusion improves the fit of the model. The hope is that when you call the function **anova(z)**, the ANOVA table that results will show no significant interaction

between **IV** and the covariate **A** and you will be able to drop the interaction term from your analysis since it doesn't account for a significant amount of the variation. You can then proceed to the second round of model fitting in ANCOVA: run the same linear model without the interaction as a simple two way (or one way) ANOVA like so:
>z <-lm(DV~IV+A, data=mydata)

Assumptions: On top of the assumptions of all of the General Linear Models class, there are a few additional assumptions baked into **ANCOVA**: linearity between covariate and response variables and equal slopes of each treatment. As usual, you need to do some preliminary visual data exploration. A scatterplot of the response variable against the covariate, using clearly different symbols for each level of factor(s), is a good first step in confirming that there is a linear relationship between covariate and response variable. For instance, simply plotting out the **DV** versus the **IV** under each condition of the covariate should result in lines that have similar slopes. If you can confirm the assumptions then you can drop the interaction terms and proceed with a straightforward one way or two way ANOVA (depending on how many variables you have in your model).

There are a number of clearly written resources with ANCOVA examples which can be found at the following sites:
`http://www.stat.columbia.edu/~martin/W2024/R8.pdf`
`http://r-eco-evo.blogspot.com/2011/08/comparing-two-regression-slopes-by.html`

11.6 Markov Chain

If you can't use an out-of-the-box general linear model, then you need to move on to more computational methods. One of these methods utilizes Markov Chains but we won't go into details about them here. If you are interested, Count Bayesie has an excellent example of how to use Markov Chains for more complicated data analysis (he even references "deep learning" AKA which, until what seems like three seconds ago, used to be called "Machine Learning"):
`https://www.countbayesie.com/blog/2015/11/21/the-black-friday-puzzle-understanding-markov-`

11.7 Principles Component Analysis: PCA

PCA is enjoying a renaissance in the bioinformatics world for good reason: it provides a straight-forward way to assign the major drivers of variation (usually genetic, for our interests) in a system. Remember that partitioning out variance is the primary goal of general linear modeling so PCA fits right into this section! PCA works a bit differently than the other modeling we have seen; it determines the most variable axis by determining the largest eigenvalue of a data set (which is a linear combination of the variables). This happens to be the single most variable "component", due to .... rules of algebra. The associated eigenvector of this eigenvalue is then easily found. If you don't remember your linear algebra: this eigenvector (remember that vectors specify directions) creates a new axis for the data (rather than the X and Y axis we started with) which runs through the most variable part. Due to some fancy algebra which we will not attempt to review here, the second most variable dimension is exactly orthogonal (a term which means 90 degrees or at a right angle to) to the first.

Why is PCA useful? **It reduces the dimensionality of your data and identifies which axes are the most important in driving the variation of the characteristic under investigation**. It may be surprising to you that you can reduce the dimensionality of your data without losing much information. This occurs **when the variables are highly correlated**. When the correlation is weak, you will need more variables to account for all the variation in your data set. PCA allows you to start with a certain number of traits - of which some have correlated values- and to extract a smaller set of traits that are uncorrelated! It's magic! By removing the redundant variables, PCA captures which variables are actually foundational and important. In a general sense, you can think of PCA reducing the noise of your system and helping highlight which variables are actually important for the characteristic that you are interested in. Much like other general models we have investigated, PCA allows us to allocate the variability to axis that are useful for our particular data. There is an excellent explanation pf PCA by Dolph Schluter (the co-author of our textbook) here on the basic R commands for setting up PCA:
`https://www.zoology.ubc.ca/~schluter/R/multivariate/`

Remember the bumpus data that we analyzed in an earlier recitation? We are going to use a slightly simplified version of that data set to walk through an example of PCA analysis and discuss how it is useful.

First of all, there are two major PCA commands in R that you will need to know: **princomp** and **prcomp**. For the most part, we will use (**prcomp** but, since there are two examples of that function being implemented in your .rmd file, we will use the other PCA command **princomp** here (just to ensure that you have an example to work through, if you want): Let's start by going through the commands and adding any useful explanations along the way:
>sparrow <-read.csv(file.choose(), header=TRUE)
we will need to turn this dataframe into a matrix to make the analysis easier:
Now we are going to investigate summary values of our data set:
>summary(sparrow)
Looking at covariance matrix and the correlation matrix:
>S <-cov(sparrow)
>S
>sd.sparrow <-apply(sparrow,2,sd)
The "2" indicates that you are calculating the standard deviations of each of the 9 columns. In this case, the results suggests significant differences between the stand. deviations - and therefore also the variances - of the 9 variables. This means that it will be best to use a correlation matrix instead of a covariance matrix.

>sd.sparrow

```
> sd.sparrow
  Total_Length           Alar         Weight    Length_Beak Length_Humurus
    3.56083146     5.52102340     1.47521499     0.70237653     0.02307821
  Length_femur   Length_Tibia    Skull_Width    Keel_Length
    0.02411329     0.04074450     0.01499579     0.03964924
```

When the correlation between variables is high, we can reduce the dimensionality (the entire point of PCA) so we need to know what the correlation is like:
>R <-cor(sparrow)
>R

```
             Total_Length      Alar    Weight Length_Beak Length_Humurus
Total_Length      1.0000000 0.6909709 0.5838648   0.4694466      0.4846190
Alar              0.6909709 1.0000000 0.5686500   0.4990738      0.6779536
Weight            0.5838648 0.5686500 1.0000000   0.5192088      0.5188943
Length_Beak       0.4694466 0.4990738 0.5192088   1.0000000      0.6229937
Length_Humurus    0.4846190 0.6779536 0.5188943   0.6229937      1.0000000
Length_femur      0.4447051 0.5782836 0.4441451   0.6164174      0.8205803
Length_Tibia      0.3776146 0.5316798 0.4544589   0.5843728      0.7460385
Skull_Width       0.4355363 0.4338913 0.4714846   0.5347534      0.5120226
Keel_Length       0.5008898 0.5801525 0.5126353   0.4903663      0.5486094
               Length_femur Length_Tibia Skull_Width Keel_Length
Total_Length      0.4447051    0.3776146   0.4355363   0.5008898
Alar              0.5782836    0.5316798   0.4338913   0.5801525
Weight            0.4441451    0.4544589   0.4714846   0.5126353
Length_Beak       0.6164174    0.5843728   0.5347534   0.4903663
Length_Humurus    0.8205803    0.7460385   0.5120226   0.5486094
Length_femur      1.0000000    0.8092996   0.5212480   0.4536862
Length_Tibia      0.8092996    1.0000000   0.4586951   0.3840558
Skull_Width       0.5212480    0.4586951   1.0000000   0.3840089
Keel_Length       0.4536862    0.3840558   0.3840089   1.0000000
```

PCA can be carried out with the covariance or the correlation matrix. The usual rule of thumb is that if variances of the variables are very different, you should use the correlation matrix so that the variances don't disproportionately influence the initial couple of principle components. We will set the argument "cor" to TRUE for the princomp function since we will use the correlation matrix. If we set "cor" to FALSE, it would use the covariance matrix instead.

>pca_sparrow <-princomp(sparrow, cor=TRUE)

```
> pca_sparrow
Call:
princomp(x = sparrow, cor = TRUE)

Standard deviations:
   Comp.1    Comp.2    Comp.3    Comp.4    Comp.5    Comp.6    Comp.7    Comp.8
2.3046882 0.9988978 0.8128043 0.7306832 0.6783798 0.6362485 0.5210455 0.4616867
   Comp.9
0.3826387

 9  variables and  136 observations.
```

>names(pca_sparrow)

```
[1] "sdev"     "loadings" "center"   "scale"    "n.obs"    "scores"   "call"
```

>loadings(pca_sparrow)

```
Loadings:
               Comp.1 Comp.2 Comp.3 Comp.4 Comp.5 Comp.6 Comp.7 Comp.8 Comp.9
Total_Length   -0.310 -0.487         0.454  0.133  0.360  0.519 -0.131 -0.156
Alar           -0.351 -0.258 -0.338  0.256  0.242        -0.643  0.338  0.209
Weight         -0.316 -0.351  0.198        -0.704 -0.461        -0.101
Length_Beak    -0.336  0.115  0.293 -0.306 -0.326  0.737 -0.207
Length_Humurus -0.378  0.252 -0.222               -0.119 -0.251 -0.581 -0.567
Length_femur   -0.363  0.413 -0.143                       0.233 -0.312  0.715
Length_Tibia   -0.341  0.462 -0.152  0.138 -0.178 -0.131  0.301  0.634 -0.295
Skull_Width    -0.295         0.783         0.480 -0.236
Keel_Length    -0.302 -0.333 -0.226 -0.775  0.218 -0.158  0.247  0.112

               Comp.1 Comp.2 Comp.3 Comp.4 Comp.5 Comp.6 Comp.7 Comp.8 Comp.9
SS loadings     1.000  1.000  1.000  1.000  1.000  1.000  1.000  1.000  1.000
Proportion Var  0.111  0.111  0.111  0.111  0.111  0.111  0.111  0.111  0.111
Cumulative Var  0.111  0.222  0.333  0.444  0.556  0.667  0.778  0.889  1.000
```

>summary(pca_sparrow)

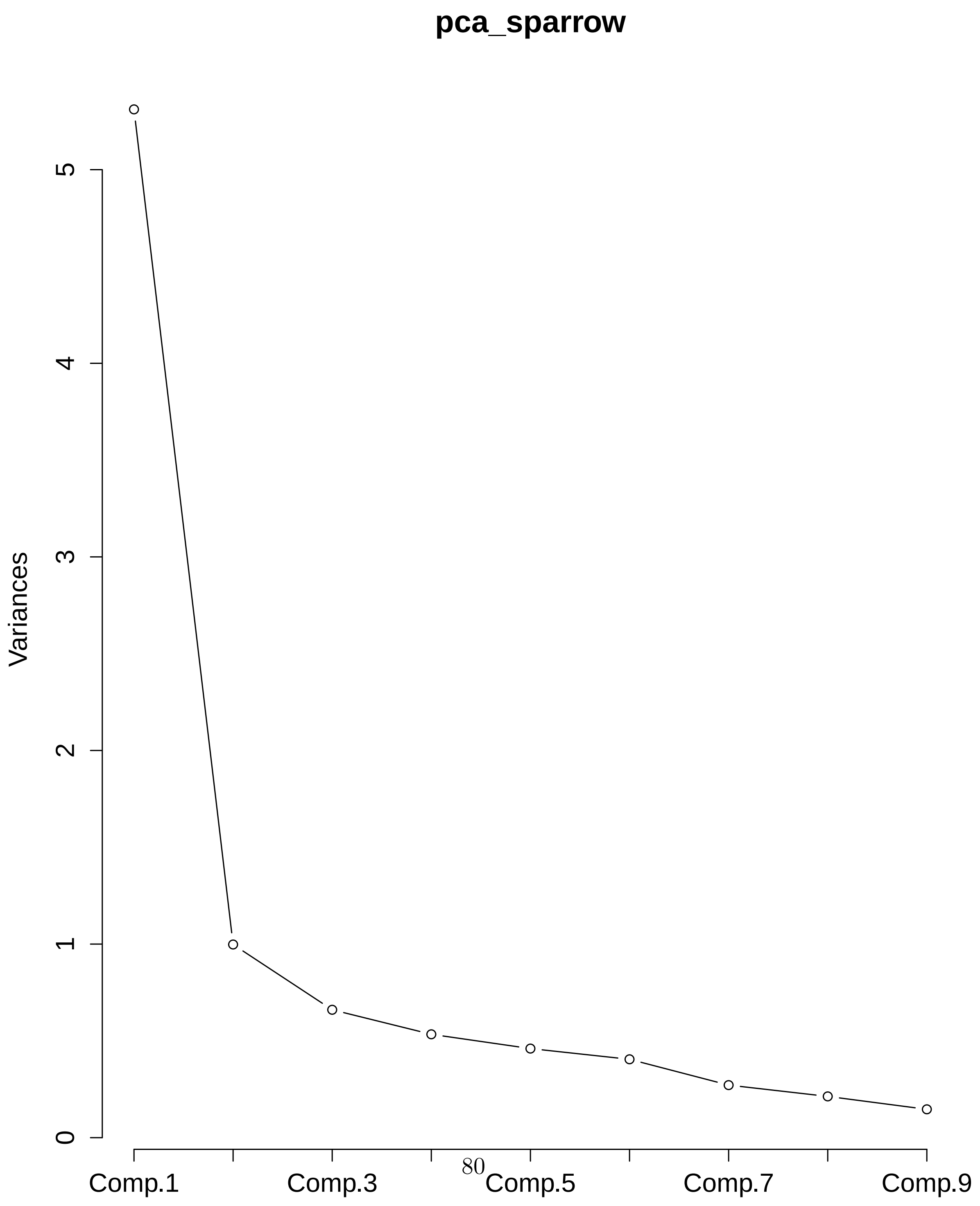
As you can see, when you call the summary function on the object (in this case pca_sparrow) that results from the princomp function, you get a summary of which of the 9 variables contributes the most to the variation in the data set. In this case, the first 5 variables are needed to account for 80-90% of the total variation in the data set – which is still a significant reduction down from 9 variables.

```
Importance of components:
                          Comp.1    Comp.2     Comp.3     Comp.4     Comp.5
Standard deviation     2.3046882 0.9988978 0.81280426 0.73068317 0.67837980
Proportion of Variance 0.5901764 0.1108663 0.07340564 0.05932199 0.05113324
Cumulative Proportion  0.5901764 0.7010427 0.77444838 0.83377037 0.88490361
                           Comp.6     Comp.7     Comp.8     Comp.9
Standard deviation     0.63624854 0.52104550 0.46168667 0.38263868
Proportion of Variance 0.04497913 0.03016538 0.02368384 0.01626804
Cumulative Proportion  0.92988274 0.96004812 0.98373196 1.00000000
```

This function produces an object that contains many useful columns including: *sdev - the standard deviation of the components*, *loadings-matrix with the eigenvectors of the columns*. The fitted object that is produced from this function can then be plotted to produce something called a ”SCREE plot” which gives you a visual interpretation of how many variables you need to mostly capture the variation in your data set. How do you tell how many principle components you need? There are a few rules. Probably the best rule is to take as many components as total between 80-90% of the total variation. You can also plot the results of the fitted princomps and see where the ”elbow” in the line is. It is at this point where you can stop including components. newline **>plot(pca_sparrow, type=”lines”)**

Just remember that PCA does have assumptions baked into it that will influence the results. PCA is especially vulnerable to violations of a balanced design. (see Gil McVean, 2009 “A genealogical interpretation of Principle Component Analysis”).

Figure 11.1: This graph shows an "elbow" which gives a graphic reminder of how many variables you need to include to account for most of the variation in your data



Chapter 12

Programming in R

12.1 Goals:

We will be introduced to the major 3 themes in R programming as well as a few details of R phenomenon (like ellipsis -which are so particular that they deserved to be highlighted all by themselves). Remember that there are additional details and examples in the accompanying .rmd file.

1. Overview of R programming language including an advertisement for and reminder about the SWIRL() package. R is a vector or functional programming language.
2. Loops - Allow us to break a problem down into repeatable actions.
3. Functions - how to create your own functions instead of only using ones that are defined by others (either built-in to basic R or methods of packages). Also, we will introduce Anonymous functions - tiny functions that are not given a name and are usually present in the function header.
4. Ellipsis - a strange, mostly-R phenomenon: three dots that, when used as the argument in a defined function, allow for a flexible number of arguments into the argument. The one rule associated with an ellipsis is that all other arguments after the ellipses have default values. This is a strict rule in R programming: all arguments after an ellipses must have default values.

12.2 Introduction to Programming in R

Programming is about breaking down a problem into smaller, more easily resolved sub-problems. Despite the overwhelming goal of programming simplifying procedures, when you are learning to program, the process can sometimes feel like this: `https://imgur.com/gallery/RadSf`.

We won't cover all of programming in this chapter but we will cover how to break problems down and the basic tools of doing so including familiarity with the structure of creating your own functions and effectively using loops. The intention is that by working through lots of examples, you will - hopefully - pick up some of the major rules of R programming. Programming in **R** follows many conventions of other languages but, of course, also has some unique features. We will not exhaustively work through **R Programming** but we will demonstrate some of the most likely scenarios that you might encounter when using **R** for data analysis. As I have mentioned previously, a new course focused on programming in R is now offered by the statistics department: STT 276 - Statistical Computing in R. if you are interested in a deeper understanding of programming in R, you should investigate taking this course!

In this brief overview chapter, we will discuss issues that are generally a part of the logic of programming languages (loops, functions, anonymous functions) and a few things that are peculiar to R since it is approximately described as a 'vector language' or a 'functional language' (another example of functional languages is Haskell, if you happen to be familiar with other languages). Besides working through examples in this chapter and in the accompanying .rmd file, you will want to install the SWIRL() package and work through the provided examples to more thoroughly cement the basics of programming in R in your mind.

1. R is mostly a functional (also sometimes referred to as a vector language) programming language. This means that higher order functions - and list comprehensions - abound! List comprehensions are, basically, an efficient way of re-writing loops so that they take place on only one line. What are higher order functions? Higher order functions are functions *that take another function as an argument and then do something with it.* Sounds complicated, right? If you have familiarity with other programming languages, you will have seen higher order functions like **filter, map, reduce** and other less popular higher order functions such as **find, position and negate**. Don't worry, though because we've *already* seen the *apply family which is a lot like the higher order map function in other languages.

2. The SWIRL() package is a fantastic entry point into R functionality. I mentioned the swirl() package in an earlier chapter - it is beloved by many R super-users because it will allow you to work through various programming concepts in an interactive environment, even as a beginner. You will need to install the swirl package and load it. For the most part, the package itself will guide you through your options and is reasonably user-friendly. It is also an efficient, although a tiny bit dry (okay: it is really, really dry), way to quickly review concepts in the R language. It should not come as a surprise that programming languages often have to solve similar problems and do so in similar ways.

12.2.1 Repeated problem solving: for loops

One of the most important features of a program is that it can be used to manipulate lots of data in *repetitive ways* so that you don't have to do the same thing every time! Two major approaches are writing a **for** loop or using the **apply** family of functions. Although we technically have both of these options available to us in R, and will briefly use both, 'good programming' in R usually involves the application of the appropriate member of the 'apply' family of functions rather than 'for loops'.

For loops

The following example, repeats the same command 10 times. The variable **i** is a counter that starts at 1 (**NOTE: R starts counting at 1 - not 0, like most programming languages**)and increases by 1 increment each time the commands between the squiggly brackets are executed. Notice that in this piece of code, you can infer the syntax of programming in R, including the use of the squiggly brackets to indicate the beginning and ending of distinct 'chunks' of code to be evaluated. You can see how these evaluate in the R cells in the accompanying .rmd file:

```
for(i in 1:10){
       print("Remember that there is a syntax to defining your loops.")
}
```

You could also use other built-in functions to get the top end of the loop, for instance *length(x)* where you need to set an argument, *x*, into the length.

```
for(i in 1:length(x)){
        print(x[i])
}
```

apply

All of the so-called *apply functions will split the data into smaller pieces, apply a function to each of the smaller pieces and then combine the results. They allow you to "automate" applying a function to multiple elements of a list, dataframe, vector or matrix. The **lapply()** function takes a list as input, applies a function to each element of the list and returns a list of the same length as the original one. The **sapply()** function takes the results of the **lapply()** function and **simplies** it. In general, if the result is a list where every element is of length one, then **sapply()** returns a vector. If the result is a list where every element is a vector of the same length (¿ 1), **sapply()** returns a matrix. If **sapply()** can't figure things out, then it just returns a list, no different from what **lapply()** would give you.

The apply family is usually much faster than a for loop in R (due to R being mostly a vector language). You can repeat a function on a column (or multiple columns) of data frame very efficiently using members of this family. In the example below, MARGIN =2 refers to columns; you could also apply the same function to each row by using MARGIN=1, instead. In this case, the function being applied to each element of the column is the mean(). That is indicated by FUN=mean. You could substitute any appropriate function in the FUN=argument including any that you have defined yourself (which we'll learn to do in the next section).

```
result <- apply(mydata, MARGIN = 2, FUN = mean, na.rm = TRUE)
```

12.2.2 Create your own functions

NOTE: The following examples are from the SWIRL() package. Please download it and work through the section on functions since this section is a merely asuperficial summary of the major ideas in SWIRL() and is therefore not as in-depth as the actual package!.

General Functions

Functions take arguments, either user inputted or a default value set by the function creator, manipulate them and return a result. In order to write a function in **R**, follow a generic pattern. For example:

```
function_name <- function(arg1, arg2){
Manipulate arguments in some way
Return a value
}
```

You can write your own function and save it in the same pathway as **R** then you will need to run the function **submit()** in the console of **R**. You don‘t need to explicitly set a value to return;**R** will simply return the last expression that was evaluated.

In a slightly more complex scenario, you are also able to pass a function as an argument to another function just like you can pass data to functions. Here is an example taken from SWIRL:

```
 Let's say you define the following functions:
 add_two_numbers <- function(num1, num2){
    num1 + num2
 }

 multiply_two_numbers <- function(num1, num2){
num1 * num2
 }

 some_function <- function(func){
    func(2, 4)
 }
```

As you can see we use the argument name "func" like a function inside of **some_function()**. By passing functions as arguments **some_function(add_two_numbers)** will evaluate to 6, while **some_function(multiply_two_numbers)** will evaluate to 8.

Below, is an exercise in which you will need to pass a function as an argument of another function: Exercise

Finish the function definition below so that if a function is passed into the **func** argument and some data (like a vector) is passed into the **dat** argument, the **evaluate()** function will return the result of **dat** being passed as an argument to **func**.

Hints: This exercise is a little tricky so I'll provide a few example of how **evaluate()** should act:

1. **evaluate(sum, c(2, 4, 6))** should evaluate to 12

2. **evaluate(median, c(7, 40, 9))** should evaluate to 9
3. **evaluate(floor, 11.1)** should evaluate to 11

Answer:

```
evaluate <- function(func, dat){
  func(dat)
}
```

Anonymous Functions

Anonymous functions are useful functions that are not named (so they are not assigned to a name with the assignment operator). For example,

```
> evaluate(function(x){x+1},6)
```

The function(x)x+1 is a so-called anonymous function since it is small - in this case confined to one line - and isn't assigned a name. When this code is evaluated it should add 1 to the number, in this case 6, that is provided to it.

Ellipses

Ellipses are particularly ingenious because they allow you to write a function that has a flexible number of arguments. This means that you will need to "unpack" arguments from an ellipses when you use the ellipses as an argument in a function. Below I have an example function that is supposed to add two explicitly named arguments called **alpha** and **beta**.
add_alpha_and_beta <- function(. . .){
First we must capture the ellipsis inside of a list and then assign the list to a variable. Let's name this variable 'argv'. It is an an accident that we are calling this 'argv' since that is often the conventional name for an argument that allows a flexible number of arguments to be passed to a function. In the Python language, this would look like: $*argv$. **argv <- list(. . .)**

We're now going to assume that there are two named arguments within args with the names 'alpha' and 'beta.' We can extract named arguments from the args list by using the name of the argument and double brackets. The 'argv' variable is just a regular list after all!
alpha <- argv[["alpha"]]
beta <- argv[["beta"]]
Then we return the sum of alpha and beta.
alpha + beta
}

How to create your own brand spanking new function?

The syntax for creating new binary operators in R is unlike anything else in R, but it allows you to define a new syntax for your function. I would only recommend making your own binary operator if you plan on using it often! User-defined binary operators have the following syntax:

%[whatever]%

where [whatever] represents any valid variable name.

Let's say I wanted to define a binary operator that multiplied two numbers and then added one to the product. An implementation of that operator is below:

"%mult_add_one%" <- function(left, right){ . Immediately, you should notice the quotation marks!

left * right + 1

}

I could then use this binary operator like **'4 %mult_add_one% 5'** which would evaluate to 21.

<u>Exercise</u>

Write your own binary operator below from absolute scratch! Your binary operator must be called %p% so that the expression:

"Good" %p% "job!"

will evaluate to: "Good job!"

<u>Answer:</u> **"%p%" <- function(){**

}

Chapter 13

Appendices

13.1 Working with RStudio Markdown language

Initial_Data_Exploration

Danielle A Presgraves

September 29, 2016

Note: in the latest version of RStudio, students had to use good ol' >install.packages("rmarkdown") function in the console in order to use the .Rmd. This was in contrast to earlier versions of RStudio, so be aware if this!

This is a sample runthrough of common commands for data set exploration in RMarkdown. In today's class, we are going to run through some basic ''data exploration" which usually means how to test assumptions of major tests by graphing. We will also work through some major tests.

First, we are going to demonstrate the basic commands of RMarkdown language and load some data. Note: if you didn't want this appear in your final rendered document, you could specify include=FALSE next to the 'r' like so {r include=FALSE}. You can control the output by using "echo"" which will evaluate the chunk of r code and return the results but not include the r code itself. You can find excellent ''cheat sheets''for RStudio and the Markdown Language that is used outside of the''chunks" here: There are MANY excellent cheat sheets here that outline functions and capabilities:

https://www.rstudio.com/resources/cheatsheets/

Here is a less pretty overview of R language functions:

https://cran.r-project.org/doc/contrib/Short-refcard.pdf

We'll begin by demonstrating the basic tools of RStudio that are going to make our lives easier (and allow us to produce beautiful documents which will impress all of our future Professors). First up: inserting a chunk and uploading data. Remember an important 'cheat' that I showed you in lecture: There is a command called file.choose() that you can type into the CONSOLE (NOT the chunk) and it will 'pop up' the files on your computer and allow you to choose the one that you want to use. If you use good programming habits and assign the results of your file.choose() into a variable, let's say "a" (so in the console this would look like: >a<-file.choose()), you can call "a" in the console after evaluating this command and it will bring up the path which you can then cut and paste the correct path into your chunk when you upload the file:

Note also that windows users may (or will) have a different path format: windows:

"C:\Users\Username\Desktop\cardiac.csv"

and some mac users:

"//Users/name/Downloads/cardiac.csv

```r
cardiac<-read.csv("~/Desktop/Bio214/BIOL300 data sets/canada obesity.csv")
BP<-read.csv("~/Desktop/Bio214/BIOL300 data sets/BP.csv")
```

let's see the names of the columns of the cardiac file:

```r
names(cardiac)
```

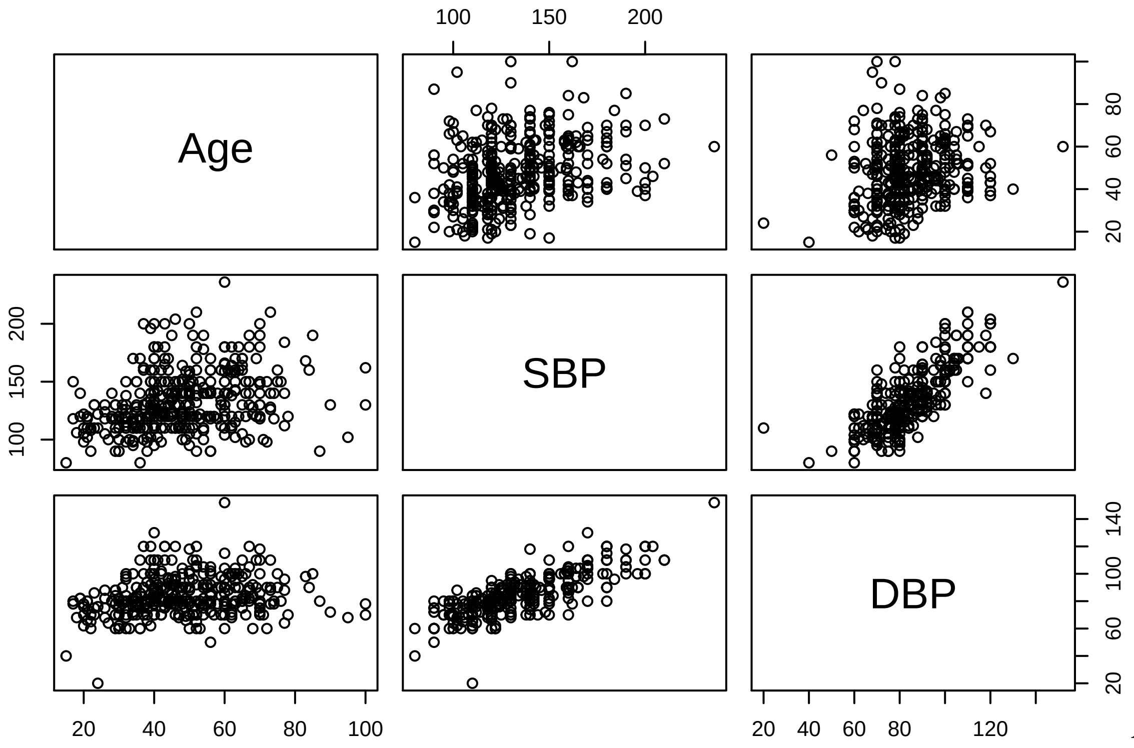
```
## [1] "Province"                          "Physical.inactivity....adults."
## [3] "Physical.inactivity....teens."  "Obesity"
```

We always want to ensure that we 'attach' and 'detach' the variables in our data sets so that we don't have to call the entire name.

```r
attach(cardiac)
attach(BP)
```

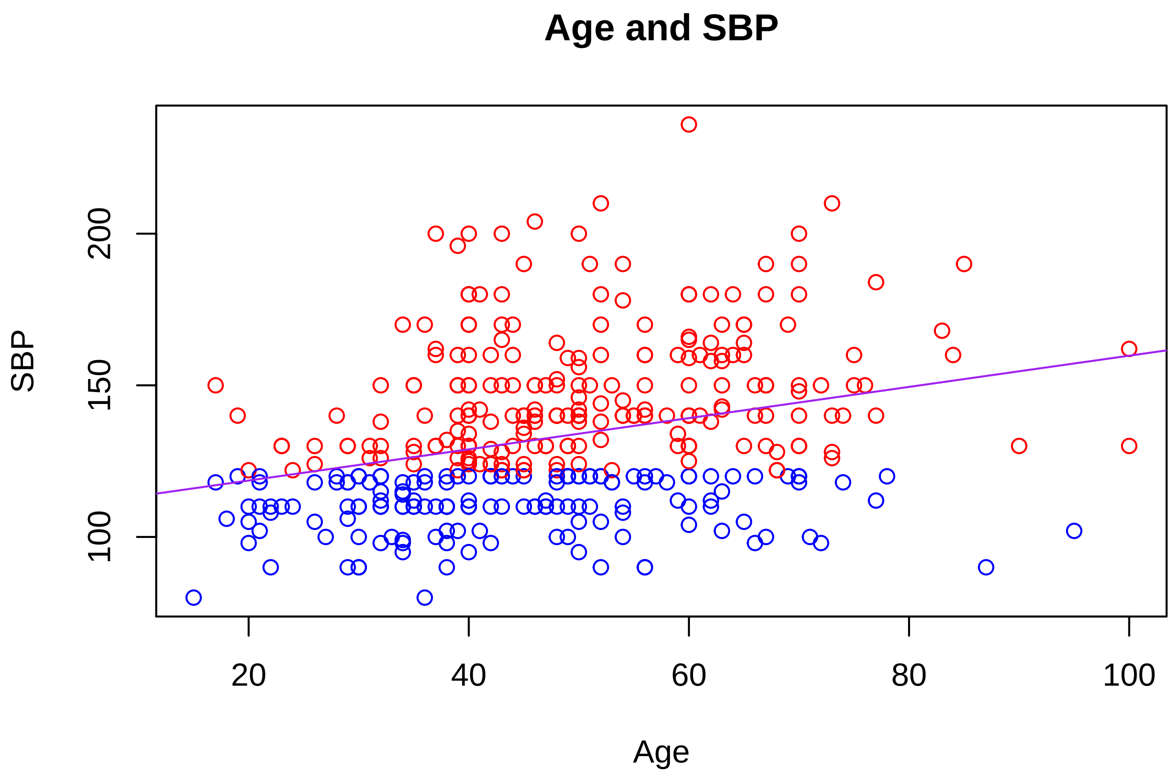
So now we have three files. Let's see if we can explore the data in each one. First, we will start with a "scatterplot matrix":

```
plot(~Age+SBP+DBP)
```



As you can see above, this just gives us some initial relationships between the columns we specified (Age, SBP and DBP). Okay, so the matrix didn't suggest any particularly strong reason to choose Age and SBP but we know that having a SBP above 120 mmHG is very predictive of your chance of having a heart attack. So let's start by plotting that out.

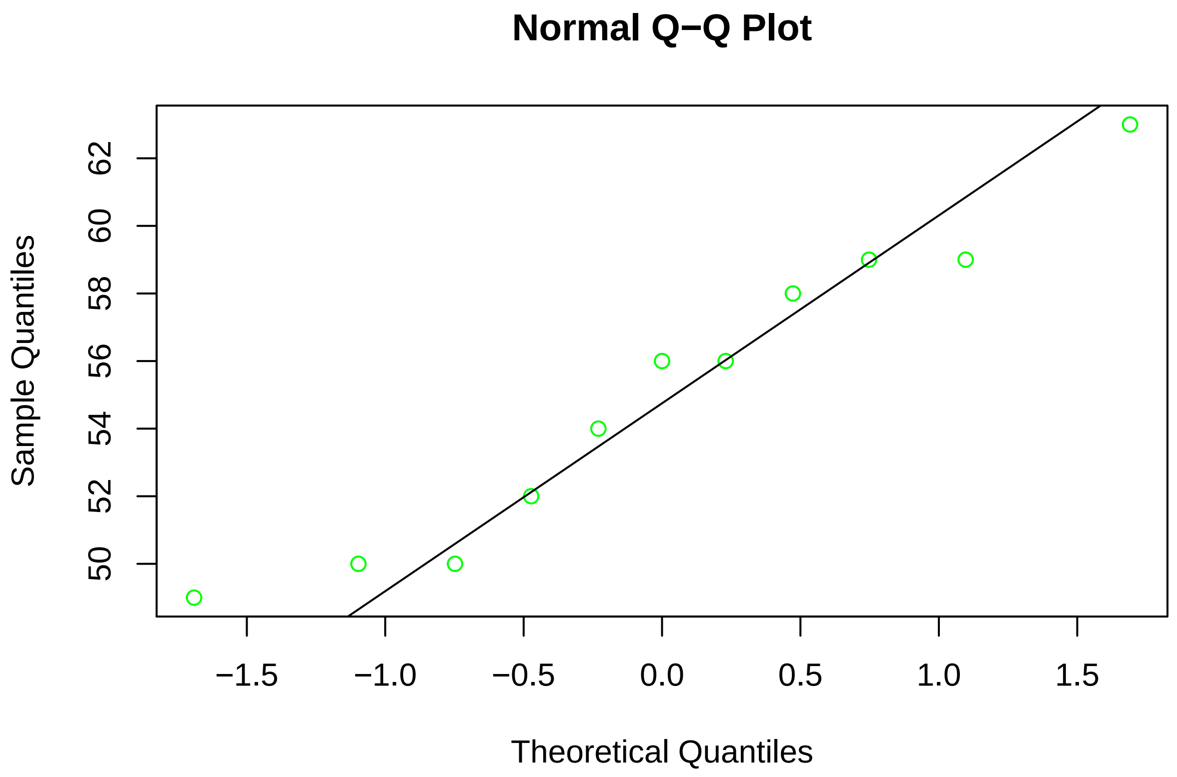
```
plot(Age, SBP,main = "Age and SBP",col=ifelse((SBP > 120), "Red", "Blue"),xlab="Age",ylab="SBP")
abline(lm(SBP~Age),col="Purple")
```



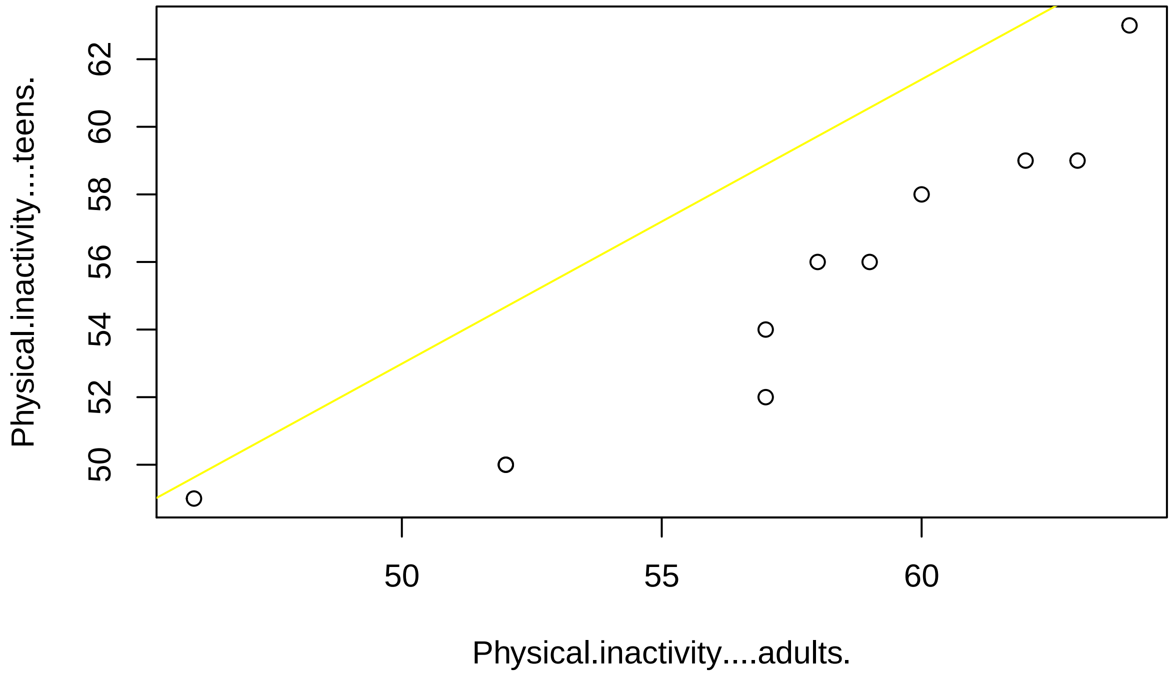
We can place a potential line suggesting by using 'lm' and we can indicate the individual measurements that are above our arbitrary cut-off of 120 mmHG.

So now let's tackle basic assumptions of most of the tests that we will use, like t-tests, ANOVA, correlation/regression: Is your data normally distributed? There are a couple of ways to answer this question: A. qqnorm plots

```
qqnorm(Physical.inactivity....teens.,col="Green")
qqline(Physical.inactivity....teens.)
```

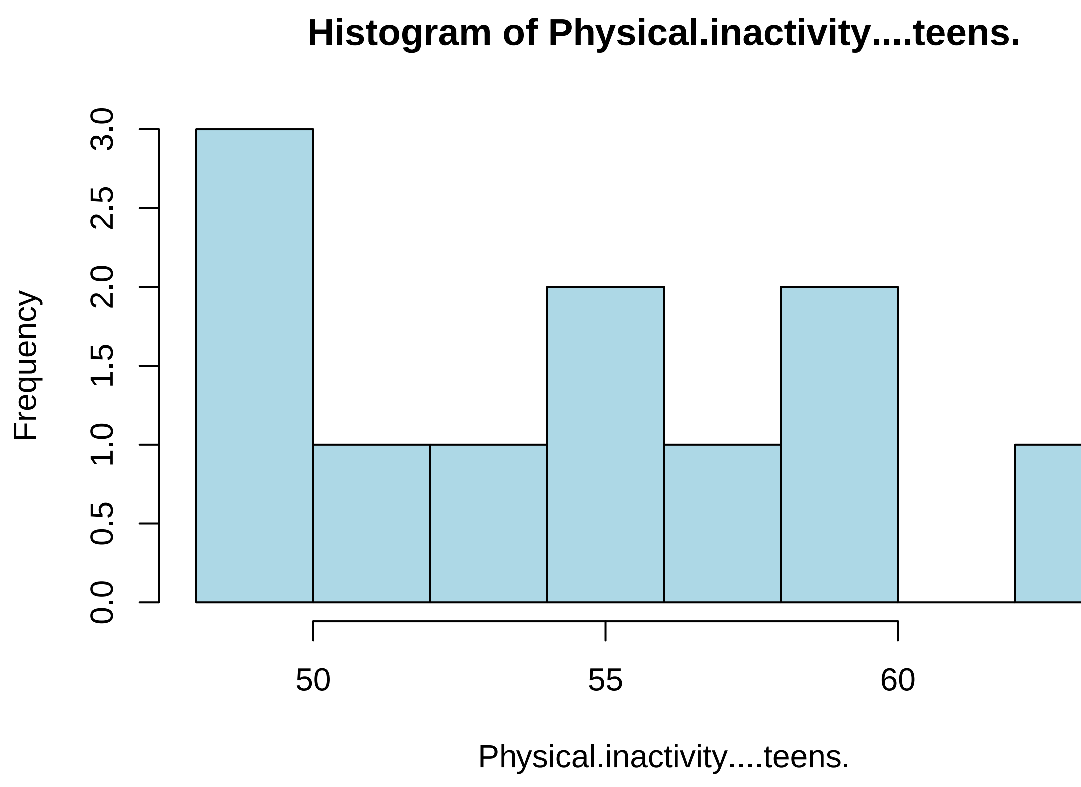


```
qqplot(Physical.inactivity....adults.,Physical.inactivity....teens.)
abline(lm(Physical.inactivity....adults.~Physical.inactivity....teens.),col="Yellow")
```

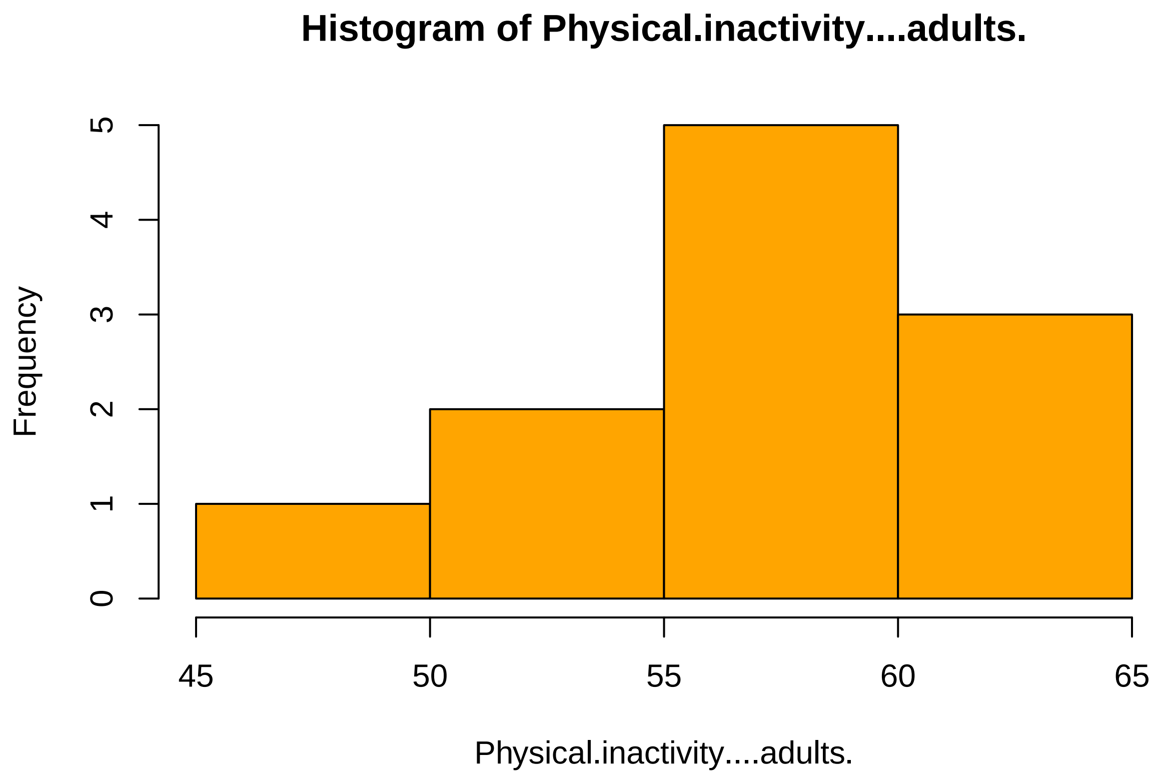


B. Histogram

```
hist(Physical.inactivity....teens.,col="Light Blue")
```



```r
hist(Physical.inactivity....adults.,col="Orange")
```



Neither of these looks particularly normally distributed. Let's try a formal test to see:

```r
teens_Normal <-shapiro.test(Physical.inactivity....teens.)
teens_Normal
```

```
##
##  Shapiro-Wilk normality test
##
## data:  Physical.inactivity....teens.
## W = 0.94635, p-value = 0.5977
```

```
adults_Normal<-shapiro.test(Physical.inactivity....adults.)
adults_Normal
```

```
##
##  Shapiro-Wilk normality test
##
## data:  Physical.inactivity....adults.
## W = 0.93179, p-value = 0.4293
```

The p-values for these tests are > 0.05 so we can't reject the null hypothesis (which is good: it means we can't reject that this data is normally distributed)

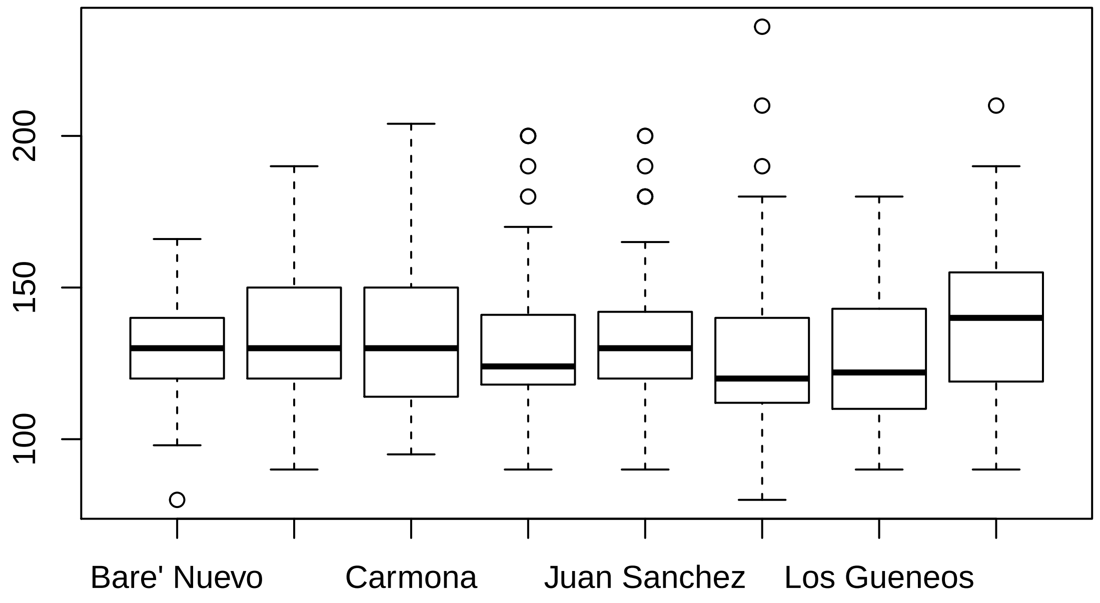
So now let's see if our BP data has the same variance for each Village by using a Bartlett test for equal variances:

```
Village_Variance<-bartlett.test(SBP,Village)
Village_Variance
```

```
##
##  Bartlett test of homogeneity of variances
##
## data:  SBP and Village
## Bartlett's K-squared = 11.389, df = 7, p-value = 0.1225
```

Another way of visually ascertaining if there are outliers or other weird variance stuff going on is with a boxplot. Let's try to create one now:

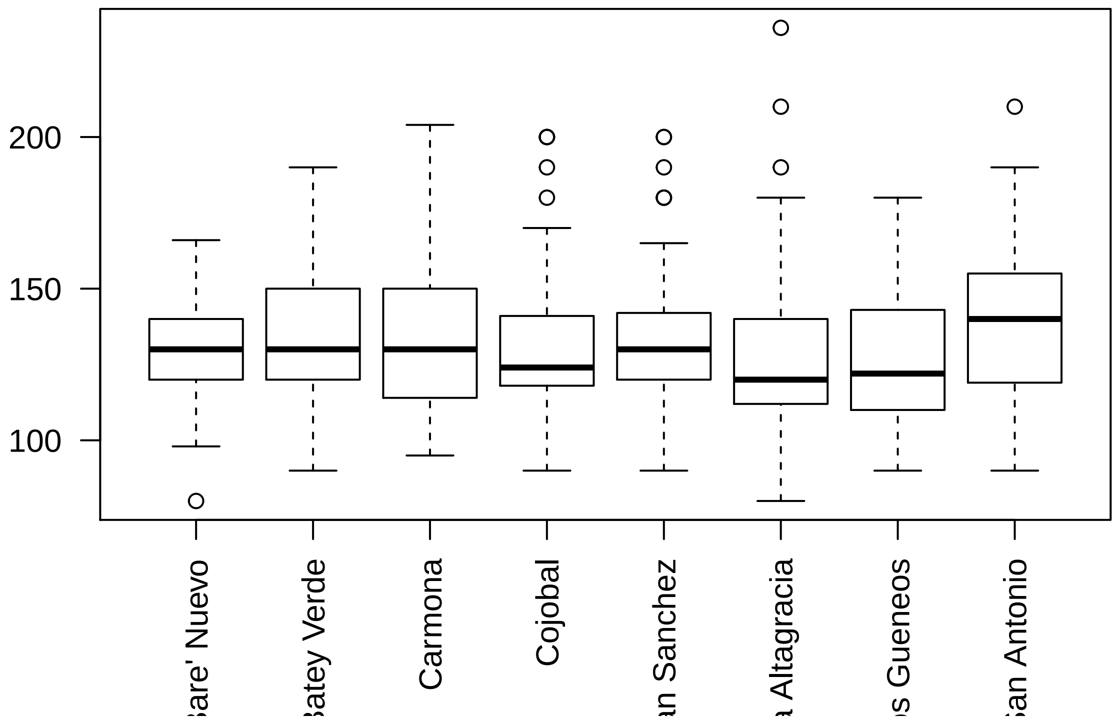
```
boxplot(SBP~Village)
```



This isn't a very pretty graph, is it? Remember that you can ALWAYS USE GOOGLE searches to find ways to make your graph look prettier by adding color, legends and labels.

As an easy way to improve the look of this boxplot, you can use the par function.

```
boxplot(SBP~Village,par(las=2))
```



Let's move on to ensuring that you understand the basics of the other kinds of graph that you may use to explore your own data.

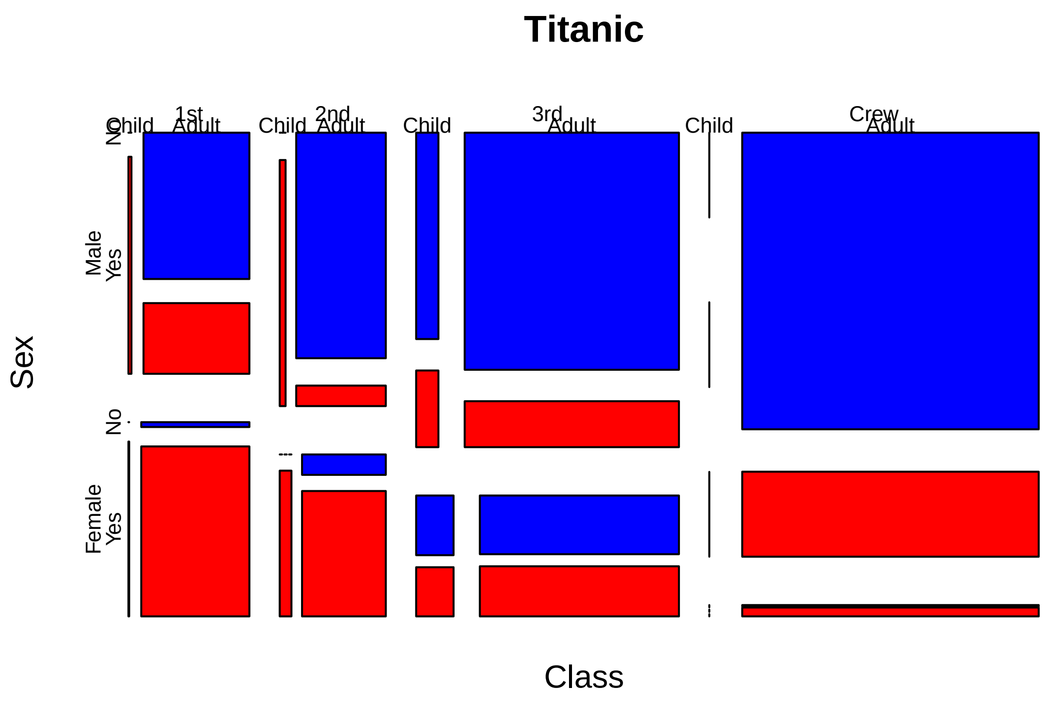
Mosaic plots are used for categorical data which are in vector or table format.

We will use the included data set "Titanic" to see how to use mosaicplot. You may have to load this data set by calling Titanic.

```
Titanic
```

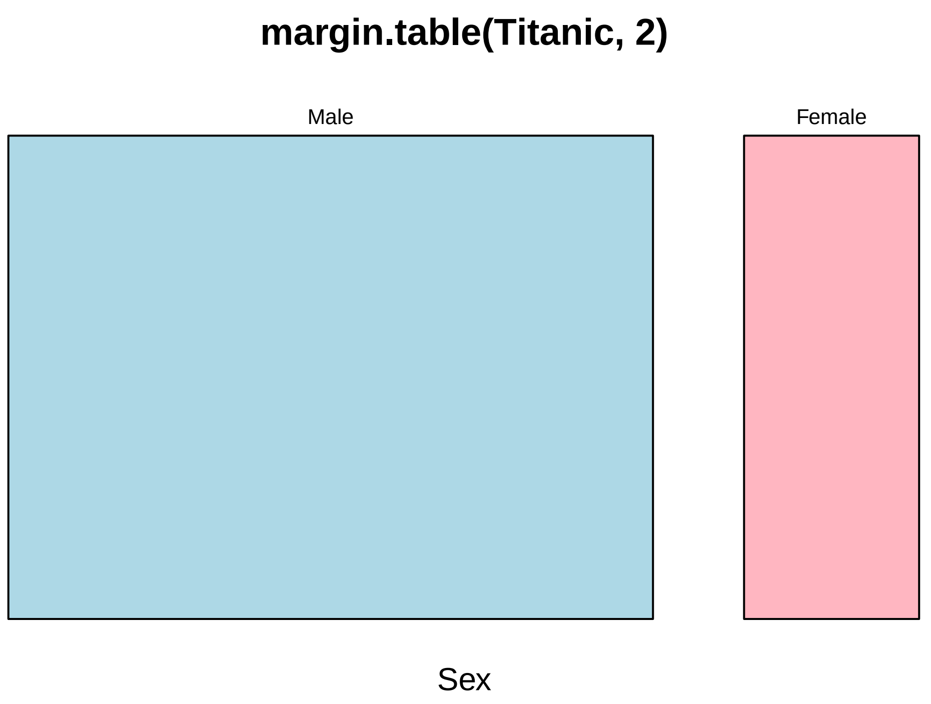
```
## , , Age = Child, Survived = No
##
##       Sex
## Class  Male Female
##   1st     0      0
##   2nd     0      0
##   3rd    35     17
##   Crew    0      0
##
## , , Age = Adult, Survived = No
##
##       Sex
## Class  Male Female
##   1st   118      4
##   2nd   154     13
##   3rd   387     89
##   Crew  670      3
##
## , , Age = Child, Survived = Yes
##
##       Sex
## Class  Male Female
##   1st     5      1
##   2nd    11     13
##   3rd    13     14
##   Crew    0      0
##
## , , Age = Adult, Survived = Yes
##
##       Sex
## Class  Male Female
##   1st    57    140
##   2nd    14     80
##   3rd    75     76
##   Crew  192     20
```

```
mosaicplot(Titanic,col=c("Blue","Red"))
```

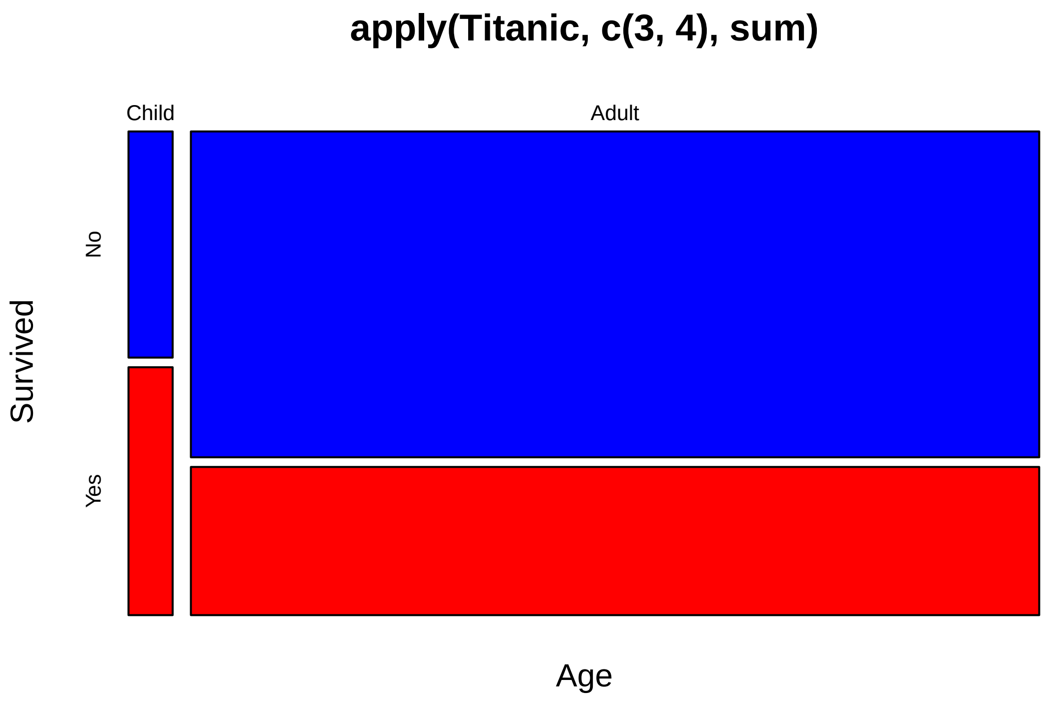


This is very ugly. There are four dimensions included in this pre-loaded data set: (1 class, 2 sex, 3 age, 4 survival) so it can be though of as a 2X2X2X2 contingency table. Let's corral the data to ask the question *Did women and children disproportionately survive the sinking of the Titanic*? Here are -as always-multiple ways to begin to cluster the data:

```
mosaicplot(margin.table(Titanic,2),col=c("Light Blue","Light Pink"))
```



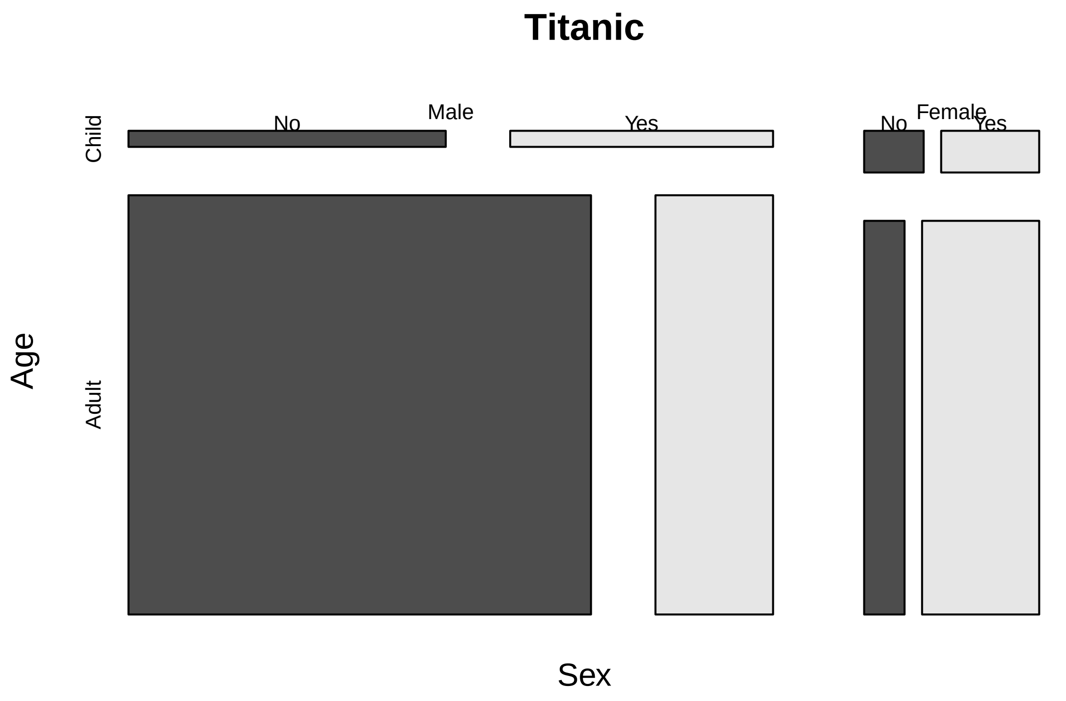
```r
mosaicplot(apply(Titanic, c(3, 4), sum),col=c("Blue","Red"))
```



```r
prop.table(apply(Titanic,c(3,4),sum))
```

```
##        Survived
## Age              No        Yes
##   Child 0.02362562 0.02589732
##   Adult 0.65333939 0.29713766
```

```r
mosaicplot(~ Sex + Age + Survived, data = Titanic, color = TRUE)
```



```r
prop.table(apply(Titanic,c(2,4),sum))
```

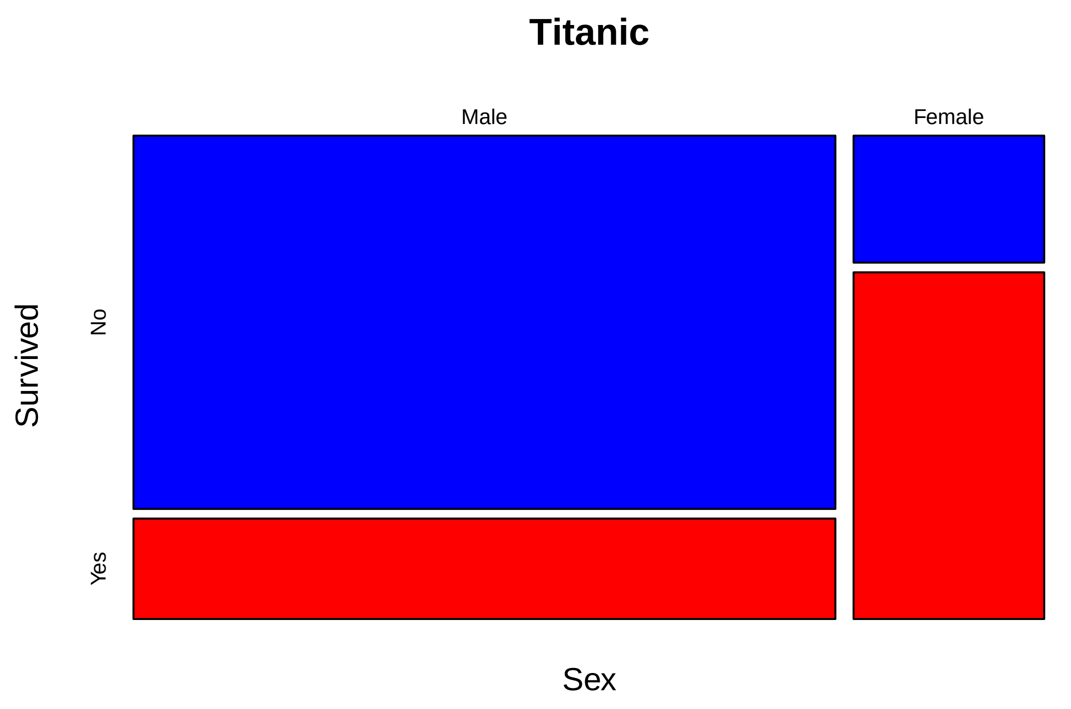
```
##         Survived
## Sex              No       Yes
##   Male   0.61971831 0.1667424
##   Female 0.05724671 0.1562926
```

```r
prop.test(apply(Titanic,c(3,4),sum))
```

```
##
##  2-sample test for equality of proportions with continuity
##  correction
##
## data:  apply(Titanic, c(3, 4), sum)
## X-squared = 20.005, df = 1, p-value = 7.725e-06
## alternative hypothesis: two.sided
## 95 percent confidence interval:
##  -0.3109899 -0.1096426
## sample estimates:
##    prop 1    prop 2
## 0.4770642 0.6873805
```

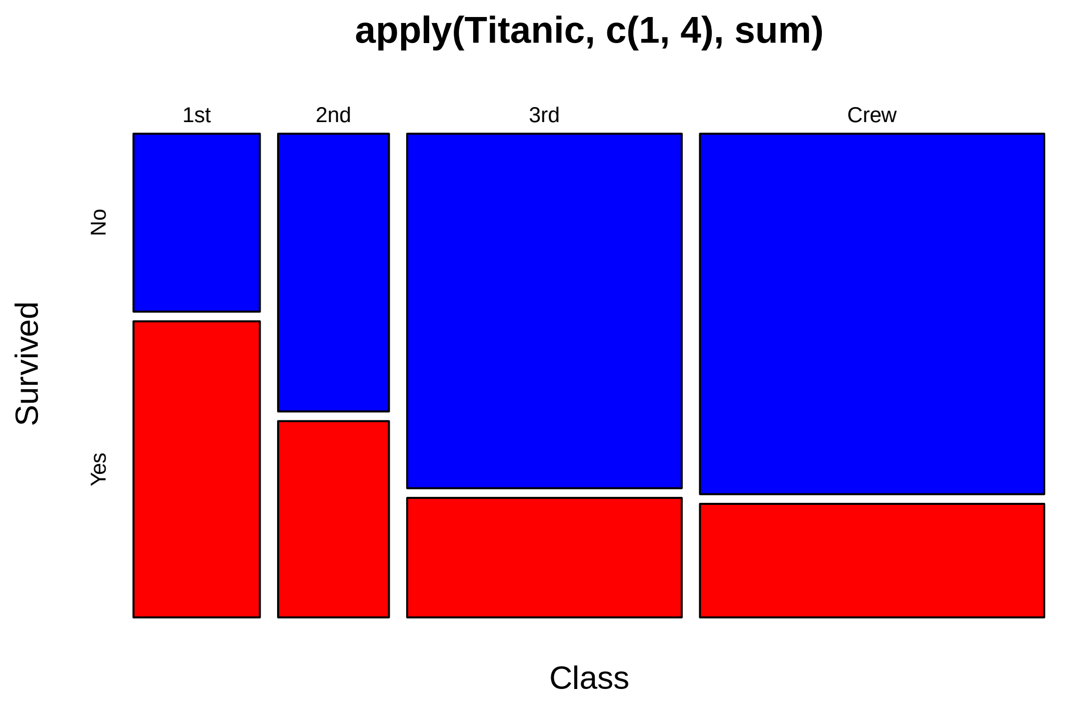
About 50% of the children aboard the HMS Titanic survived the sinking and approximately 3/4 of the women survived(regardless of whether or not they were children or adults) whereas only about 17/79 of the men survived.

```r
mosaicplot(~Sex+Survived, data=Titanic, col=c("Blue","Red"))
```



A more interesting pattern was seen earlier, though, in our general data exploration. Did your chance of survival depend on your class? Once again, we can corral our data visually and statistically:

```
mosaicplot(apply(Titanic, c(1, 4), sum),col=c("Blue","Red"))
```

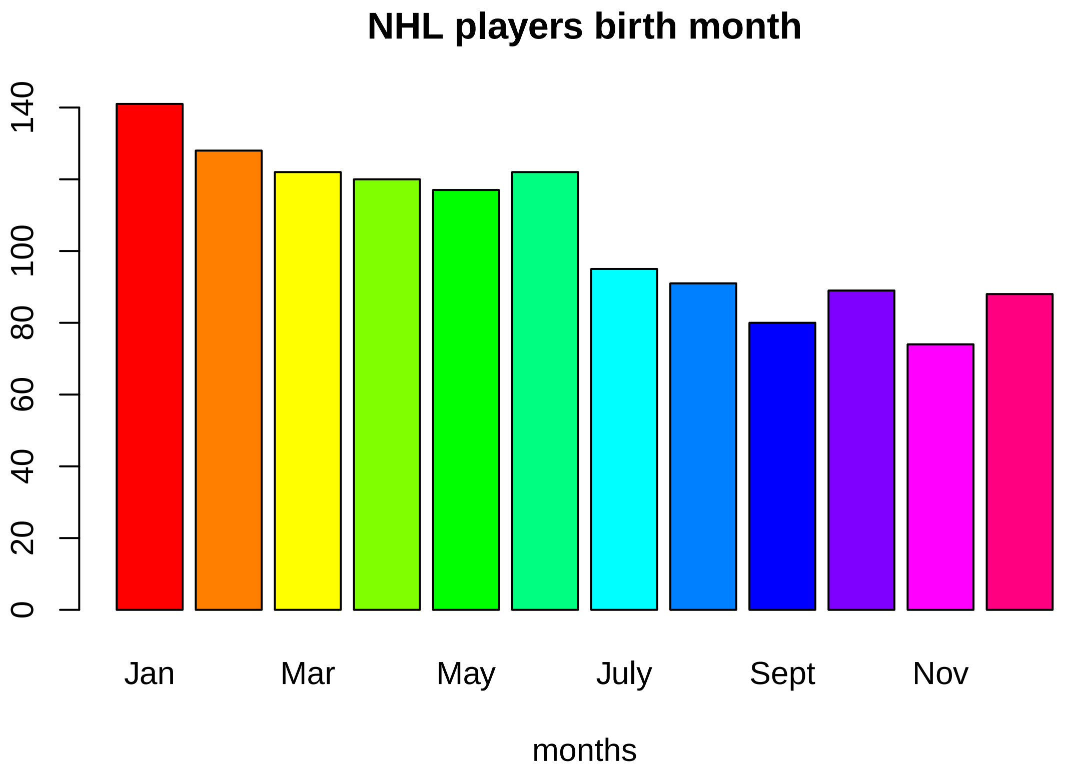


```
prop.table(apply(Titanic,c(1,4),sum))
```

```
##       Survived
## Class          No        Yes
##   1st  0.05542935 0.09223080
##   2nd  0.07587460 0.05361199
##   3rd  0.23989096 0.08087233
##   Crew 0.30577010 0.09631985
```

So now, lets see how we do a bar graph. We will use my favourite data set, adapted from "outliers" by Malcolm Gladwell: the birth month of NHL players.

```
NHL<-read.csv("~/Desktop/Bio214/BIOL300 data sets/NHL births.csv")
attach(NHL)
barplot(Number.of.players, main="NHL players birth month",xlab="months",names.arg=c("Jan","Feb","Mar","
```



Now that we have explored the data to ensure that most of the asssumptions of the major tests work, we shall proceed to use those tests.

First up, a straightforward *t test*. Always ensure that you have created an object to park your results in. Also remember to test for assumptions in your data such as a normal distribution and reasonably equal variances (or use welch's t-test by stating var.equal=FALSE in the options to the test). *Note that I somehow experienced a brain vacation and attempted to run a t-test comparing SBP and Age previously. I have replaced this with a more appropriate dataset. Sorry about that!*

```
test_T<-t.test(Physical.inactivity....adults.,Physical.inactivity....teens.)
test_T
```

```
##
##  Welch Two Sample t-test
##
## data:  Physical.inactivity....adults. and Physical.inactivity....teens.
## t = 1.0264, df = 19.348, p-value = 0.3174
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -2.262118  6.625754
## sample estimates:
## mean of x mean of y
## 57.27273 55.09091
```

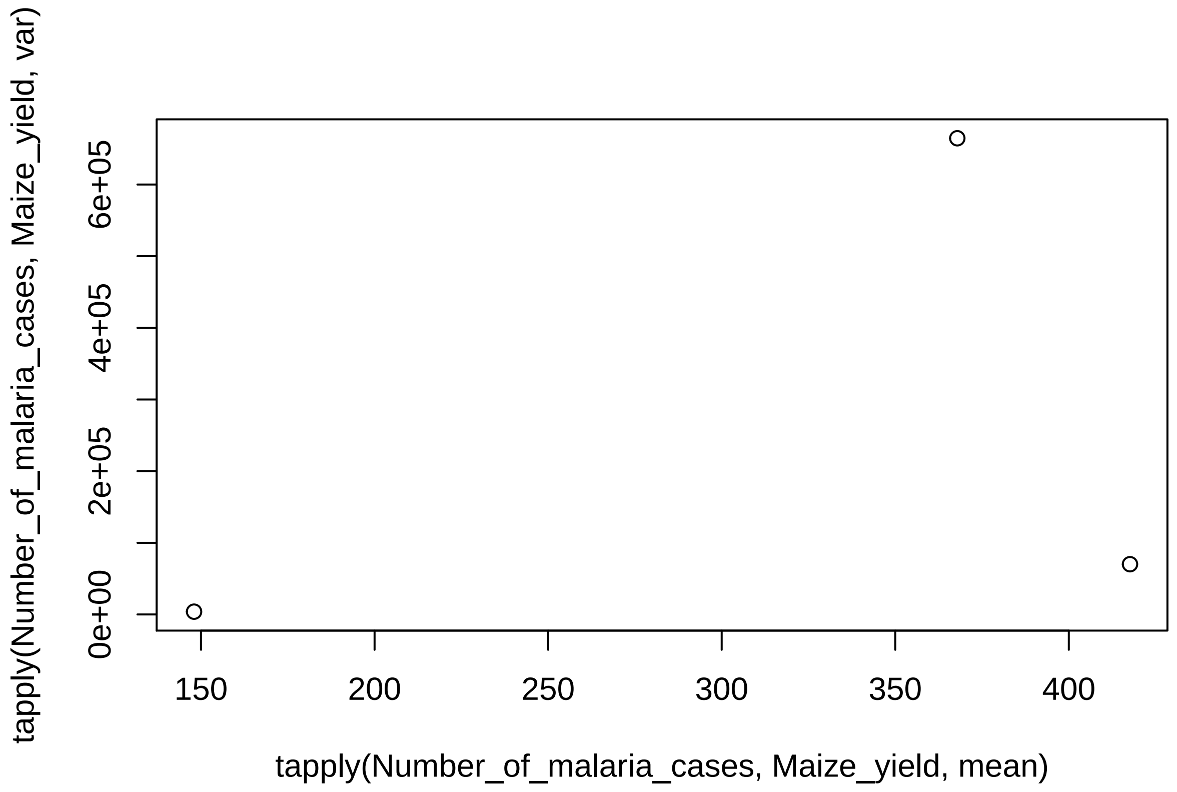
```
test_T_student<-t.test(Physical.inactivity....adults.,Physical.inactivity....teens.,var.equal = TRUE)
test_T_student
```

```
##
##  Two Sample t-test
##
## data:  Physical.inactivity....adults. and Physical.inactivity....teens.
## t = 1.0264, df = 20, p-value = 0.317
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -2.252514  6.616151
## sample estimates:
## mean of x mean of y
##  57.27273  55.09091
```

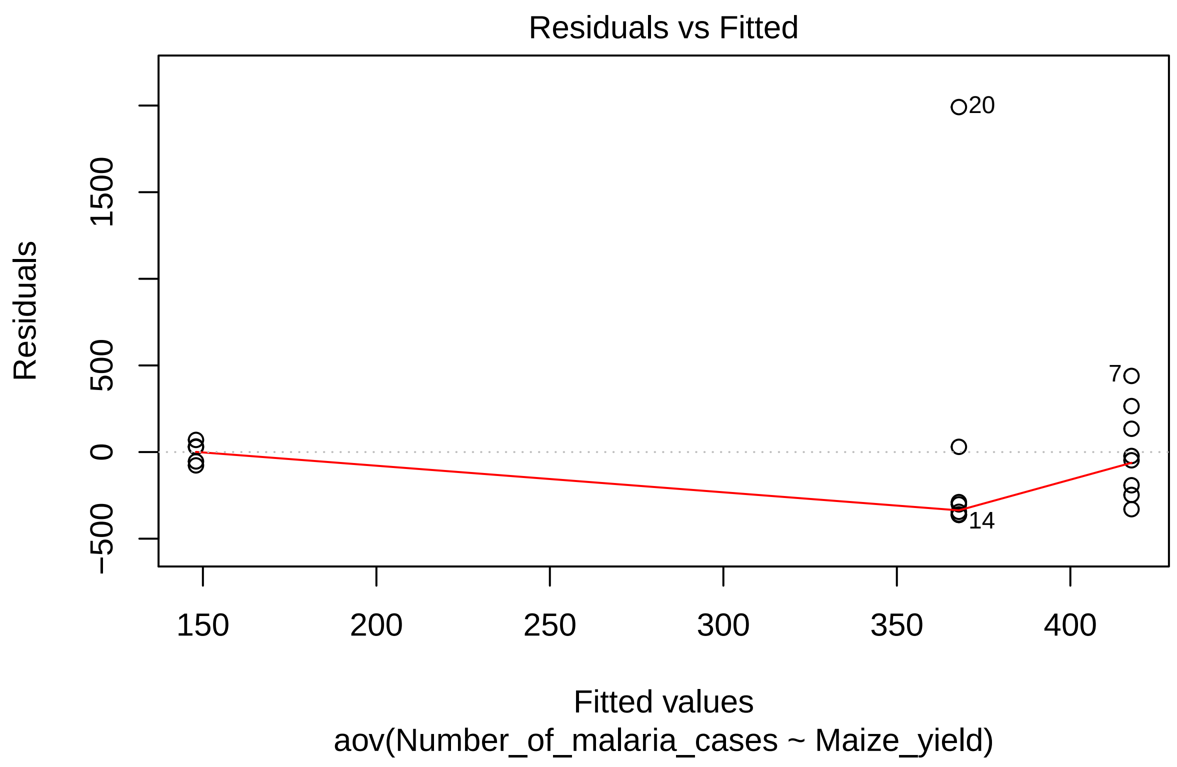
You can notice that there is not a large difference between the results of these two tests *except in the number of degrees of freedom.*

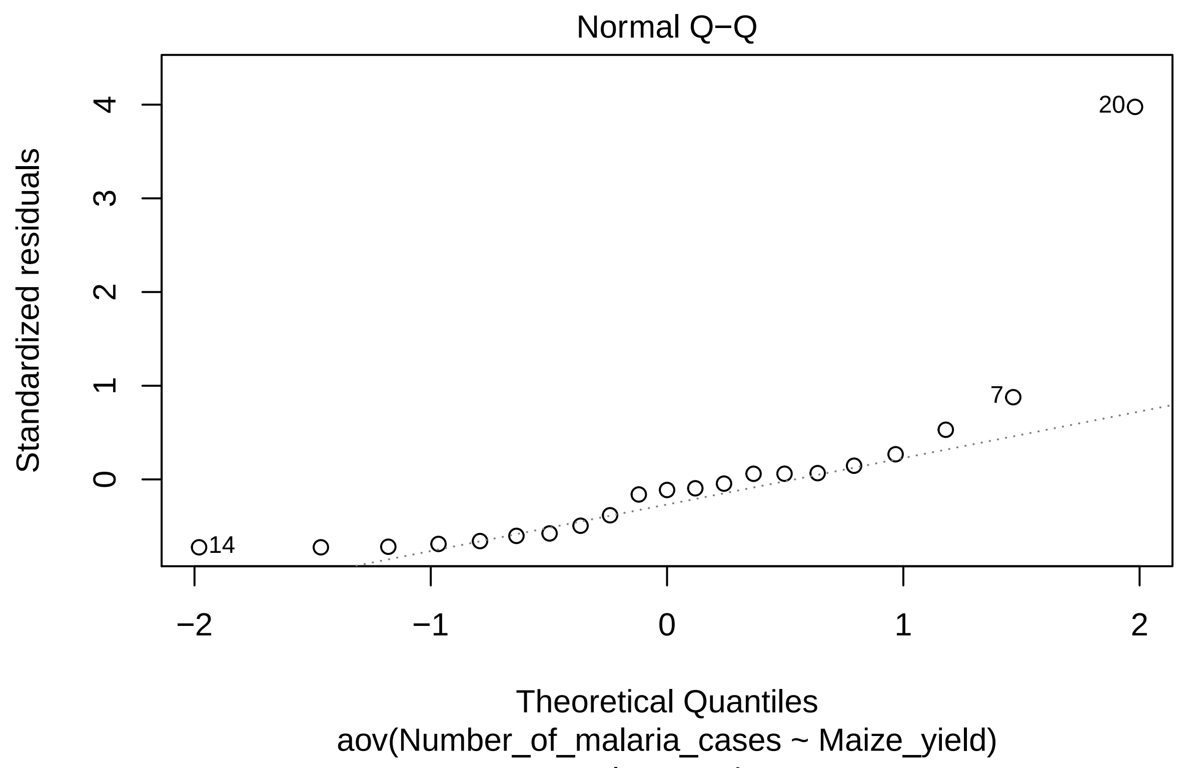
Another common test that many of you will want to work through is ANOVA. There is one more step in testing assumptions necessary for ANOVA: Checking the homogeneity of variance between the different groups. If your data are normally distributed, you may use the bartlett test for equal variances. If not, you can quickly visually confirm equal-ish variance by graphing the mean versus the variance for each of the groups. This is to ensure that your data points are scattered evenly and if they aren't that the variance between groups increases with increasing means of the groups. If your data doesn't fulfill the assumptions, you may need to investigate various transformations of the data to see if you can get a transformation that works or you may need to use the kruskal-wallis nonparametric version of ANOVA.

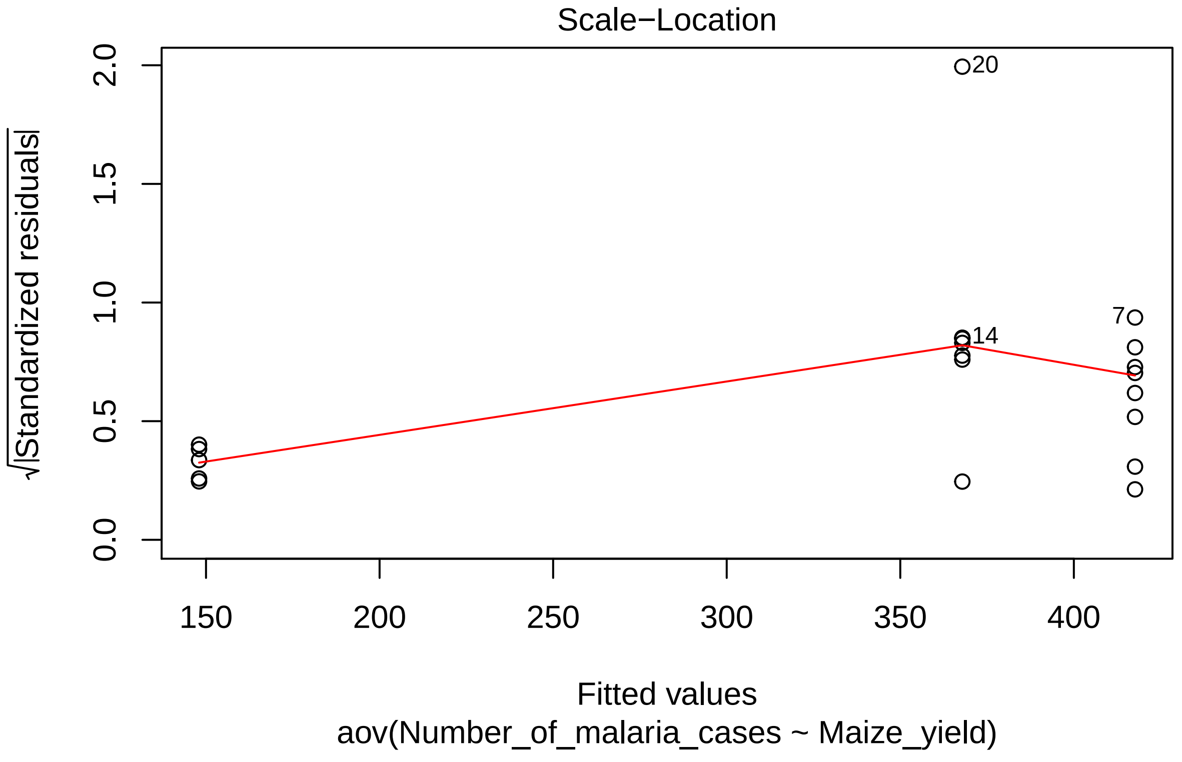
```
malariaMaize<-read.csv("~/Desktop/Bio214/BIOL300 data sets/malariaMaize.csv")
attach(malariaMaize)
plot(tapply(Number_of_malaria_cases,Maize_yield,mean),tapply(Number_of_malaria_cases,Maize_yield,var))
```

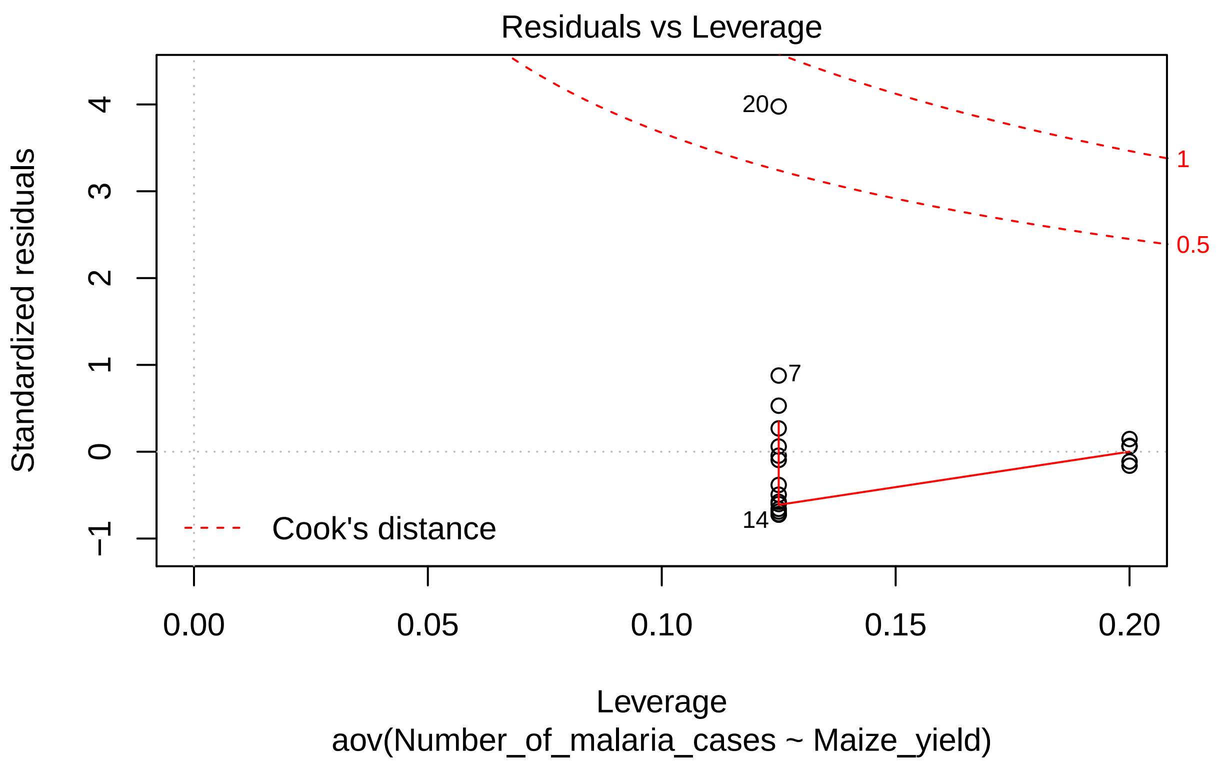


```
result.aov<-aov(Number_of_malaria_cases~Maize_yield)
plot(result.aov)
```




Normal Q−Q
Standardized residuals
Theoretical Quantiles
aov(Number_of_malaria_cases ~ Maize_yield)
20
7
14
−2
−1
0
1
2
0
1
2
3
4


Scale−Location
√|Standardized residuals|
Fitted values
aov(Number_of_malaria_cases ~ Maize_yield)
20
14
7
150
200
250
300
350
400
0.0
0.5
1.0
1.5
2.0



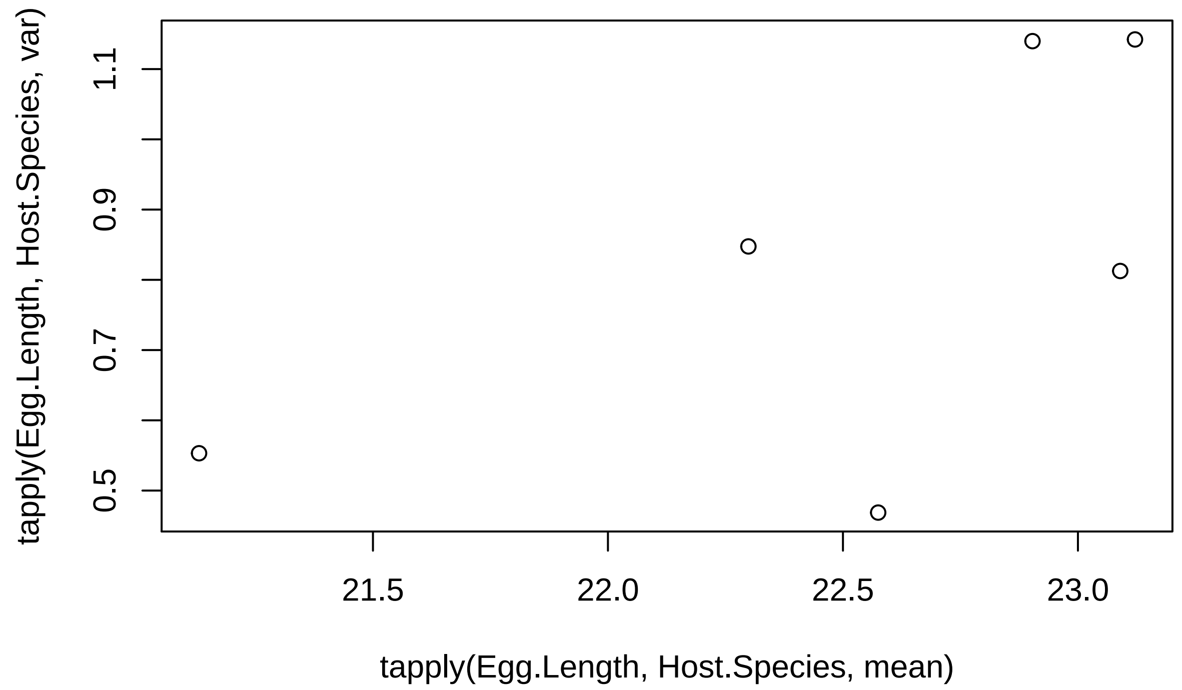
Once you have a significant result with the ANOVA test, determined by the ANOVA table which can be viewed with the *summary* command, you will want to track down which of the groups are actually different in their mean values. You do this by using the Tukey Honestly Significant test.

```
summary(result.aov)
```

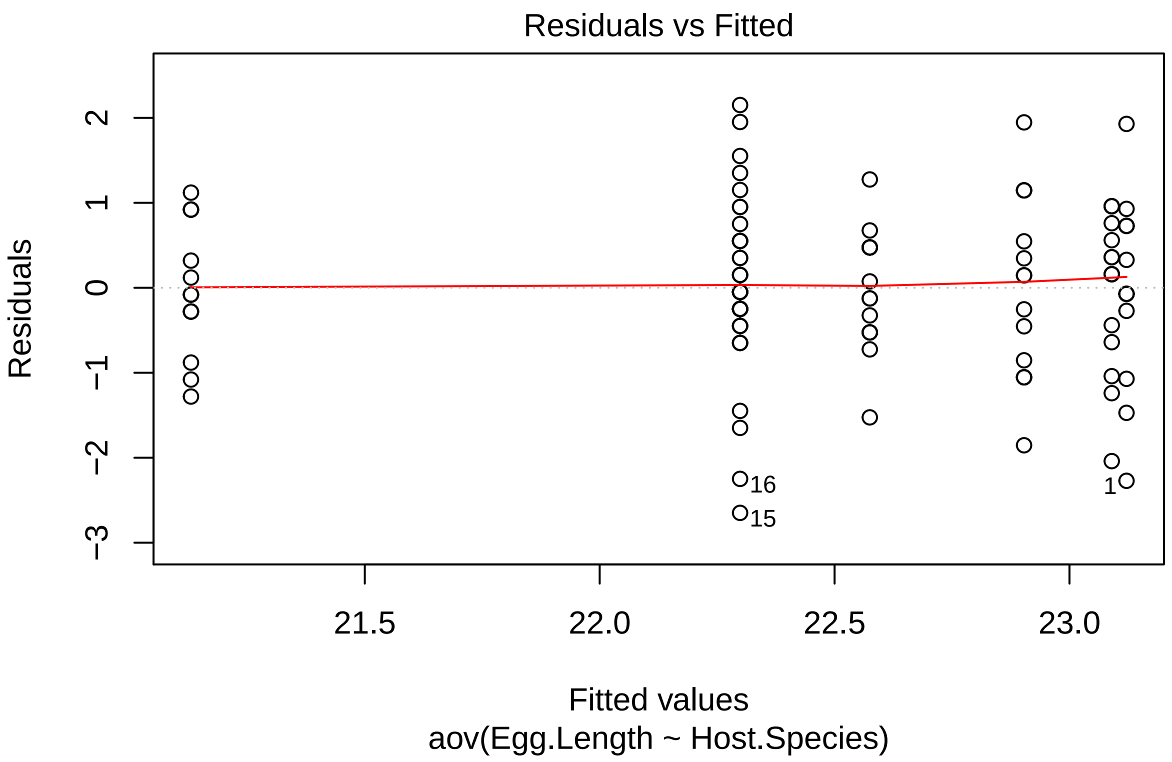
```
##             Df  Sum Sq Mean Sq F value Pr(>F)
## Maize_yield  2  238100  119050   0.415  0.666
## Residuals   18 5158481  286582
```

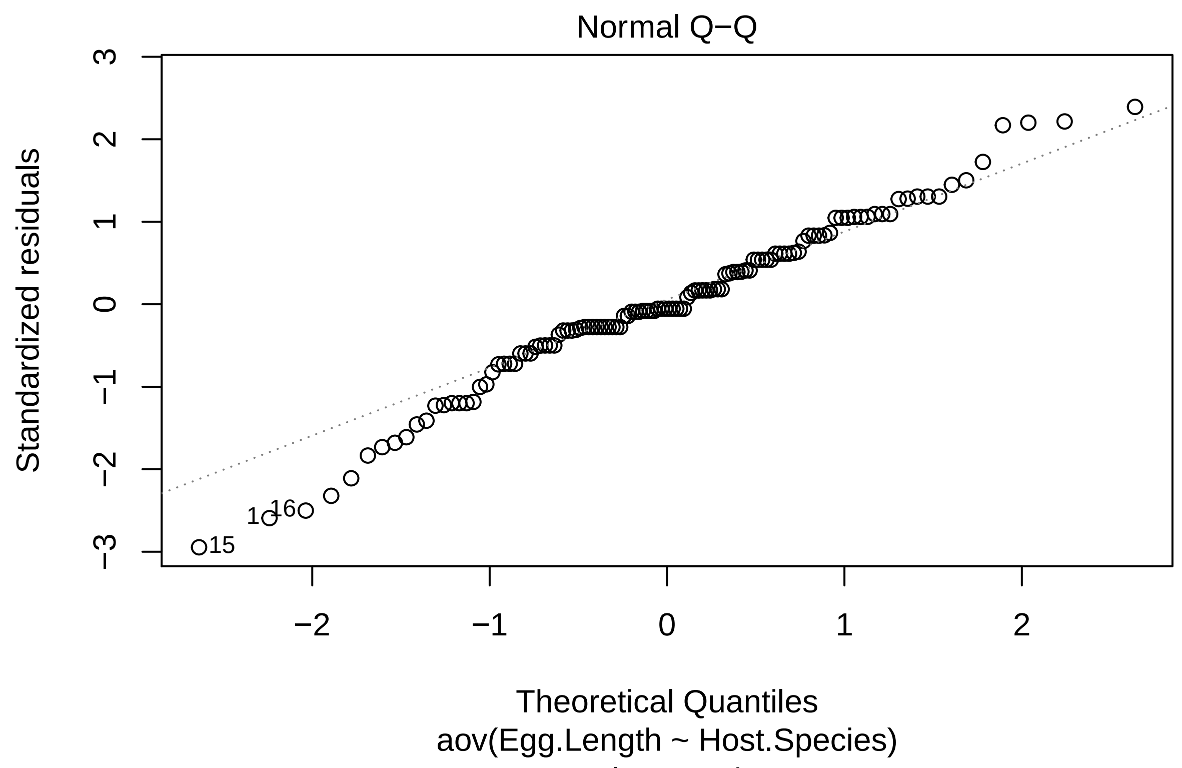
The summary of the malaria data does not allow us to reject the null hypothesis of equality of means between groups. Let's see what happens with we explore a new data set.

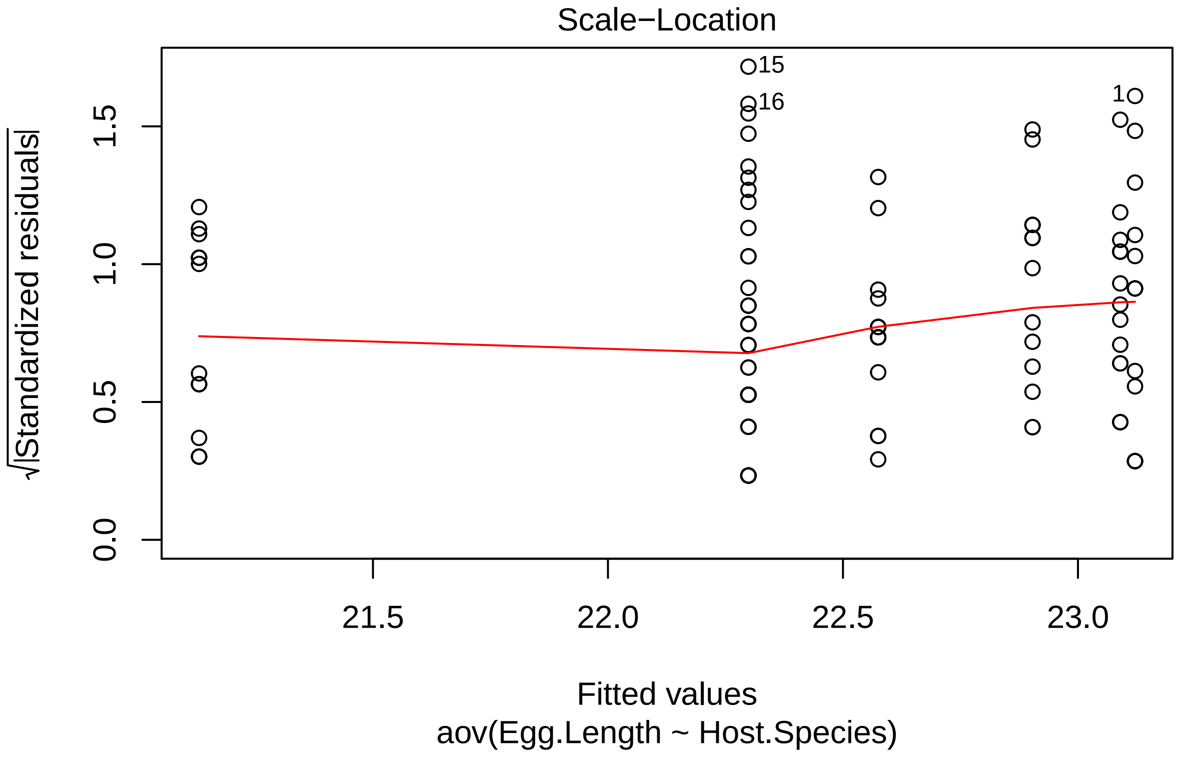
```
cuckoo_eggs<-read.csv("~/Desktop/Bio214/BIOL300 data sets/cuckooeggs.csv")
attach(cuckoo_eggs)
plot(tapply(Egg.Length,Host.Species,mean),tapply(Egg.Length,Host.Species,var))
```

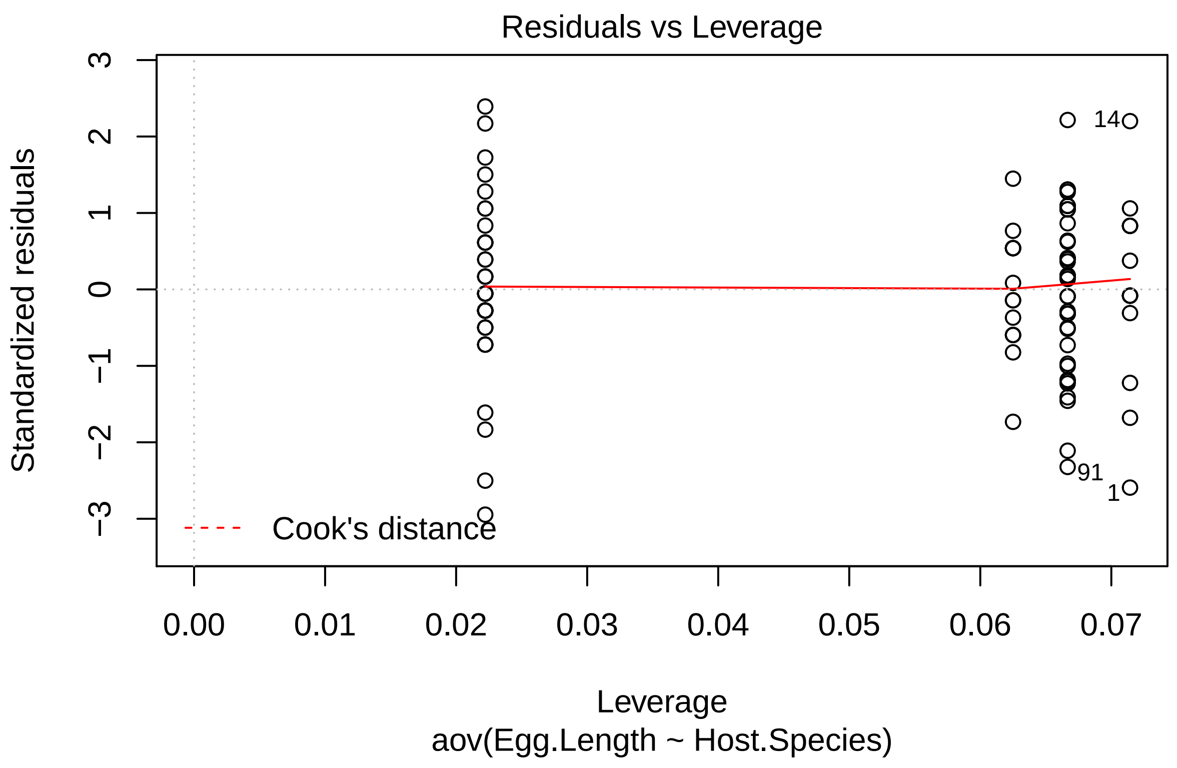


```r
Cuckoo_length.aov<-aov(Egg.Length~Host.Species)
plot(Cuckoo_length.aov)
```




Normal Q−Q
Standardized residuals
Theoretical Quantiles
aov(Egg.Length ~ Host.Species)
15
16
1


Scale−Location
√|Standardized residuals|
Fitted values
aov(Egg.Length ~ Host.Species)
15
16
1



```r
summary(Cuckoo_length.aov)
```

```
##               Df Sum Sq Mean Sq F value   Pr(>F)    
## Host.Species   5  42.94   8.588   10.39 3.15e-08 ***
## Residuals    114  94.25   0.827                     
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```r
TukeyHSD(Cuckoo_length.aov)
```

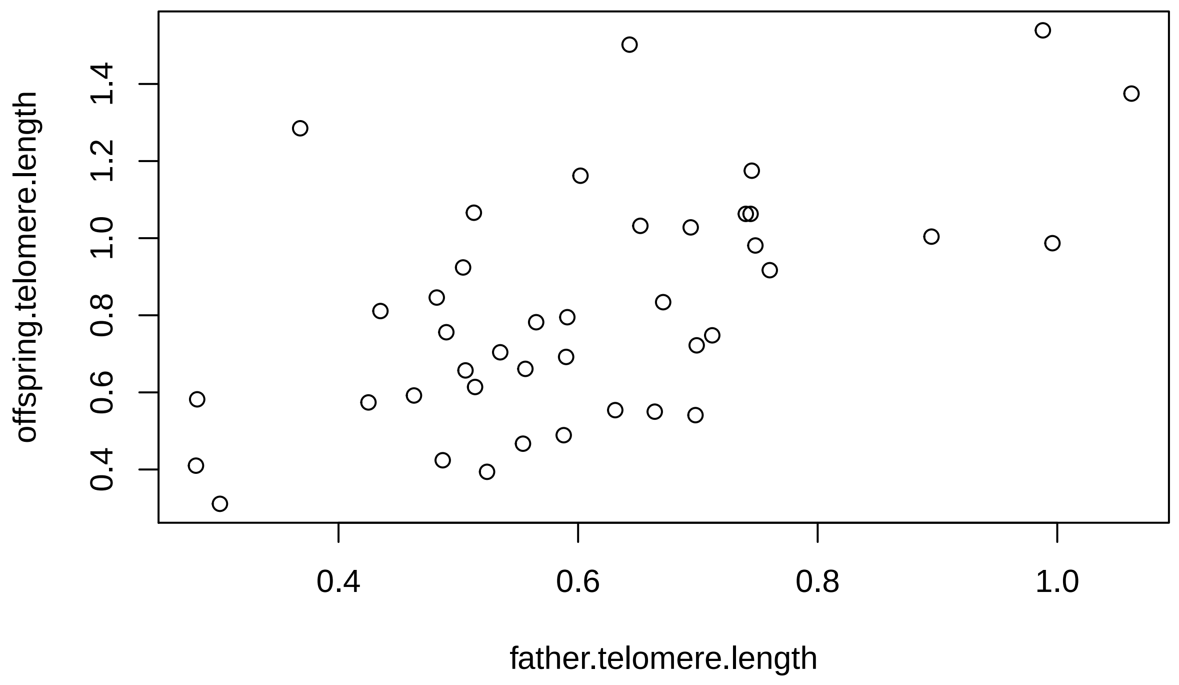
```
##   Tukey multiple comparisons of means
##     95% family-wise confidence level
## 
## Fit: aov(formula = Egg.Length ~ Host.Species)
## 
## $Host.Species
##                                   diff          lwr         upr     p adj
## Meadow Pipit-Hedge Sparrow -0.82253968 -1.629133605 -0.01594576 0.0428621
## Pied Wagtail-Hedge Sparrow -0.21809524 -1.197559436  0.76136896 0.9872190
## Robin-Hedge Sparrow        -0.54642857 -1.511003196  0.41814605 0.5726153
## Tree Pipit-Hedge Sparrow   -0.03142857 -1.010892769  0.94803563 0.9999990
## Wren-Hedge Sparrow         -1.99142857 -2.970892769 -1.01196437 0.0000006
## Pied Wagtail-Meadow Pipit   0.60444444 -0.181375330  1.39026422 0.2324603
## Robin-Meadow Pipit          0.27611111 -0.491069969  1.04329219 0.9021876
## Tree Pipit-Meadow Pipit     0.79111111  0.005291337  1.57693089 0.0474619
## Wren-Meadow Pipit          -1.16888889 -1.954708663 -0.38306911 0.0004861
## Robin-Pied Wagtail         -0.32833333 -1.275604766  0.61893810 0.9155004
## Tree Pipit-Pied Wagtail     0.18666667 -0.775762072  1.14909541 0.9932186
## Wren-Pied Wagtail          -1.77333333 -2.735762072 -0.81090459 0.0000070
## Tree Pipit-Robin            0.51500000 -0.432271433  1.46227143 0.6159630
## Wren-Robin                 -1.44500000 -2.392271433 -0.49772857 0.0003183
## Wren-Tree Pipit            -1.96000000 -2.922428738 -0.99757126 0.0000006
```

Onto the last type of test that most of you may find useful: correlation and regression. Once again, we will need to ensure that our data sets comply with the assumptions of each of these tests. For correlation, that simply means that we should confirm bivariate normality with boxplots (or qqnorm plots) along with a relationship that is best described as linear. Linear regression additionally requires homogeneity of variance which can be determined with the resid function (similar to aov) that belongs to the *lm* class of functions.

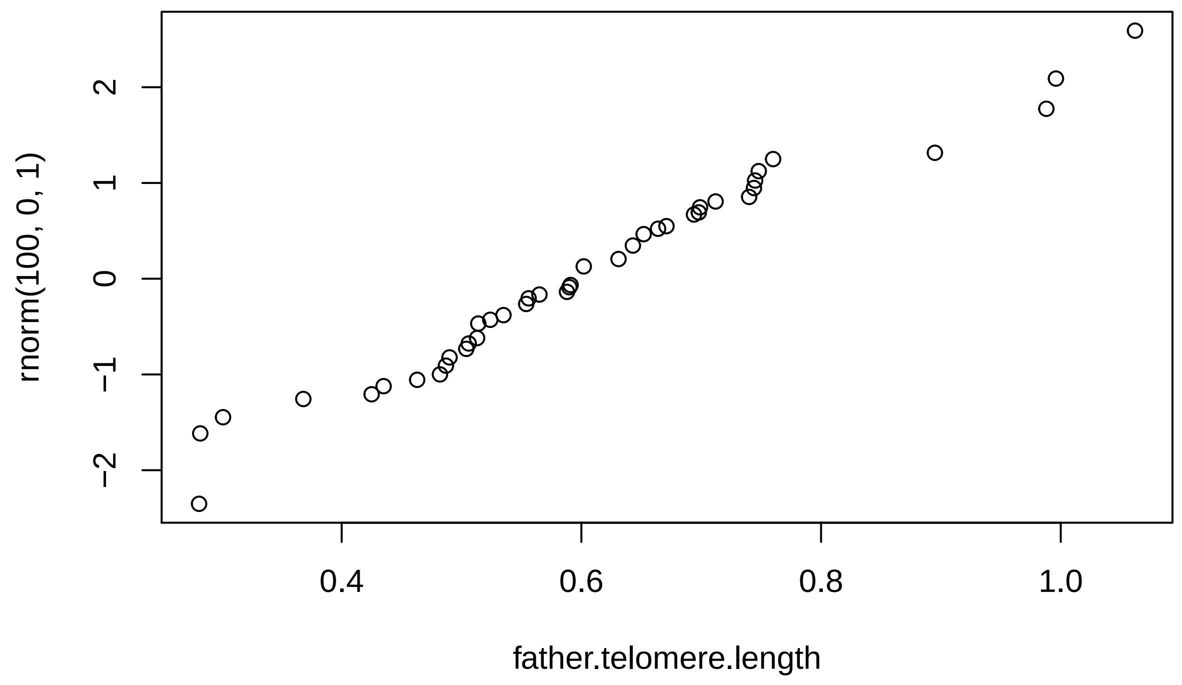
```r
telomere_inheritance<-read.csv("~/Desktop/Bio214/BIOL300 data sets/telomere inheritance.csv")
attach(telomere_inheritance)
telomere_inheritance
```

```
##    father.telomere.length offspring.telomere.length
## 1                   0.281                     0.410
## 2                   0.301                     0.311
## 3                   0.282                     0.582
## 4                   0.425                     0.574
## 5                   0.463                     0.592
## 6                   0.514                     0.614
## 7                   0.506                     0.657
## 8                   0.535                     0.704
## 9                   0.556                     0.661
## 10                  0.590                     0.692
## 11                  0.490                     0.756
## 12                  0.435                     0.811
## 13                  0.482                     0.846
## 14                  0.504                     0.924
## 15                  0.524                     0.394
## 16                  0.487                     0.424
## 17                  0.554                     0.467
## 18                  0.588                     0.489
## 19                  0.631                     0.554
## 20                  0.664                     0.550
## 21                  0.698                     0.541
## 22                  0.513                     1.066
## 23                  0.368                     1.285
## 24                  0.602                     1.162
## 25                  0.745                     1.175
## 26                  0.740                     1.063
## 27                  0.744                     1.063
## 28                  0.694                     1.028
## 29                  0.652                     1.032
## 30                  0.748                     0.981
## 31                  0.760                     0.917
## 32                  0.895                     1.004
## 33                  0.996                     0.987
## 34                  0.699                     0.722
## 35                  0.712                     0.748
## 36                  0.671                     0.834
## 37                  0.591                     0.795
## 38                  0.565                     0.782
## 39                  0.643                     1.502
## 40                  0.988                     1.539
## 41                  1.062                     1.375
```

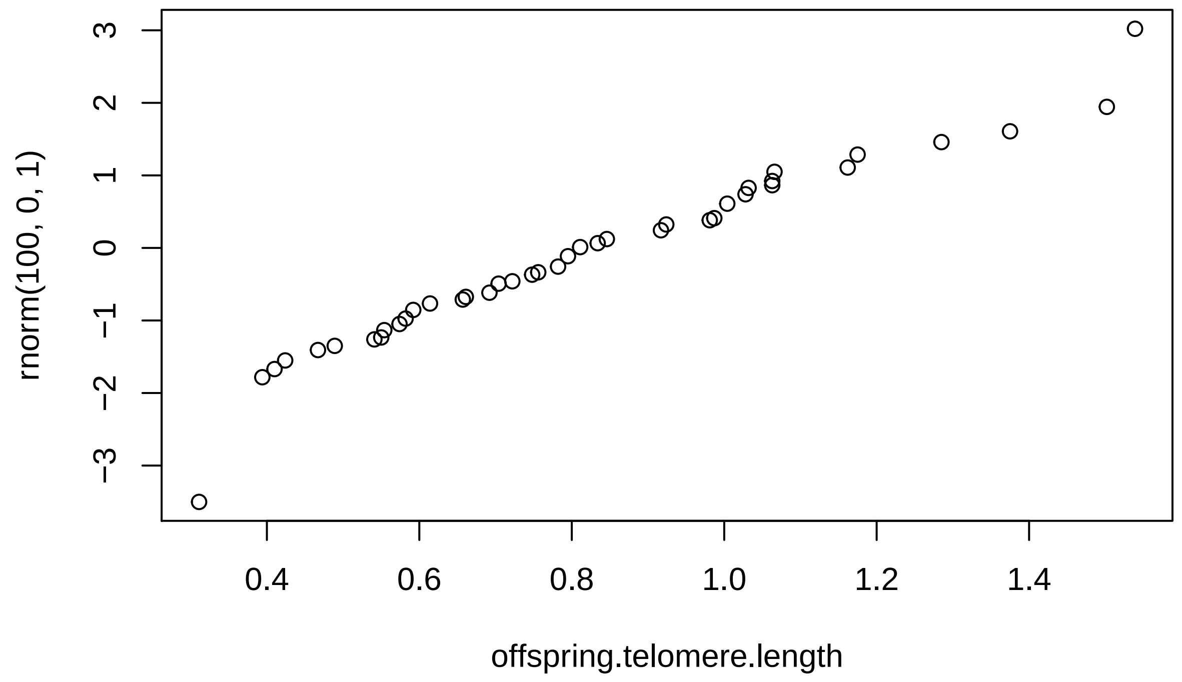
```r
plot(father.telomere.length,offspring.telomere.length)
```



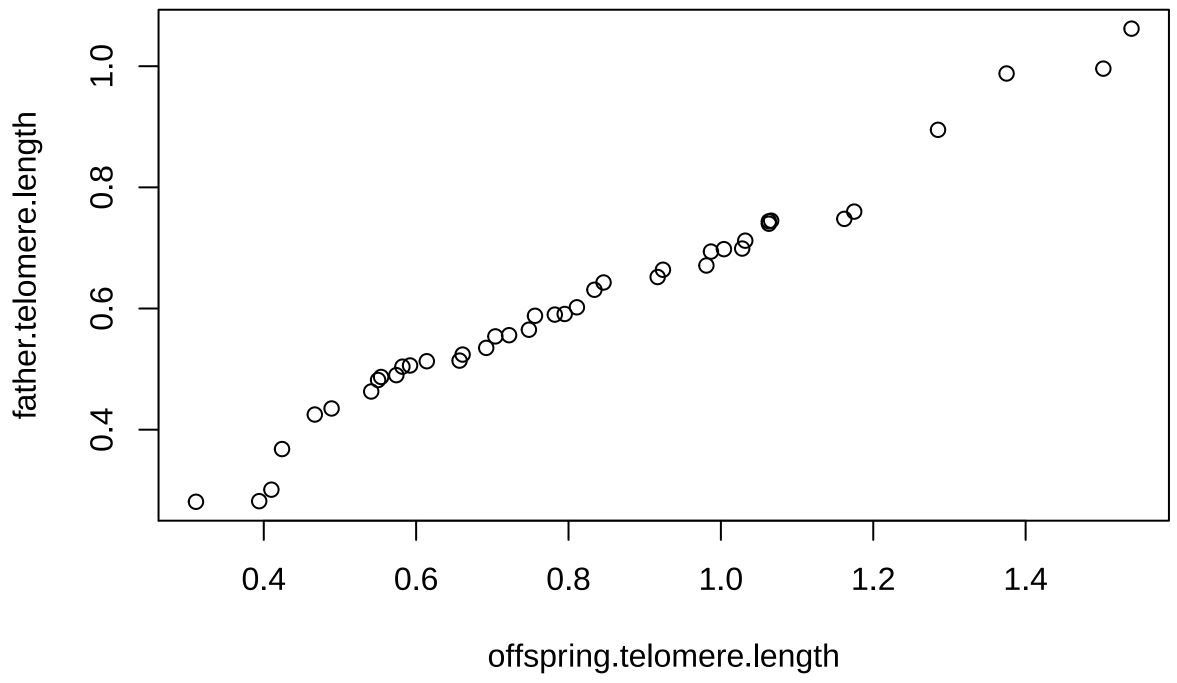
```r
qqplot(father.telomere.length,rnorm(100,0,1))
```



```r
qqplot(offspring.telomere.length,rnorm(100,0,1))
```



```r
qqplot(offspring.telomere.length,father.telomere.length)
```



```r
shapiro.test(offspring.telomere.length)
```

```
##
##  Shapiro-Wilk normality test
##
## data:  offspring.telomere.length
## W = 0.96542, p-value = 0.2424
```

```r
shapiro.test(father.telomere.length)
```

```
##
##  Shapiro-Wilk normality test
##
## data:  father.telomere.length
## W = 0.9609, p-value = 0.1691
```

```
cor.test(~father.telomere.length+offspring.telomere.length)
```

```
##
##  Pearson's product-moment correlation
##
## data:  father.telomere.length and offspring.telomere.length
## t = 4.4358, df = 39, p-value = 7.287e-05
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  0.3302762 0.7526477
## sample estimates:
##      cor
## 0.579086
```

If your data had serious deviations from normality or linearity, you would have specific either a spearman nonparametric test or a kendall nonparametric test depending on your sample size.

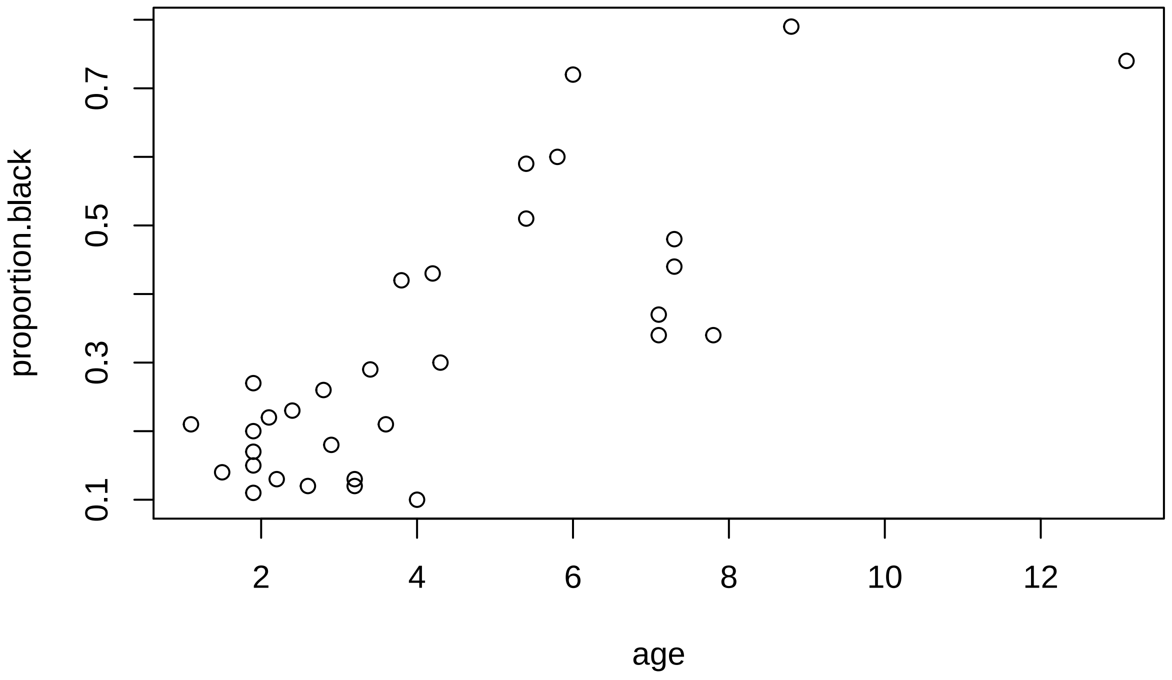
Lastly, we will now tackle a more sophisticated test: linear regression.

Once again, we begin by loading our dataset and testing assumptions:

```
Nose_colour<-read.csv("~/Desktop/Bio214/BIOL300 data sets/17e1LionAges.csv")
attach(Nose_colour)
Nose_colour
```

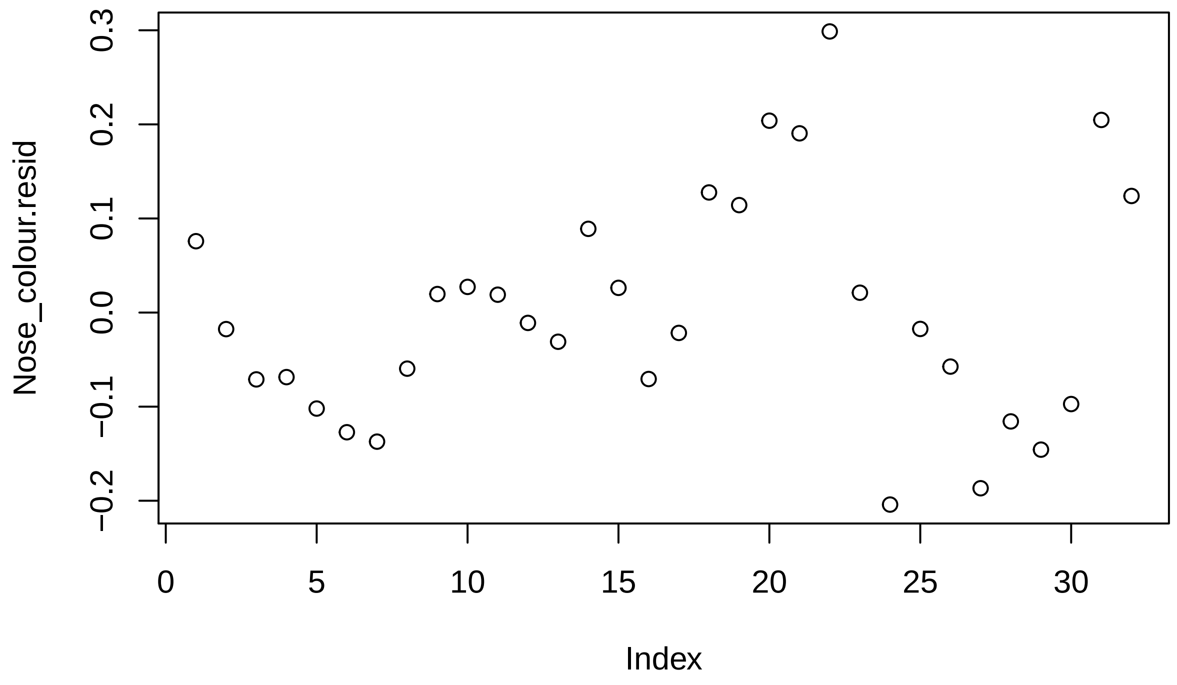
```
##     age proportion.black
## 1   1.1             0.21
## 2   1.5             0.14
## 3   1.9             0.11
## 4   2.2             0.13
## 5   2.6             0.12
## 6   3.2             0.13
## 7   3.2             0.12
## 8   2.9             0.18
## 9   2.4             0.23
## 10  2.1             0.22
## 11  1.9             0.20
## 12  1.9             0.17
## 13  1.9             0.15
## 14  1.9             0.27
## 15  2.8             0.26
## 16  3.6             0.21
## 17  4.3             0.30
## 18  3.8             0.42
## 19  4.2             0.43
## 20  5.4             0.59
## 21  5.8             0.60
## 22  6.0             0.72
## 23  3.4             0.29
## 24  4.0             0.10
## 25  7.3             0.48
## 26  7.3              0.44
## 27  7.8              0.34
## 28  7.1              0.37
## 29  7.1              0.34
## 30 13.1              0.74
## 31  8.8              0.79
## 32  5.4              0.51
```

```
plot(age,proportion.black)
```



We ask the question *is the amount of black on a lions nose dependent on their age?*

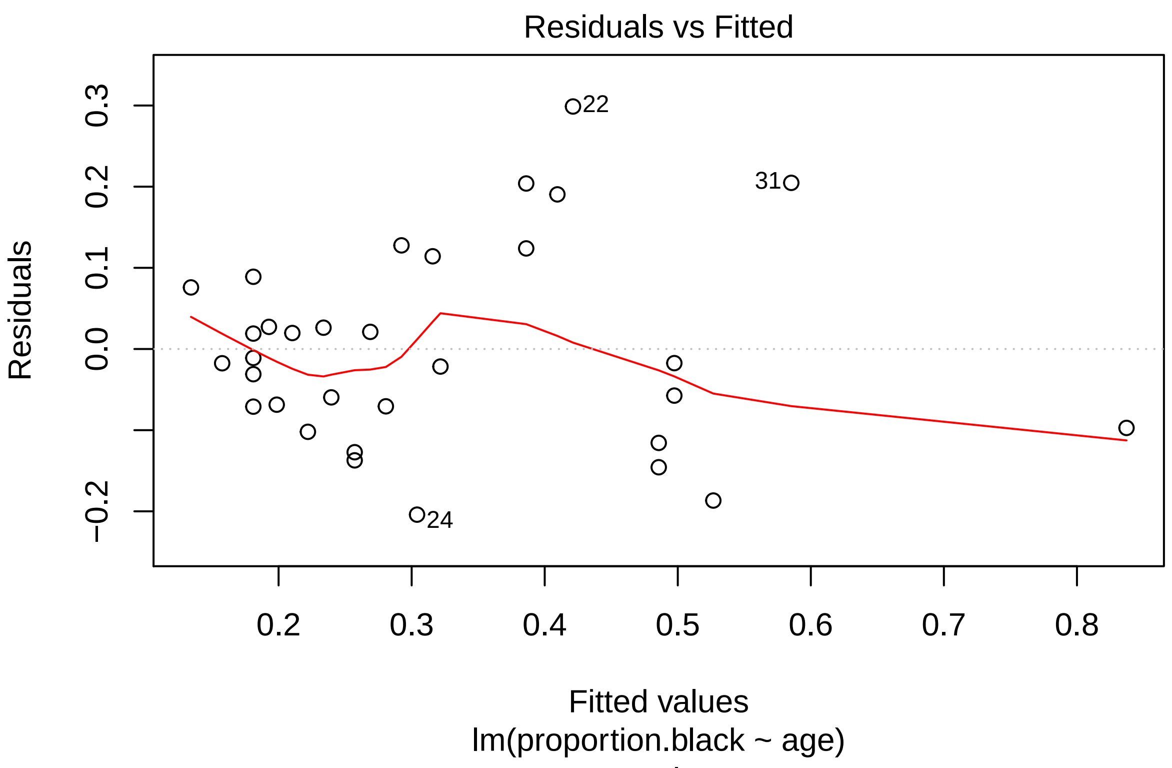
```
Nose_colour.lm<-lm(proportion.black~age)
Nose_colour.resid<-resid(Nose_colour.lm)
plot(Nose_colour.resid)
```

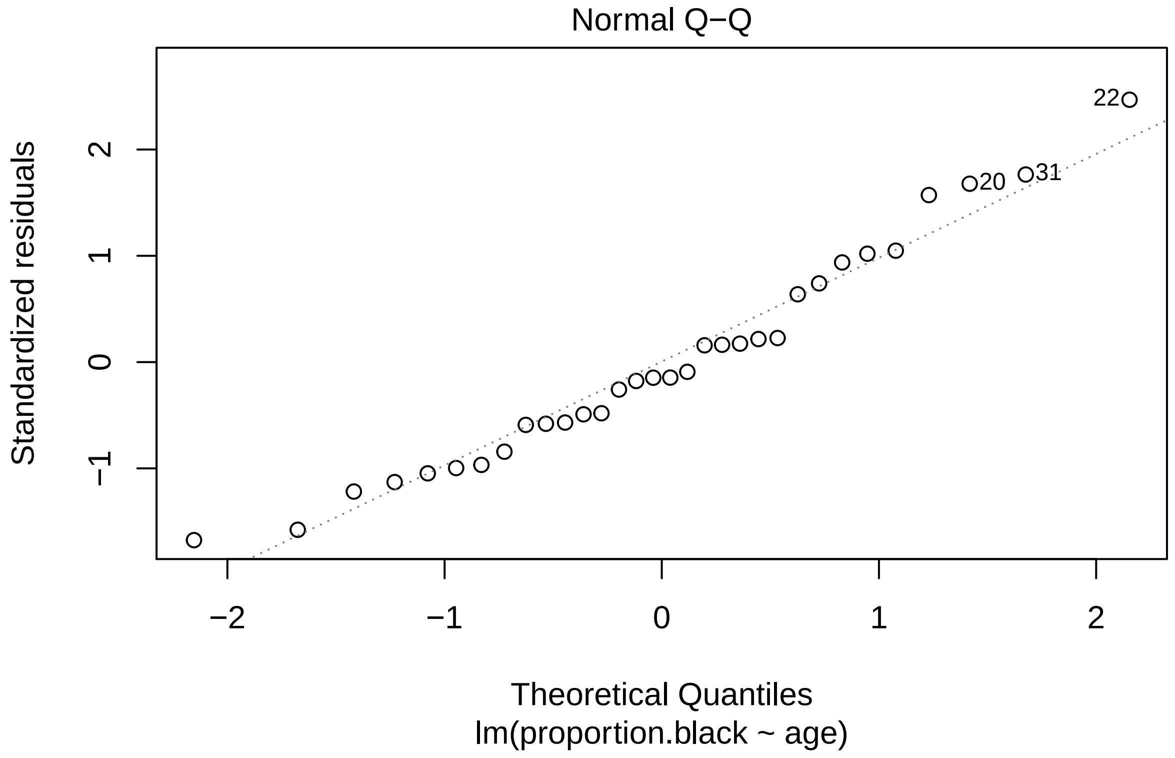


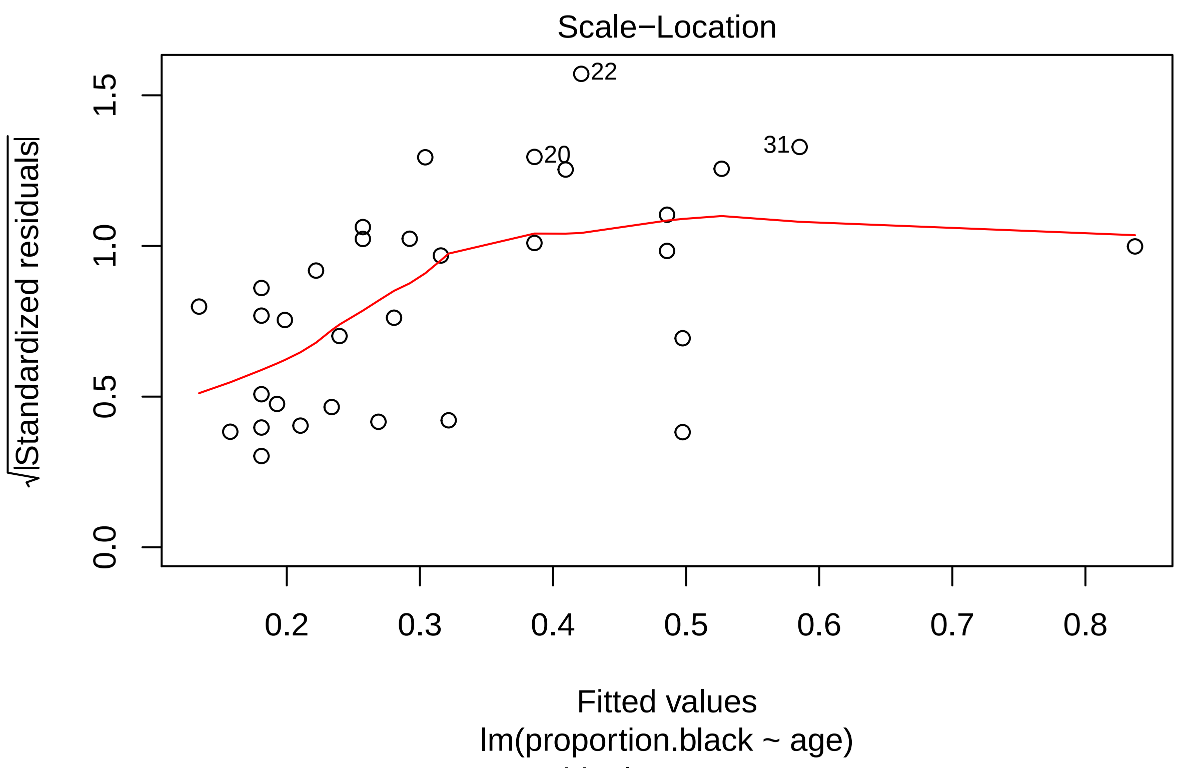
```r
summary(Nose_colour.lm)
```

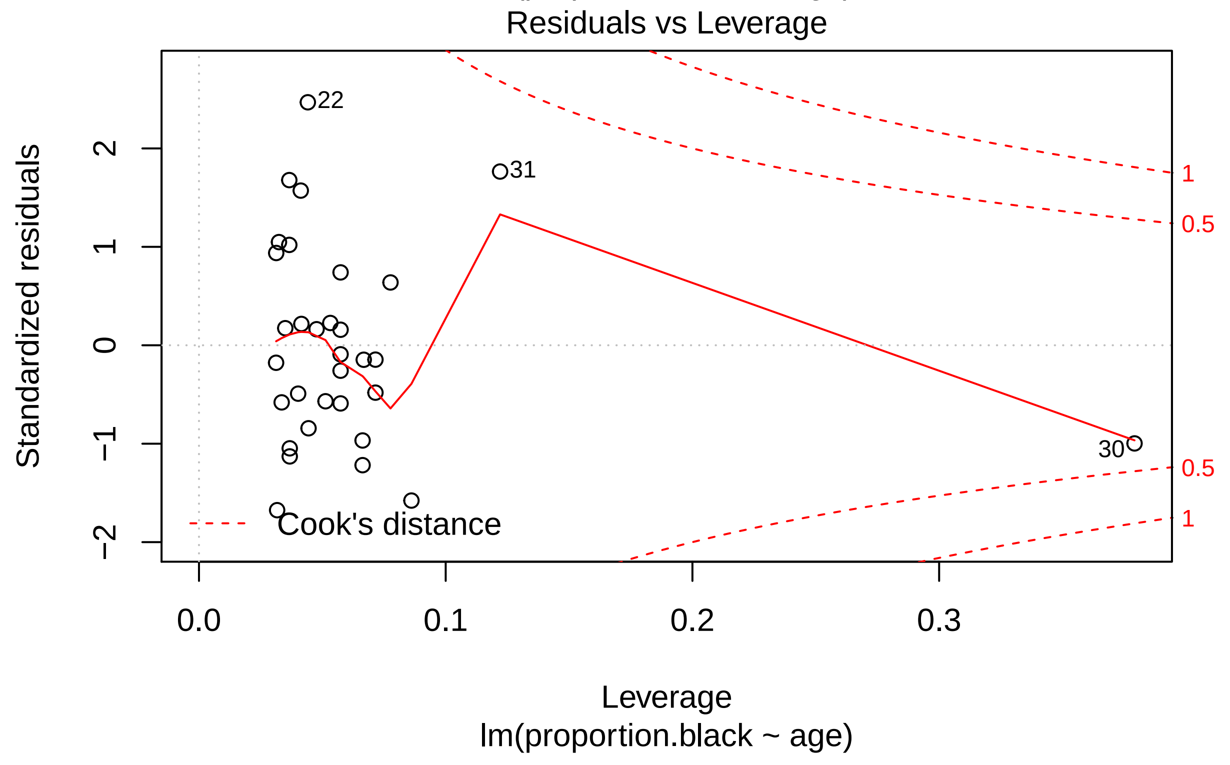
```
##
## Call:
## lm(formula = proportion.black ~ age)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.20406 -0.07758 -0.01750  0.07913  0.29876
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 0.069696   0.041956   1.661    0.107
## age         0.058591   0.008307   7.053 7.68e-08 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.1238 on 30 degrees of freedom
## Multiple R-squared:  0.6238, Adjusted R-squared:  0.6113
## F-statistic: 49.75 on 1 and 30 DF,  p-value: 7.677e-08
```

```r
plot(Nose_colour.lm)
```


Residuals vs Fitted
Residuals
0.3
0.2
0.1
0.0
−0.2
0.2
0.3
0.4
0.5
0.6
0.7
0.8
22
31
24
Fitted values
lm(proportion.black ~ age)


Normal Q−Q
Standardized residuals
2
1
0
−1
−2
−1
0
1
2
22
20
31
Theoretical Quantiles
lm(proportion.black ~ age)





Remember at the very end of your session to detach all of the datasets that you have attached in the beginning in order to free up the variable names for future analysis.

```r
detach(cardiac)
detach(BP)
detach(NHL)
detach(malariaMaize)
detach(cuckoo_eggs)
detach(telomere_inheritance)
detach(Nose_colour)
```

In order to ensure that you are saving your file in html format you should do the following: first, “Knit” your file. This should cause a window to open up containing your html file. A copy of your html file will be placed wherever the original .rmd file was saved.

13.2 And now for some ggplot2 package graphs

RBook_ggplot2

Danielle A Presgraves

11/7/2017

we will attempt to accomplish the following in today's lecture: 1. Review RStudio commands for basic plotting functions 2.a. Introduce ggplot2 package for slightly easier (or at least different) plotting and data manipulations. b. Download ggplot2 package

We will need to load the ggplot2 package. This is a package, developed by the R guru Hadley Wickham (his personal website is here:http://hadley.nz; a website that shows you how to obtain ggplot2 is here: http://ggplot2.org; finally, a website that contains basic documentation for ggplot2 is here: http://docs.ggplot2.org/current/), that should be easier to use on complex data than the base packages of R.

You will need to use the following command to install a package:

```
#install.packages("ggplot2")
```

You will probably get asked to choose a CRAN mirror - this doesn't matter that much. then you will need to load the package by either checking it off in the packages panel or typing library("ggplot2") at the command prompt in the console.

We are going to manipulate a dataset that is built into ggplot2:

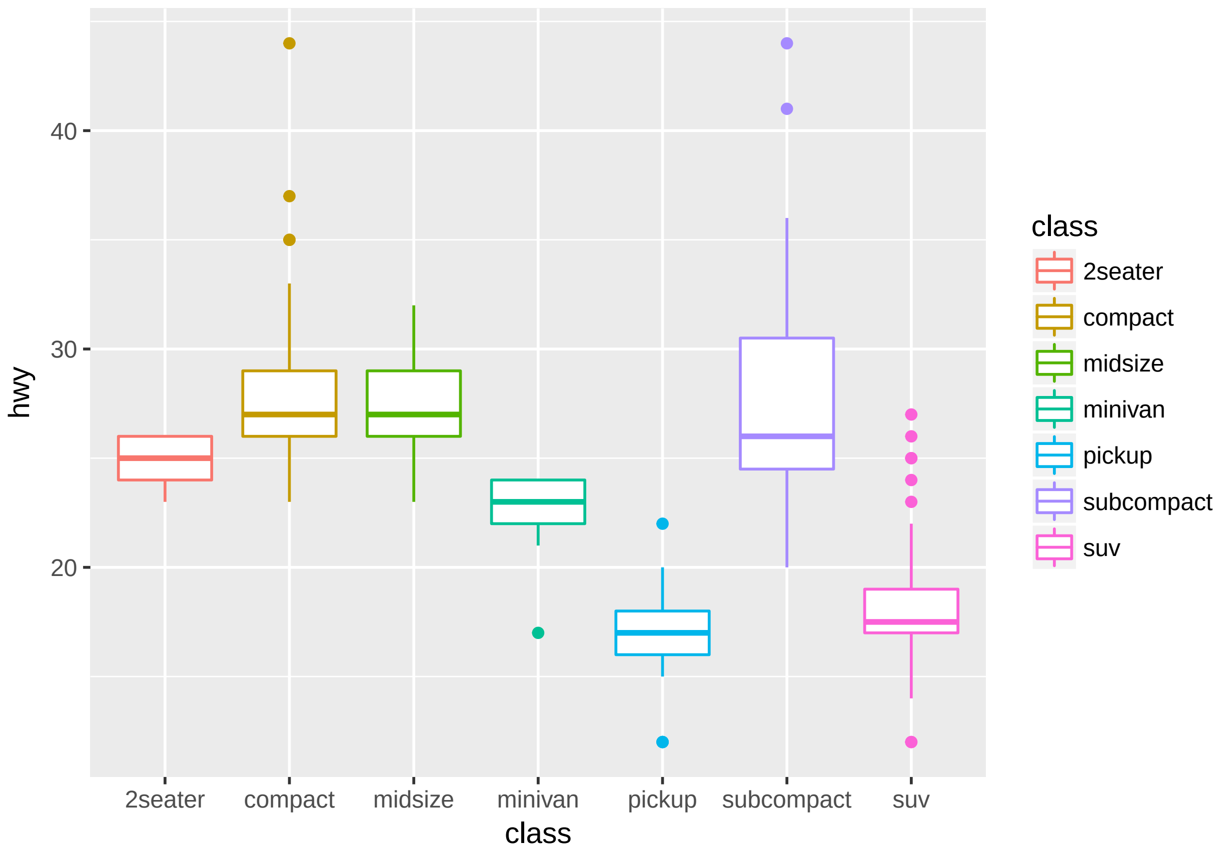
```
library(ggplot2)
mpg
```

```
## # A tibble: 234 x 11
##    manufacturer      model displ  year   cyl      trans   drv   cty   hwy
##           <chr>      <chr> <dbl> <int> <int>      <chr> <chr> <int> <int>
##  1         audi         a4   1.8  1999     4   auto(l5)     f    18    29
##  2         audi         a4   1.8  1999     4 manual(m5)     f    21    29
##  3         audi         a4   2.0  2008     4 manual(m6)     f    20    31
##  4         audi         a4   2.0  2008     4   auto(av)     f    21    30
##  5         audi         a4   2.8  1999     6   auto(l5)     f    16    26
##  6         audi         a4   2.8  1999     6 manual(m5)     f    18    26
##  7         audi         a4   3.1  2008     6   auto(av)     f    18    27
##  8         audi a4 quattro   1.8  1999     4 manual(m5)     4    18    26
##  9         audi a4 quattro   1.8  1999     4   auto(l5)     4    16    25
## 10         audi a4 quattro   2.0  2008     4 manual(m6)     4    20    28
## # ... with 224 more rows, and 2 more variables: fl <chr>, class <chr>
```

Remember that we can use head/tail command to see the first six or last six rows of the data set to see what the columns are etc. You can now use the ggplot function. Rememeber when we quickly visited the generic plot function that is present in base R? ggplot is the generic plot function that is part of the ggplot2 package. You always start a plot by using the ggplot function since that creates a coordinate system and then you add 'layers'.

ggplot uses geometrical objects, called *geoms*, to visually represent the dataset. Bar charts use *geom_bar*, histograms use *geom_histogram*, boxplots use *geom_boxplot* and scatterplots use *geom_point*. All of these geoms have a wide range of arguments that can be used with them. There is a cheat sheet here: https://www.rstudio.com/wp-content/uploads/2015/03/ggplot2-cheatsheet.pdf

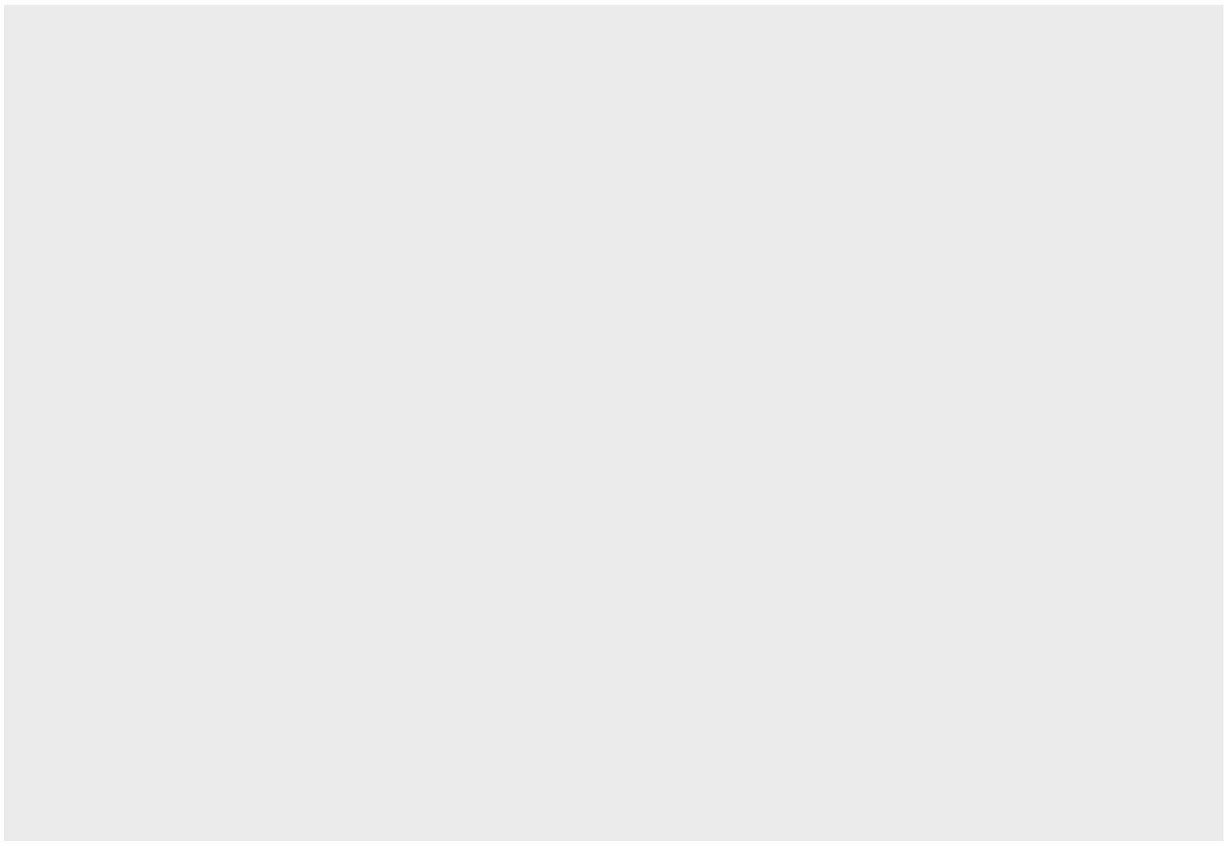
```
ggplot(data = mpg,mapping = aes(x = class,y=hwy,color=class)) + geom_boxplot()
```



In the chunk below, we call the ggplot function on the built in dataset (called mpg) and then we *layer* on the actual points that we are trying to graph.

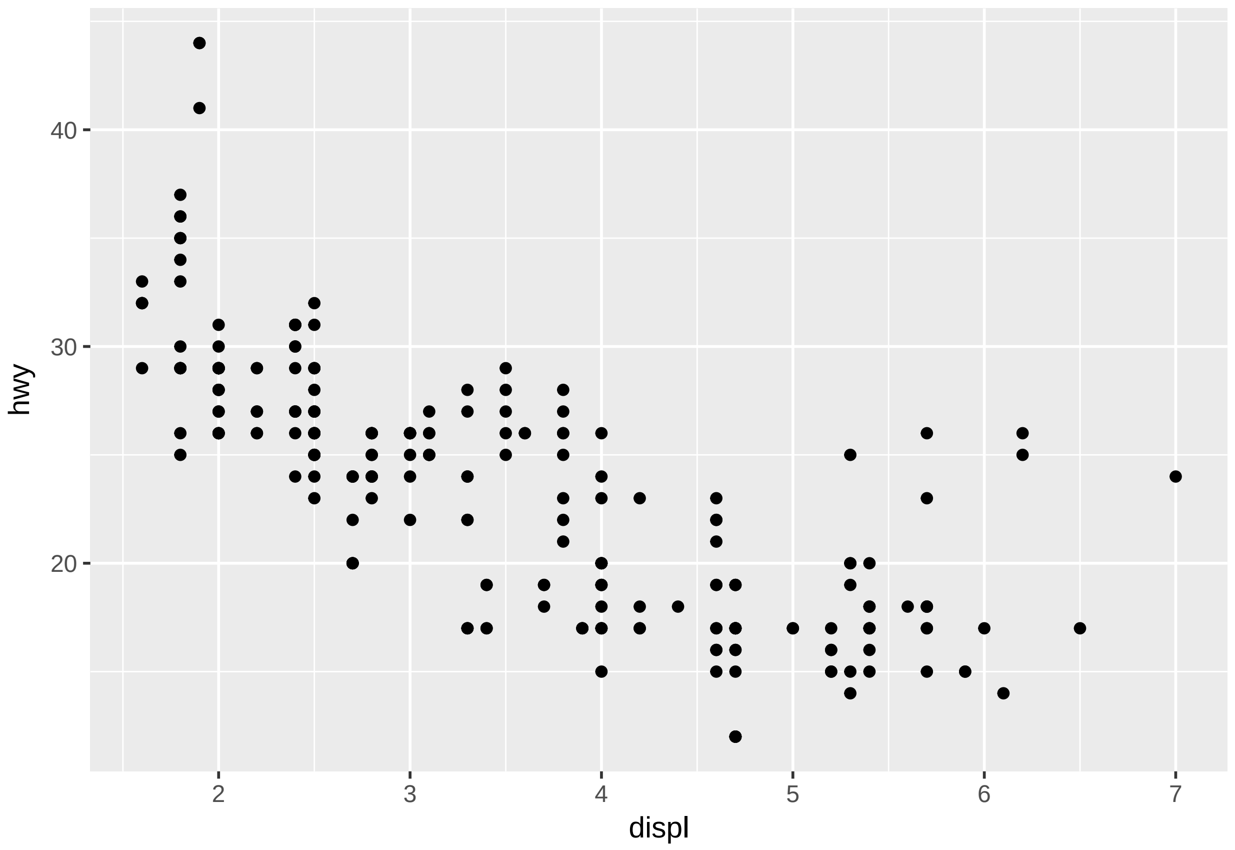
In fact, we could start off by running just the ggplot function on our dataset:

```
ggplot(data=mpg)
```



What does this give us? It *should* give us an empty plot (and an error message) since we haven't actually specified any points to graph. *geom_point* adds points, in a layer, to our axes. Every one of the *geom_something* functions takes a mapping argument which governs the visual features (the *aesthetics* thus 'aes') of each particular layer in your graph including features such as size, shape and colour of your points.

```r
ggplot(data = mpg) + geom_point(mapping = aes(x = displ,y = hwy))
```



The produced plot shows a potentially negative relationship between engine size (displ) and fuel efficiency.

There is a common format that you will use for ggplot: #ggplot(data=data.frame path)+geom_function(mapping=aes(MAPPIN

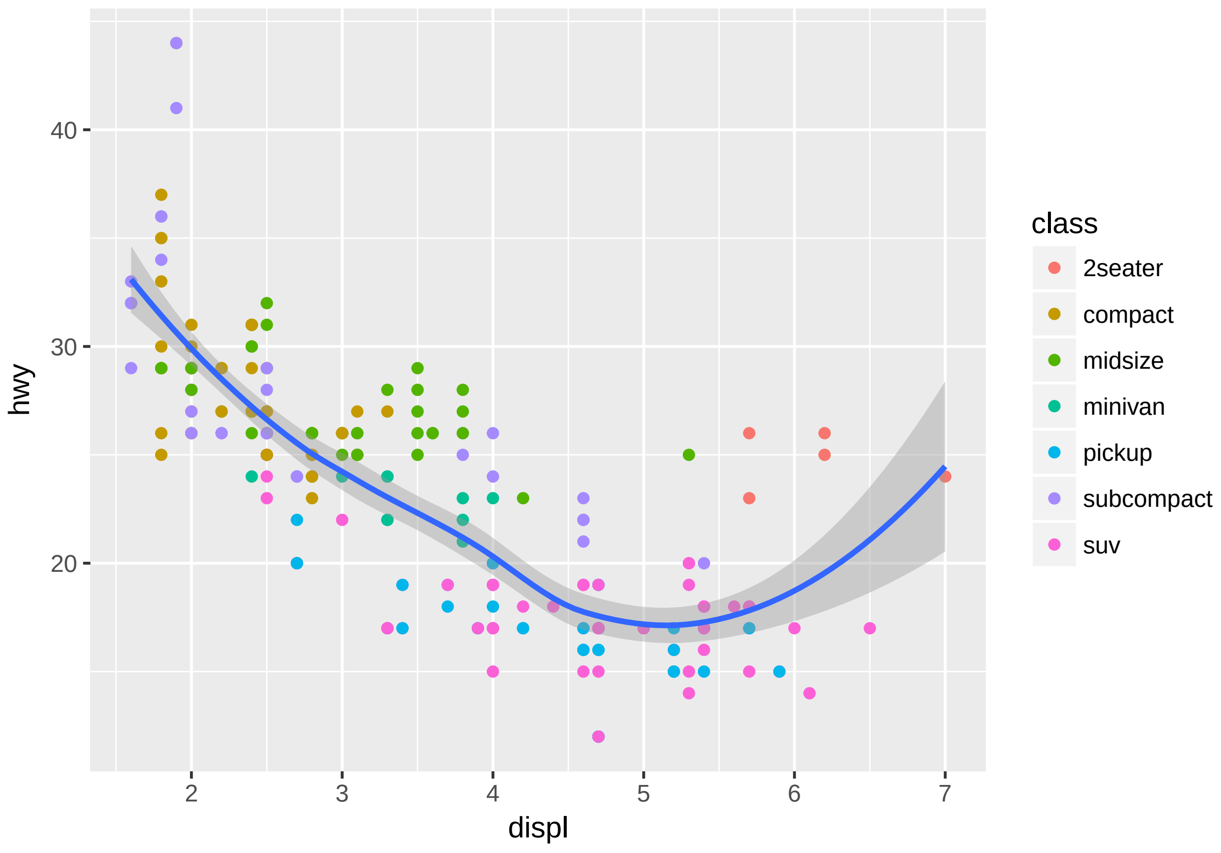
We can specify the colour of each class of car, for instance. We first want to ensure that we are specifying the correct variable - so be sure to look at the column names and the first six rows to get a sense of the categories available. We show two plots on the same data below (one of them uses *geom_point* and the other *geom_smooth*)

```
head(mpg)
```

```
## # A tibble: 6 x 11
##   manufacturer model displ  year   cyl      trans   drv   cty   hwy    fl
##          <chr> <chr> <dbl> <int> <int>      <chr> <chr> <int> <int> <chr>
## 1         audi    a4   1.8  1999     4   auto(l5)     f    18    29     p
## 2         audi    a4   1.8  1999     4 manual(m5)     f    21    29     p
## 3         audi    a4   2.0  2008     4 manual(m6)     f    20    31     p
## 4         audi    a4   2.0  2008     4   auto(av)     f    21    30     p
## 5         audi    a4   2.8  1999     6   auto(l5)     f    16    26     p
## 6         audi    a4   2.8  1999     6 manual(m5)     f    18    26     p
## # ... with 1 more variables: class <chr>
```

```
ggplot(data = mpg,mapping = aes(x = displ, y = hwy)) + geom_point(mapping = aes(x = displ,y = hwy,color
```

```
## `geom_smooth()` using method = 'loess'
```



```
#ggplot(data = mpg) + geom_smooth(mapping = aes(x = displ, y = hwy))
```

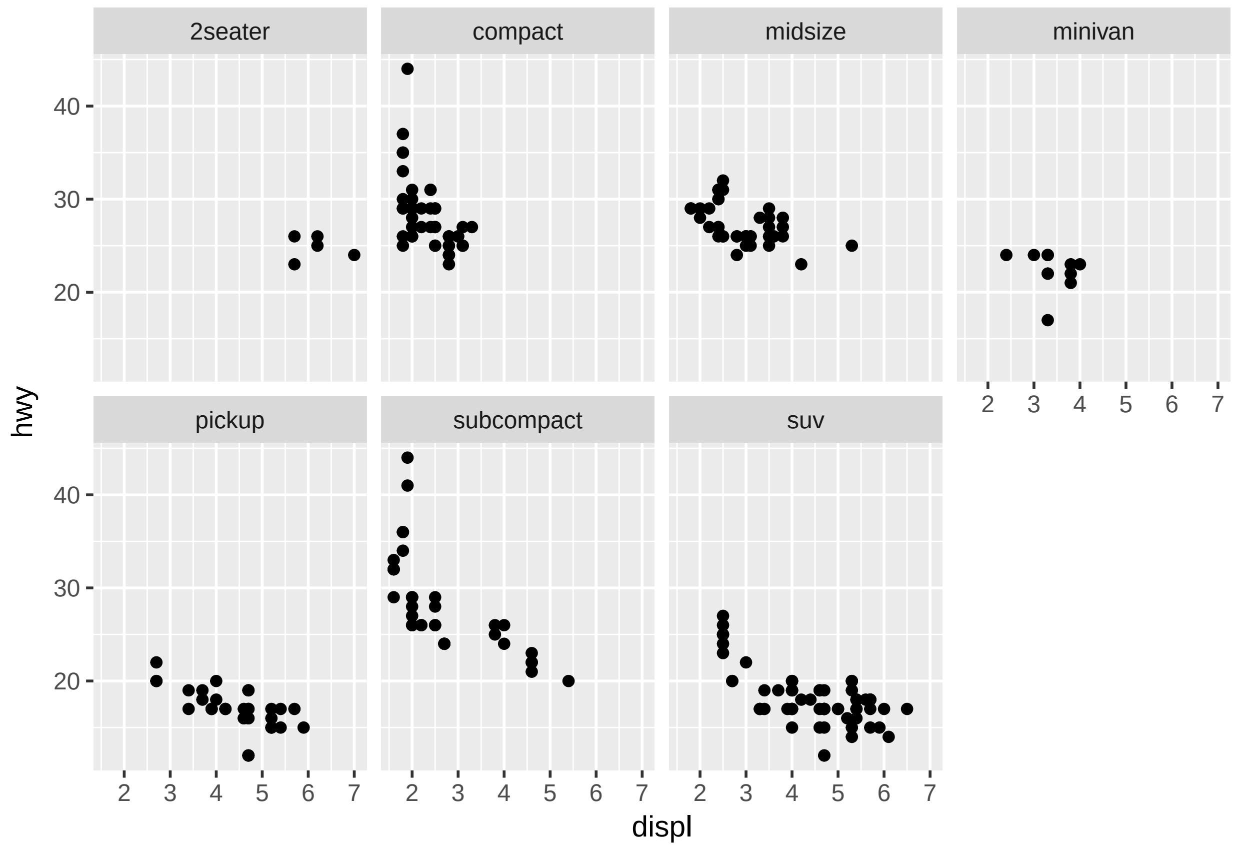
Voila! This produces a legend and everything in a very intuitive way! (This is why people get so excited about ggplot2 - there is still a tiny bit of a learning curve but graphic manipulations are even easier than in basic R).

ggplot uniquely maps a defined aesthetic criteria to a variable using unique values of the variable (sometimes called 'factors'). This is called 'scaling' and includes having R automatically add a lovely legend explaining the correspondance. You can also use size, shape and alpha (transparency of points) - and any combination of these features- but you will need to set these up outside of the aes bracket, like so:

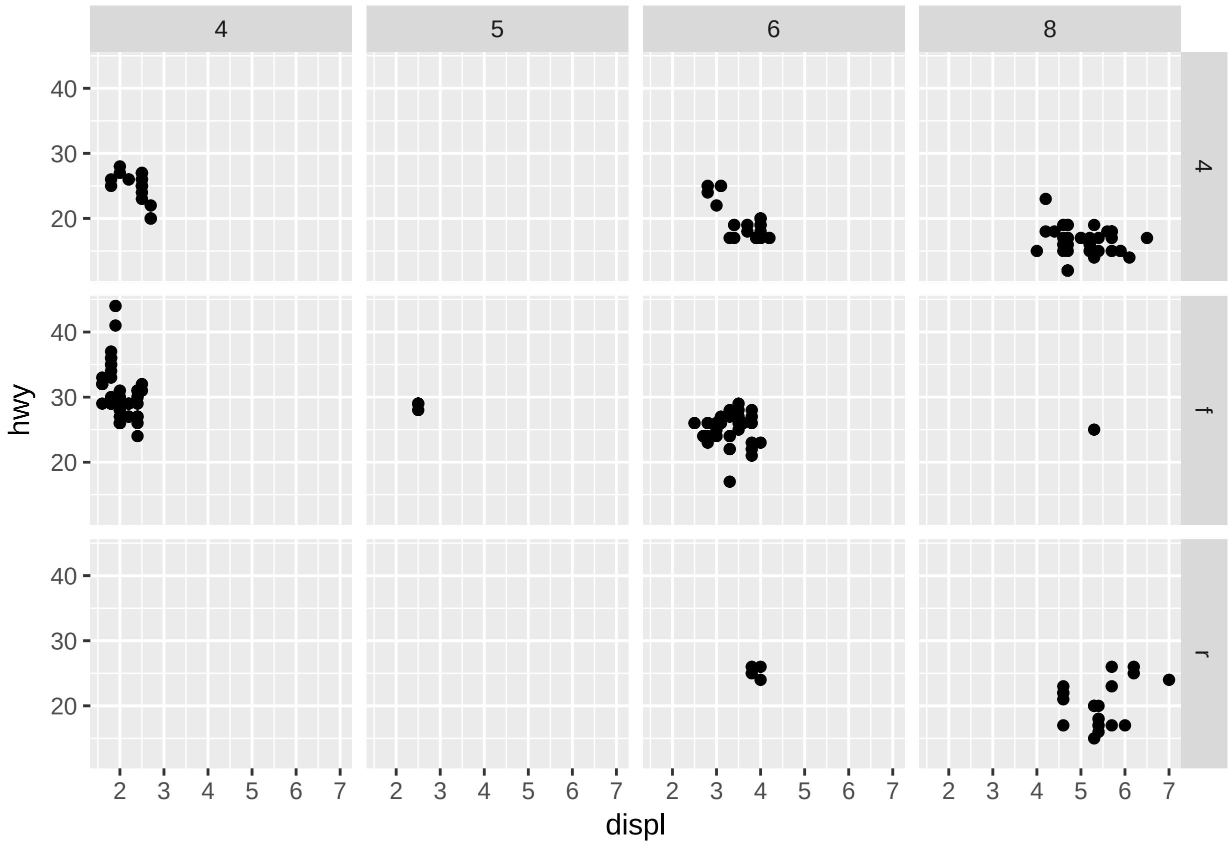
```
color in a string:
ggplot(data = mpg) + geom_point(mapping = aes(x = displ,y = hwy),color = "purple")
or size in mm:
ggplot(data = mpg) + geom_point(mapping = aes(x = displ,y = hwy),size=1.1)
or shape (you can google shapes in ggplot2 for a table of numbers and their corresponding shapes; what
ggplot(data = mpg) + geom_point(mapping = aes(x = displ,y = hwy),shape=11)
Also, remember when I used the elseif statement to specify SBP above 120 in the previous lecture? You c
ggplot(data = mpg) + geom_point(mapping = aes(x = displ,y = hwy,color=displ>5)
```

There is also (of course!) a way to display multiple graphs in the same place, which is particularly useful to do for categorical variables. You need to split your data into subplots using something that ggplot calls a 'facet'. For a discrete categorical variable, you can use *'facet_wrap'* and for a combination of two variables, you can use *'facet_grid'*.

```r
ggplot(data = mpg) + geom_point(mapping = aes(x = displ, y = hwy)) + facet_wrap(~class, nrow = 2)
```



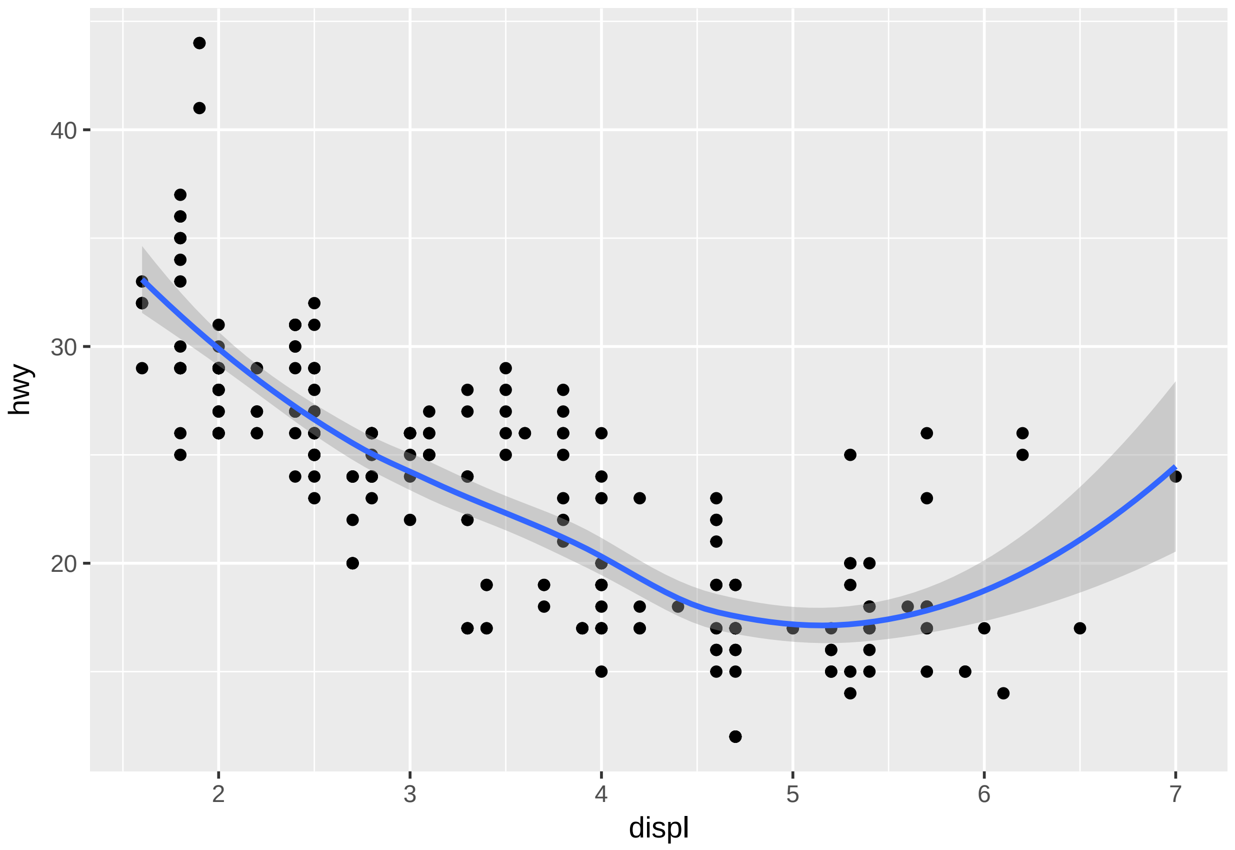
```r
ggplot(data = mpg) + geom_point(mapping = aes(x = displ, y = hwy)) + facet_grid(drv ~ cyl)
```



You can layer multiple geoms on each other, too.

```
ggplot(data = mpg) + geom_point(mapping = aes(x = displ, y = hwy)) +geom_smooth(mapping = aes(x = displ
```

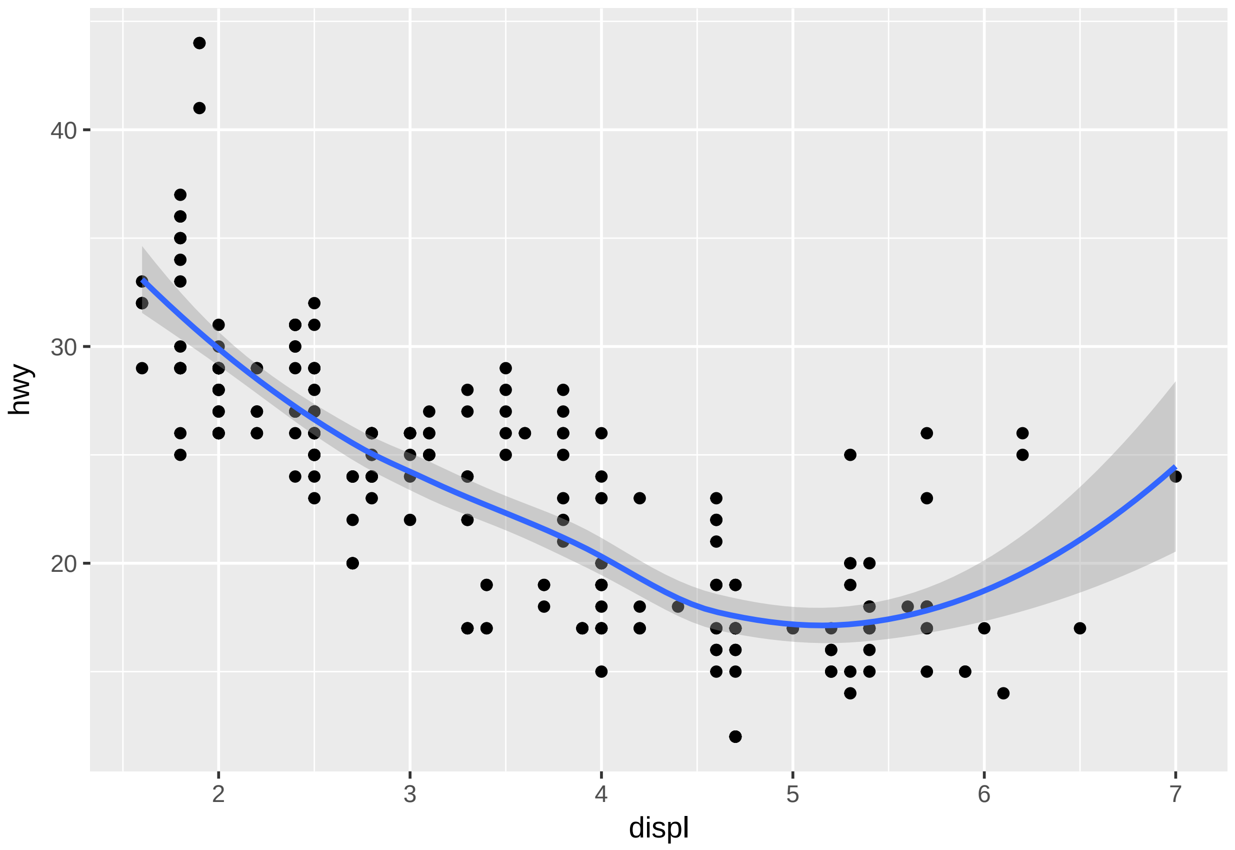
```
## `geom_smooth()` using method = 'loess'
```



Advanced: you can put the universal aesthetic features into the ggplot function and it will apply them to all the layers. You can separately place the aesthetic features that you want associated with ONLY one particular layer in the geom of THAT layer.

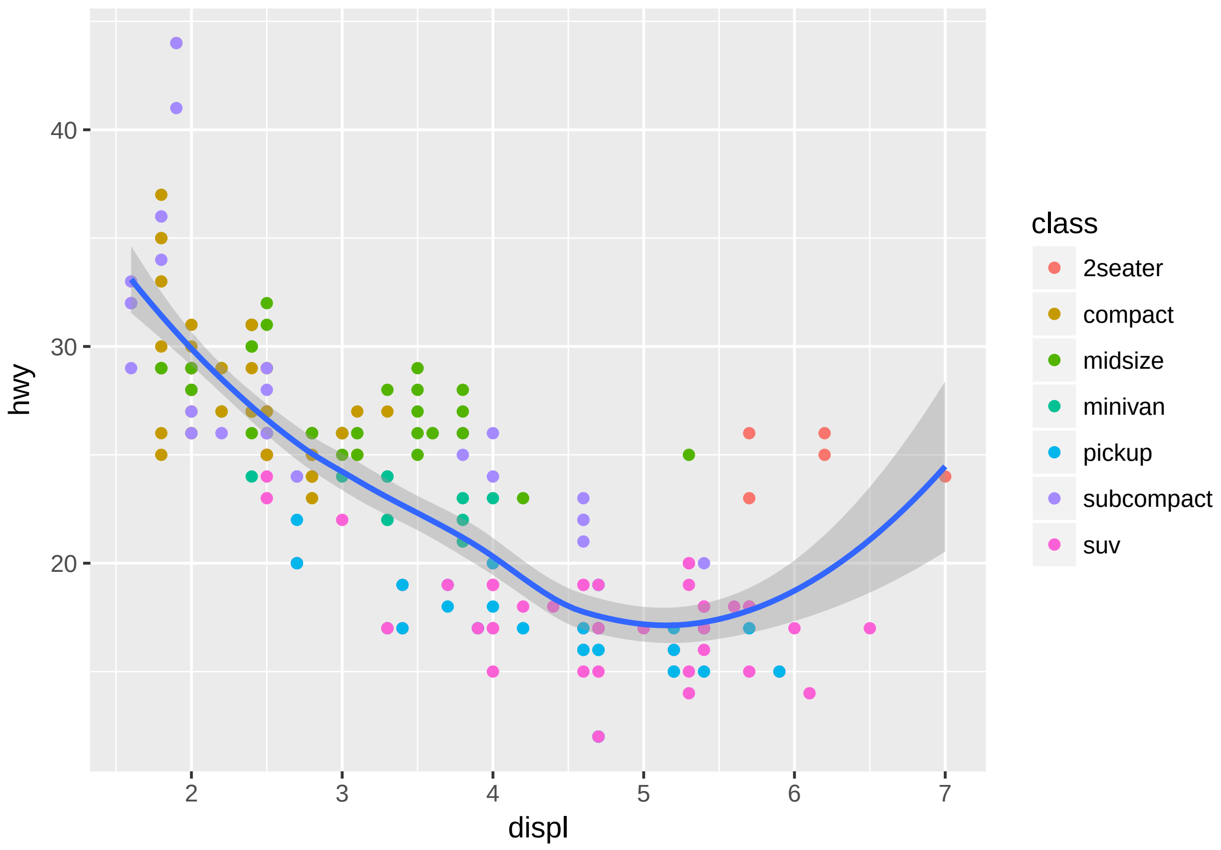
```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) + geom_point() + geom_smooth()
```

```
## `geom_smooth()` using method = 'loess'
```



```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) + geom_point(mapping = aes(color = class)) + geom_
```

```
## `geom_smooth()` using method = 'loess'
```

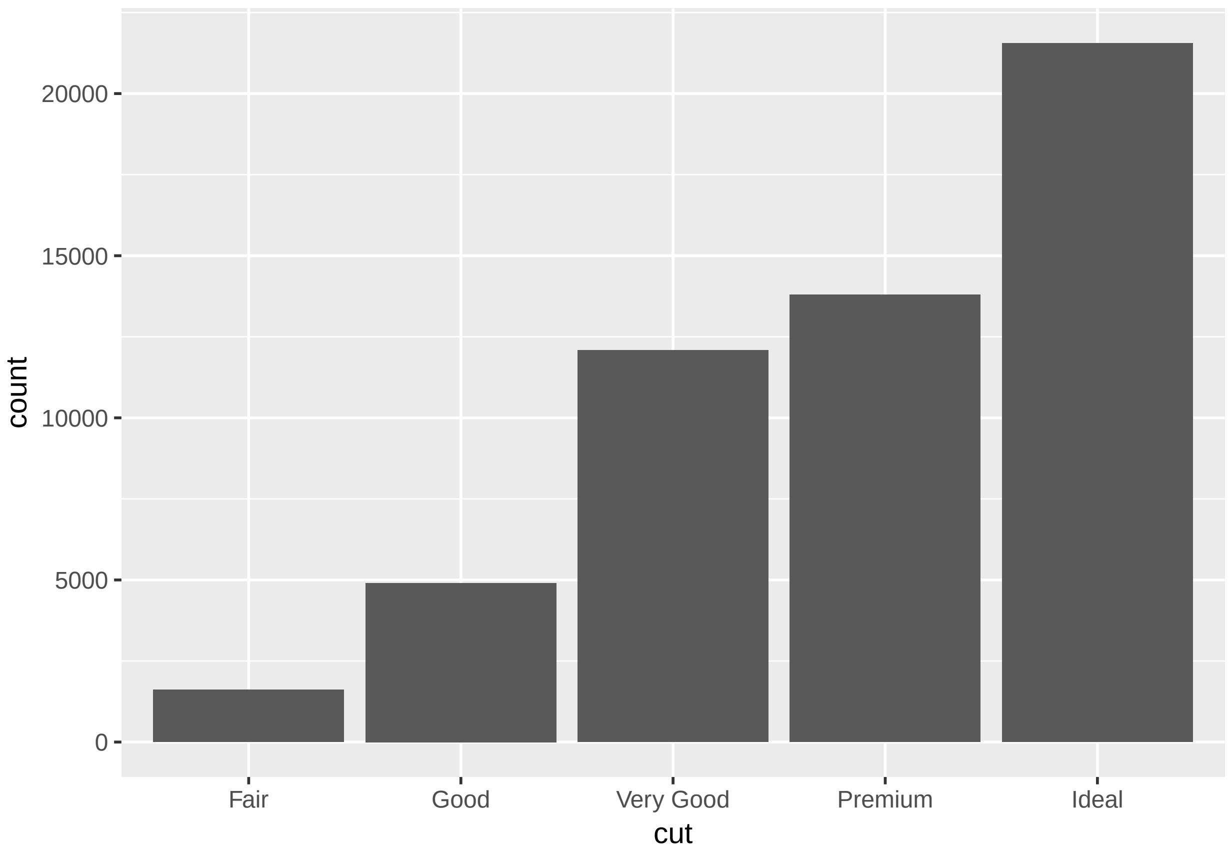


Onwards to explore some bar chart awesomeness. Load diamonds built in dataset and then use a bar graph (using a geom):

```
tail(diamonds)
```

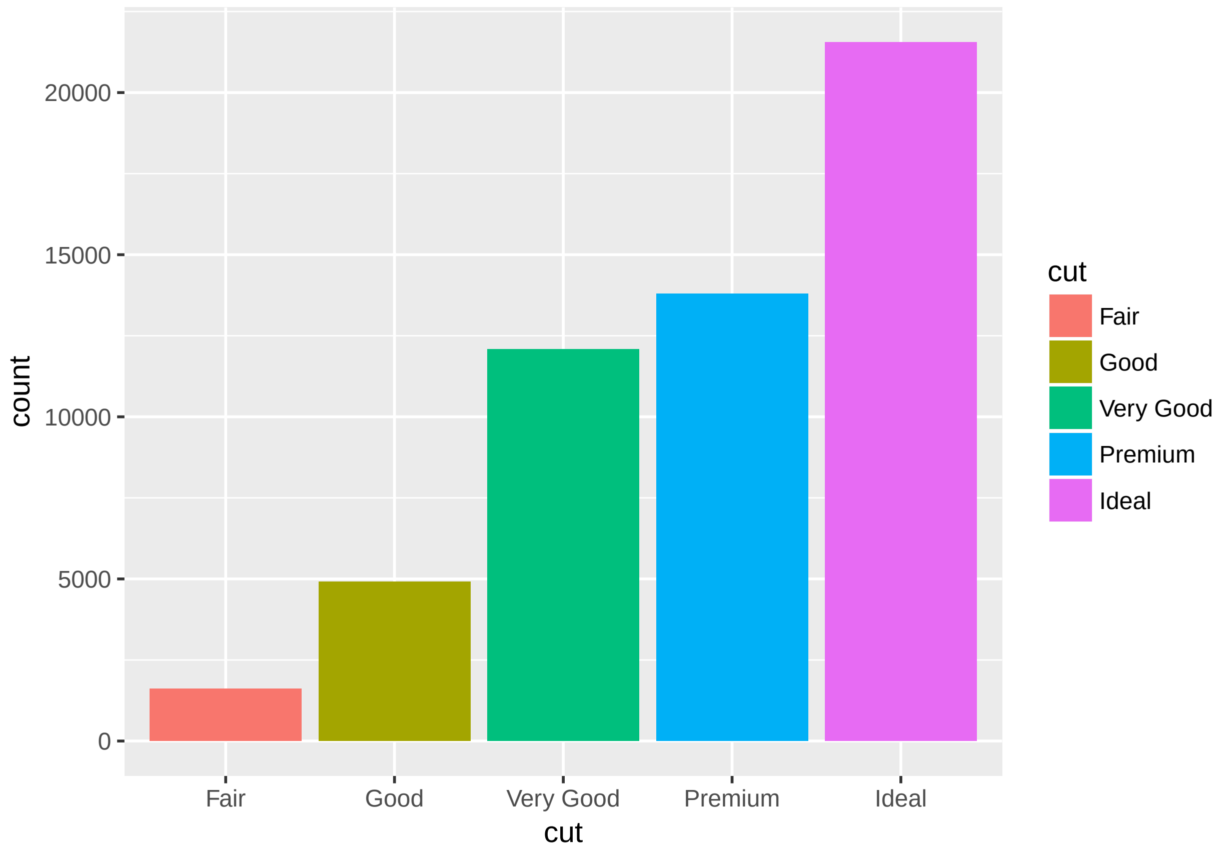
```
## # A tibble: 6 x 10
##   carat       cut color clarity depth table price     x     y     z
##   <dbl>     <ord> <ord>   <ord> <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1  0.72   Premium     D     SI1  62.7    59  2757  5.69  5.73  3.58
## 2  0.72     Ideal     D     SI1  60.8    57  2757  5.75  5.76  3.50
## 3  0.72      Good     D     SI1  63.1    55  2757  5.69  5.75  3.61
## 4  0.70 Very Good     D     SI1  62.8    60  2757  5.66  5.68  3.56
## 5  0.86   Premium     H     SI2  61.0    58  2757  6.15  6.12  3.74
## 6  0.75     Ideal     D     SI2  62.2    55  2757  5.83  5.87  3.64
```

```
ggplot(data = diamonds) + geom_bar(mapping = aes(x = cut))
```

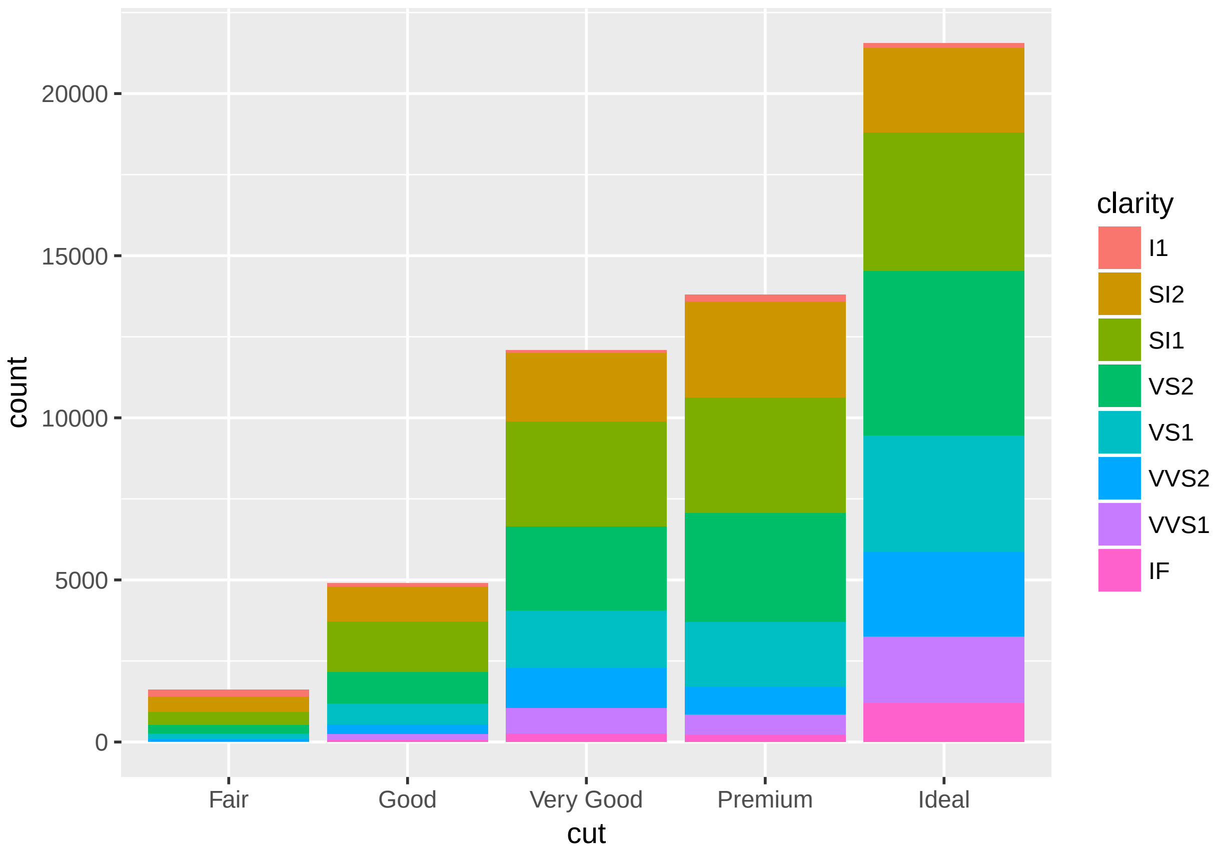


As is the case with scatterplots, you can specify colors with the aesthetic feature. You can also choose to display as a stacked bar plot (by 'filling' with yet another variable from the dataset) or side by side bar plot:

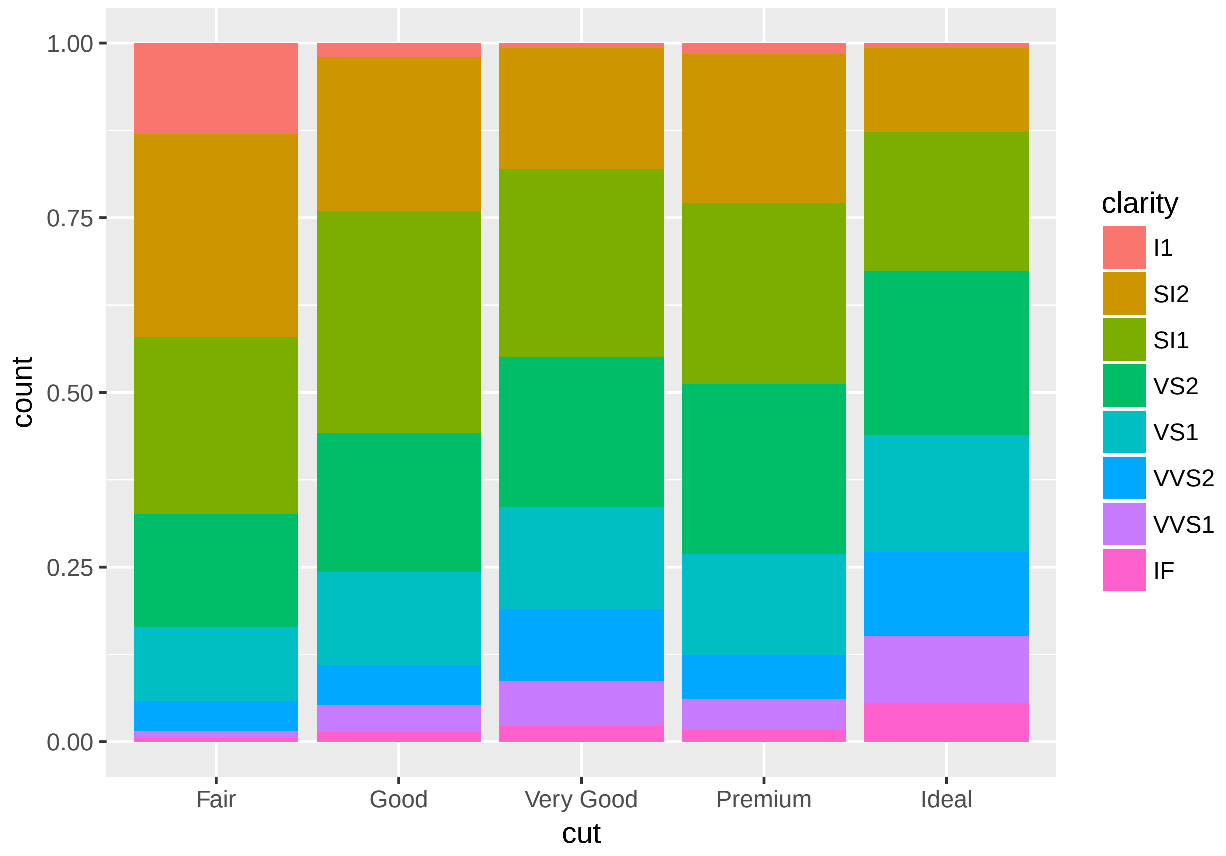
```r
ggplot(data = diamonds) + geom_bar(mapping = aes(x = cut, fill = cut))
```



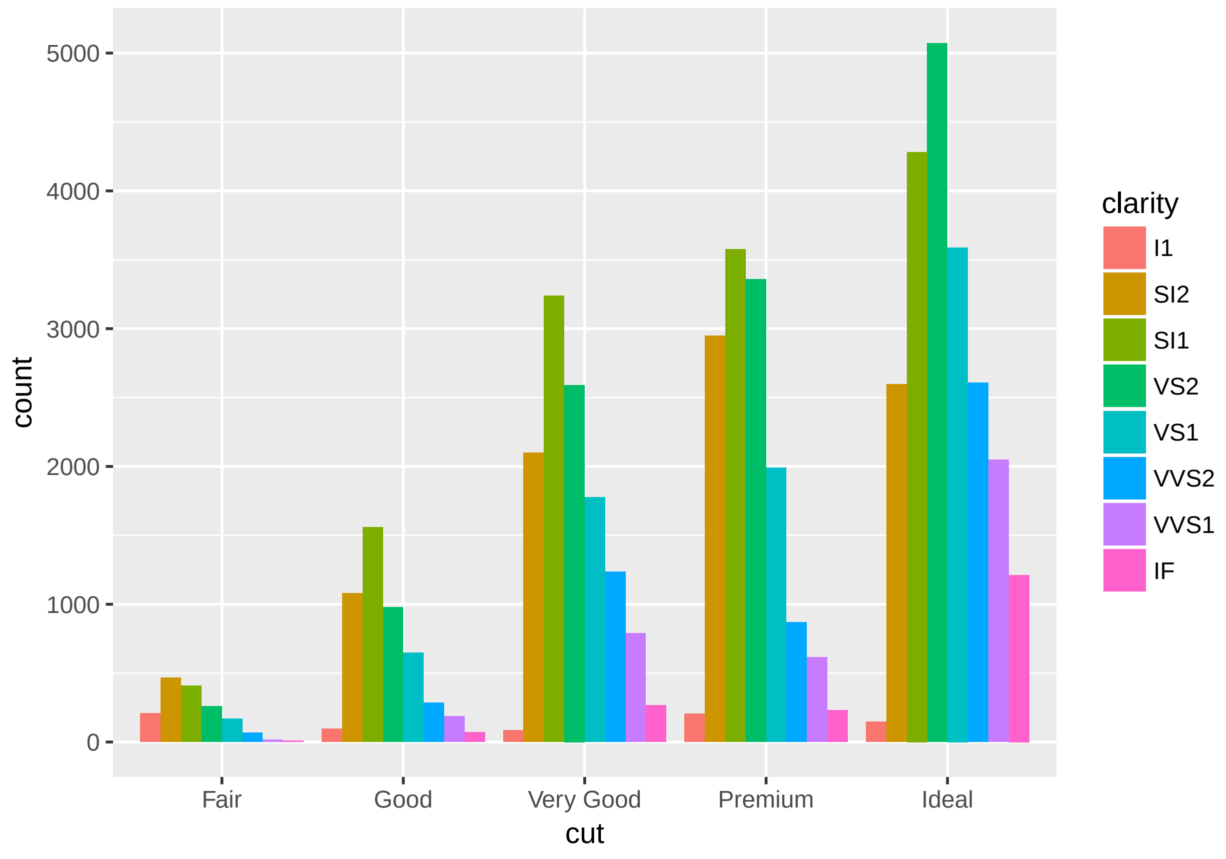
```
ggplot(data = diamonds) + geom_bar(mapping = aes(x = cut, fill = clarity))
```



```
ggplot(data = diamonds) + geom_bar(mapping = aes(x = cut, fill = clarity), position = "fill")
```



```
ggplot(data = diamonds) + geom_bar(mapping = aes(x = cut, fill = clarity), position = "dodge")
```



clarity
I1
SI2
SI1
VS2
VS1
VVS2
VVS1
IF
count
5000
4000
3000
2000
1000
0
Fair
Good
Very Good
Premium
Ideal
cut

13.3 A couple of Probability examples

13.3.1 Goals

- Practice with probability calculations
- Investigate some interesting (and famous!) probability problems

13.3.2 Activities:

1. **Let's make a deal!** An old TV game show called "Let's Make a Deal" featured a host, Monty Hall (a Canadian!), who would offer various deals to members of the audience. In one such game, he presented a player with three doors. Behind one door was a fabulous prize (say, a giant pile of money), but behind the other two were relatively worthless gag gifts (say, donkeys). The player was invited to choose one of the doors. Afterwards, the host would open one of the other two doors to reveal that it contained a donkey. Then, the player is offered a choice: they can either stay with their original door choice, or they can switch to the other closed door. An animation of this problem can be found by clicking **here**. The question becomes, which strategy is better – to stick with the original or to switch doors? The answer is surprising and illustrative of two major themes of statistics:

   (a) Immediate intuition is not always correct; this is why it is always necessary to reformulate a question clearly and to logically distill the steps (in fact, statistical thinking really is just that!)

   (b) Bayesian statistics, in general, is about including "extra" information and updating probabilities. Most of the disagreement within the Bayesian community centers on what prior probability to use to describe and weight the importance of the "extra" information.

   An example

   Draw out a probability tree to answer the following questions (no cheating yourself! Answer a and b before moving on to c) and hand this in with Problem Set 3.

   (a) Find the probability of finding the door with the pile of money, if the player follows the strategy of staying with their original choice.

   (b) Find the probability of success with the strategy of switching to the other closed door.

   (c) Using the applet found **here** play the game 20-30 times using each of the two strategies. Was your answer correct? Remember that because the game in the simulation is a random process, the proportion of successes will not necessarily exactly match your theoretical predictions.

There is a Python solution to this problem. The code is thus:

```
import random


def monty():
    #simulates one round of the monty hall game without switching
    cardoor = random.choice([1,2,3])
    guessNum = int(raw_input("Which door would you like to open?  "))
    if guessNum == cardoor:
            print("You've won a car!")
    else:
            print("Even better: you've won a GOAT")

monty()
```

A more sophisticated version of this program that allows the user to switch, just like on the show:

```
import random


def niceMonty():
    #simulates one round of the Monty Hall game with switching
    cardoor = random.choice([1,2,3])
    guessNum = int(raw_input("Which door would you like to choose? "))
    if cardoor == 1:
        if guessNum ==1:
            open = random.choice([2,3])
        elif guessNum ==2:
            open = 3
        elif guessNum ==3:
            open == 2
    elif cardoor == 2:
        if guessNum ==1:
            open = 3
        elif guessNum == 2:
            open = random.choice([1,3])
        elif guessNum == 3:
            open =1
    elif cardoor == 3:
        if guessNum == 1:
            open = 2
        elif guessNum == 2:
            open = 1
        elif guessNum == 3:
            open = random.choice([1,2])
    print("Look, there's nothing behind door ",open)
    response = raw_input("Would you like to switch doors? (y or n)")
    if response == "y":
        guessNum = int(raw_input("Which door would you like? "))
    if guessNum == cardoor:
        print("You've won a car!")
    else:
        print("You've won a goat!")

niceMonty()
```

We can even compare the ratio of winning the car to not winning the car via a simulation in Python:

```
import random
#a simulation that compares the switching and non switching strategy

def noswitch():
    cardoor = random.choice([1,2,3])
    guessNum = random.choice([1,2,3])
    if guessNum == cardoor:
            return 1 #we won a car!
    else:
            return 0 #we won 0 cars

def switch():
    cardoor = random.choice([1, 2, 3])
    guessNum = random.choice([1, 2, 3])
    if cardoor == guessNum:
        #we guessed correctly but switched!
        return 0
    else:
        #we guessed a door X, the car is behind Y so Monty open Z
        return 1  # we won a car
```

```
def marilyn(games):
    print("No switching strategy....")
    cars = 0
    for game in range(0,games):
        cars = cars + noswitch()
    print("You've won a car this many times: ",cars, " out of ",games, " games ")
    print("The win-lose ratio was: ", 1.0*cars/games )
    print("The switching strategy... ")
    cars = 0
    for game in range(0, games):
        cars = cars + switch()
    print("You've won a car this many times: ", cars, " out of ", games, " games ")
    print("The win-lose ratio was: ", 1.0 * cars / games)


marilyn(100)
```

2. **The Birthday Problem How many individuals do you need to have in a class to have over 50% that at least two people will share the same birthday? How many individuals do you need to have greater than 95%?** If you are still confused, a short video explaining this problem can be found **here**. There are a few other simulations on the web to help you figure this out. Here are two that use slightly different approaches to answer the question: **Here** and **Here** . The second link has a section that explains the answer to the problem.

   Answer to the Monty Hall problem

Monty Hall Problem in two explanations:

Explanation 1:

Reality: The prize is behind door A

Stay strategy:

Initial choice	**A**	**B**	**C**	**Result**
A	**Car**			**Stay and Win**
B		**Goat**		**Stay and Lose**
C			**Goat**	**Stay and Lose**

Switch strategy:

Initial choice	**A**	**B**	**C**	**Result**
A	**Car**			**Switch and lose**
B		**Goat**		**Switch and Win**
C			**Goat**	**Switch and Win**

This is because the host has more information than you and so he ALWAYS shows the empty (or Goat) door which leaves only the door with the prize….

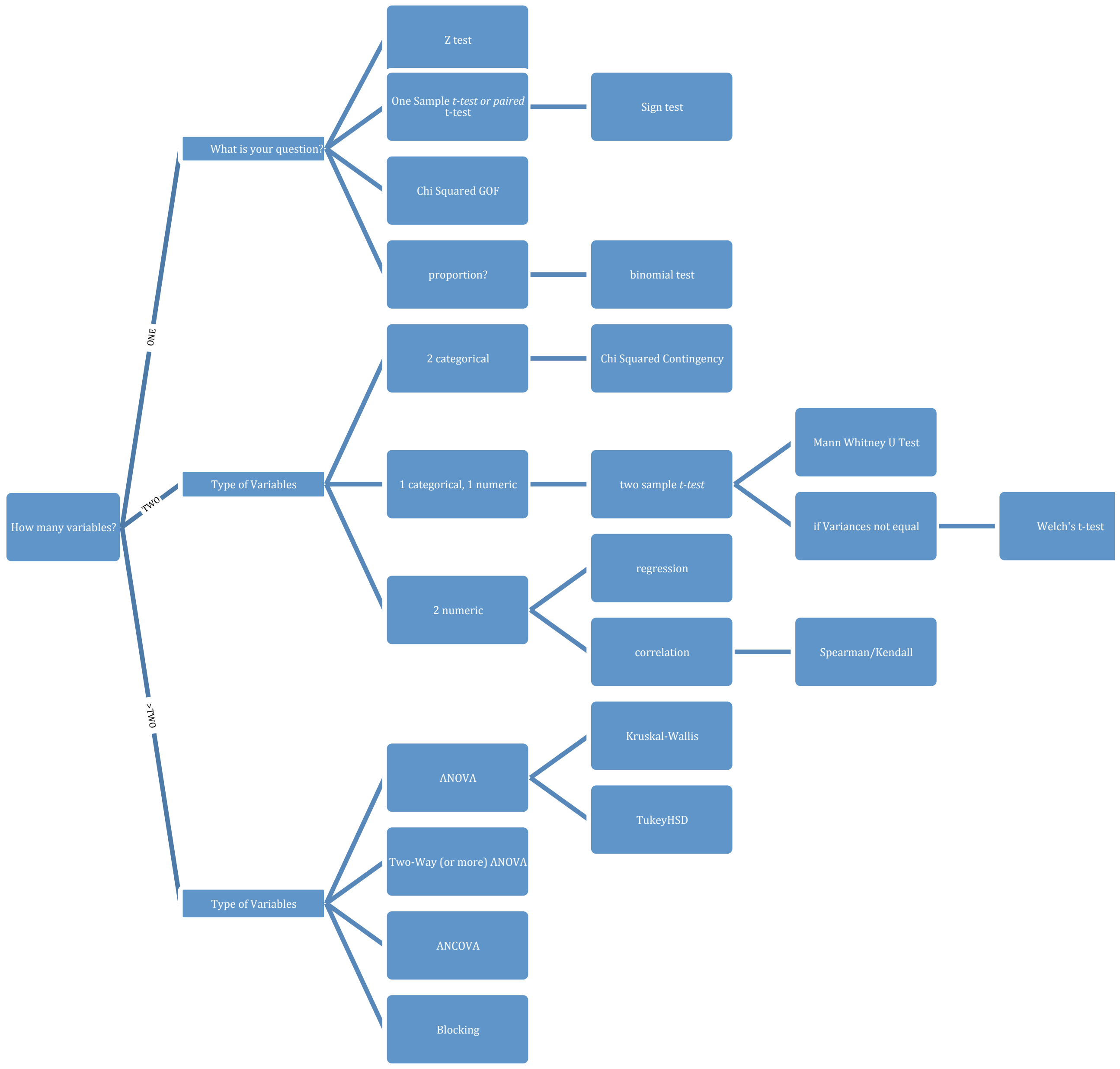
Explanation 2: *(again prize is behind door 1)*

				Strategy
Player picks		Host Opens	Total Probability of Event	Stay: Switch
1/3 **Door 1**	**1/2**	**Door 2**	**1/6**	**Car : Goat**
	1/2	**Door 3**	**1/6**	**Car : Goat**
1/3 **Door 2**		**Door 3**	**1/3**	**Goat : Car**
1/3 **Door 3**		**Door 2**	**1/3**	**Goat : Car**

P(car|Stay) = 1/3
P(car|Switch) = 2/3

http://xkcd.com/

13.4 What statistic?



<u>To test for equal variances:</u>
-Bartlett test
<u>To test for equal normality:</u>
-Shapiro-Wilk test

13.5 Quick reference commands

13.5.1 Quick Reference for R (so far . . .):

As the class proceeds, you will add to this handy list. Notice that many of the session management commands are similar to UNIX commands.

13.5.2 Session Management:

- ls() list the objects in the current environment
- rm() remove objects from the current environment
- q() quit the session of R
- list.files() - lists all files in working directory
- getwd() - current working directory
- setwd("Name of directory") - setting your working directory to name of directory
- args("function name") - list the arguments that a particular function takes
- dir.create() - creates a directory in the current directory
- file.create() - creates a new file
- file.exists() - checks to see if a particular file exists in the current working directory
- file.info()
- file.rename(from,to)
- file.copy(from,to)
- file.path()
- dir.create() - similar to mkdir in Unix
- unlink() - remove directories but you will need to include the argument recursive=TRUE

13.5.3 Getting Help:

- ?function Getting help
- help(function) get help in official help pages
- example(function) run the examples associated with the manual page for the function

13.5.4 Importing/Exporting Data:

- WhateverYouNameYourData¡-read.cvs(file.choose())
- WhateverYouNameYourData ¡-c(. . .) combine or concatenate objects that you enter
- names(dataframe) gives the names of each of the columns in a dataframe
- attach(dataframe) forces R to integrate the name information about each vector so that you can deal with each column in a dataframe separately. Remember to detach(dataframe) at the end of the session so that you dont accidentally cause name conflicts to arise.
- $ - also a way to manipulate only one column of a dataframe. You use this in the following way: dataframename$columnname

<u>Save graph as a pdf:</u>

1. pdf("name_plot.pdf")
2. plot(data) in whichever graph you want
3. dev.off()

13.5.5 Math Functions:

- mean(x)
- var(x)
- sd(x)
- length(x)
- quantile(x)
- median(x)
- min(x)
- max(x)
- range(x)
- a:b - create a vector of numbers from a to b
- seq(a,b,by=interval) - another way to create a vector of numbers from a to b by defined intervals or you can get random numbers between a and b by specifying length argument

- seq.along() - create a sequence that is as long as the object you pass to it
- rep - create a sequence of the same number ie. 0,0,0,0 etc
- paste(vectorname,collapse=" ") - function used on character vectors to specify how to display the individual elements, in this case you want to print out all the words and separate them each by a space

13.5.6 Useful Logical functions

In particular, the **split-apply function-combine** are useful

1. lapply
2. sapply
3. vapply
4. tapply

- isTRUE()
- identical() - compares whether or not two passed objects are the same
- xor() - exclusive OR where one of the two passed objects must be false and one must be true in order to evaluate as true
- sample(n) - samples "n" numbers without replacement
- which() - takes FALSE and TRUE vectors and returns the indices of the items that are TRUE
- any()
- all()
- unique() - returns a vector of the unique values passed to it
- head() - default gives first 6 rows of data set but you can specify more rows
- tail() - last 6 rows
- summary() - gives different information based on class of each variable
- str() - gives class of data, obervation number, number of factors

13.5.7 simulation

- sample() - with or without replacement
- rbinom(num obs., size of each sample, prob) - randomly simulated binomial distribution. All distributions also have dbinom for density, pbinom for probability and qbinom for quantiles
- rnorm()
- rpois()
- replicate()

13.5.8 Some Useful Plotting commands:

- hist(x, breaks) histogram of the frequencies of vector x. The optional “breaks” specifies how bins are constructed. For now, it has a default or you could specify the number of bins
- plot(x) - scatterplot
- plot(x,type=“b”) - lineplot
- boxplot(x∼y)
- barplot(x)

13.6 Data sets

Whitlock and Schluter provided us with most of these data sets

13.6.1 Data set names

13.6.2 Chapter 1

13.6.3 Chapter 2

- **BP.csv**
- **TELOMERES.csv**
- **blowflies.csv**
- **Bat_tongues.csv**
- **question 18 on page 58 of textbook - direct data entry**

13.6.4 Chapter 3

- **caffeine.csv**
- **caffeine_Starbucks.csv**

13.6.5 Chapter 4

- **bumpus.csv**

13.6.6 Chapter 5

There are two datasets given to you in the chapter that you will need to input yourself and there two, given below, that are already part of the **R** package.

- **Titanic**
- **HairEyeColor**

13.6.7 Chapter 6

13.6.8 Chapter 7

13.6.9 Chapter 8

13.6.10 Chapter 9

- **malariaMaize**

13.6.11 Chapter 10

- **17e1LionAges.csv**

13.6.12 Chapter 11

- **bumpus_PCA_groomed.csv**