A (very) brief R tutorial

Mark Peterson

Table of Contents

Chapter 1:
Plotting in R

1.1. Background

When starting out with statistics and data analysis, you may have calculated many statistics by hand, possibly with a calculator or spreadsheet. By-hand calculations are a great starting point to understand basic concepts and acquaint yourself with data analysis. However, as data sets grow larger and statistical questions grow more complex, being able to run your analyses in a computer program will become increasingly important. Here, we are going to explore one of these tools: R, a statistics environment designed to be a fully operational programming language.

This tutorial is going to focus primarily on how to run analyses, rather than what they mean. Throughout, there are questions related to interpretation of the outputs, asking you to state what you think is going on in the data. In these places, please refer to other sources, such as a statistics text, materials from a statistics course, or Wikipedia, which has fantastic descriptions of many statistical concepts and tests. In this chapter, we will work through some simple examples of data manipulation and plotting, slowly building your repertoire and confidence.

1.1.1. Why R?

R was designed with high quality graphics explicitly in mind, which makes the default plots it produces elegant while allowing users full control over the outputs. In addition, this flexibility extends to the handling and display of raw data, making it possible to store, manipulate, and analyze a wide variety of data types in a single program. This flexibility comes at the cost of a large learning curve, especially as many of you are probably new to computer programming. Finally, R is free, both free as in beer (no cost) and free as in speech (the source code is available, can be manipulated, and can run on any platform), making it an accessible choice for students and researchers.

RStudio provides a useful wrapper for R - it allows us to save what we are doing each step of the way, and displays useful information (graphs, help files, etc.) alongside our progress. For all of our activities, we will be working in RStudio.

1.2. Chapter Goals

- How to install and utilize a statistical package
- Be able to import data into R
- Be able to graphically represent your data
- How to comfortably approach command line and computer code problems

1.3. Installing R and RStudio

Below are basic instructions for installing R and RStudio, on each linux, Mac, and Windows. However, you can vist http://cran.r-project.org/bin for more details on the installation of R and http://www.rstudio.com/ide/download/desktop for information on RStudio.

1.3.1. Install in Linux

To install R on a Linux distribution, there are two main approaches. You can either download precompiled binaries for your specific installation, or compile yourself from source code. Binaries (which I recommend for most users) and information on how to install them are available at: http://lib.stat.cmu.edu/R/CRAN/bin/linux/.

RStudio binaries are available for Fedora and Ubuntu/Debian. Copy the current link for your system (examples below) from http://www.rstudio.com/ide/download/desktop, and install with either of the below (for Fedora or Debian respectively):

```
sudo yum install http://download1.rstudio.org/rstudio-0.97.551-x86_64.rpm

sudo apt-get install http://download1.rstudio.org/rstudio-0.97.551-amd64.deb
```

Alternatively, a tarball of the source code is available for both R and RStudio at their respective download pages, if you prefer to install from source.

1.3.2. Install on Mac

In Mac, download the current version of R from http://cran.r-project.org/bin/macosx/ (currently R-3.1.0). Make sure to get the version for your OS version. This can be installed like other .pkg files – double-click and follow the on screen directions.

To install Rstudio on a Mac, download the latest .dmg RStudio file (currently 0.98.507) from http://www.rstudio.com/ide/download/desktop. Install by mounting the .dmg file (double-click on it). An app should appear in a new Finder window. Drag it from the new window into your "Applications" folder. From now on, you can open RStudio directly from your Applications.

1.3.3. Install in Windows

In Windows, download (from http://cran.r-project.org/bin/windows/base/) the current version of R (currently R-3.1.0.exe), and install like other .exe files: double-click and follow the on screen directions.

RStudio for Windows can be installed by downloading the latest .exe file (currently RStudio 0.97.551) from http://www.rstudio.com/ide/download/desktop. Again, double-click and follow the on screen instructions.

1.4. Getting Started

First, open RStudio. Then, select "File → New → R Script" to open a new document. Save this R script file (Either "File → Save" or `Ctrl+s`) in your own directory (folder) with a file name begining with your last name. As we go along, make sure to save frequently - there is no such thing as saving too often. This document will hold all of our commands for the lesson. This file will:

- Provide us with a record of everything we do
- Allow assessment of your progress
- Provide you access to previously used and learned commands
- Allow us to fix small mistakes in long pieces of code without retyping everything
- Give you something to turn in at the end of the lab

Now, lets start by making some notes to ourselves. In R, anything that is preceded by a "#" is ignored. This lets us make comments, which can be incredibly helpful if we ever need to come back to the code, or share it with others. At the end of this chapter, you will each submit your script, along with an additional file we will produce near the end of the lab.

A note on convention: every command I tell you to type will look like the box below. In general, you can copy what you see below directly into your script file; however, things in ALL-CAPS should be replaced with your information. In addition, be careful with copy-paste, as pdf files occasionally have hidden characters. Make sure to run every line you copy over (including comments; both "comments" and "running" are defined soon) as R will throw an error if there is a hidden character, and they are much harder to find if you try to run multiple lines at the same time (as you will at the end of the chapter).

```
## Script written by FIRSTNAME LASTNAME
## Euk Genomics Class First Script
## YEAR-MON-DD
```

This information will help to remind you of what you were doing when writing the code, what its purpose is/was, and when it was written. Believe me, when you come back to this to work on your projects, you *will* appreciate having these comments. Comments can also help others to understand your code, and should become a habit for you. These directions will include comments throughout, but feel free to add your own notes to the code as well. Part of the grade for this assignment, if you are doing this as part of a class, will include your comments and answers to questions in this document.

Now, we need to see how these commands are actually passed to R. In RStudio, if you press `Ctrl+Enter`, it will send the highlighted text (or current line if nothing is highlighted) to the command line. Highlight the three lines of comments (either with the mouse, or by holding shift and using the arrow keys) and press `Ctrl+Enter`. This action sends the comments to the Console and executes any code. Alternatively, you can click "Run" above the script window to accomplish the same thing. Here, the text is commented, so nothing happens.

Next, we need to tell R where it should be looking for your documents, and where it should save outputs. This is similar to opening a folder, but we need to tell R in text. This is called "setting the working directory", and it will be the folder in which R opens and saves any files that you use, until you change the directory. In RStudio, this can be done from a drop-down menu. Click "Session" $\rightarrow$ "Set Working Directory" $\rightarrow$ "To Source File Location." This will set the directory to the folder in which your script file (where you are typing commands) is located. If you want to select a different directory, click instead on "Choose Directory."

You will notice that a command showed up in the terminal window, which should read something similar to:

```
## Manually set working directory
## Make sure that this is set for your computer, not mine
setwd('~/Documents/learningR/')
```

Copy this line into your script, near the very top. This will make it easier to load in your data, which should be saved (and unzipped) inside the same directory (folder) where you save your script. When you set the working directory, always put it near the very top of your script. That way, if you are working on a different computer (or sharing your script) it is easy to see (and change) the working directory to something that works for the local computer.

1.5. Entering Simple Data

The most important thing to be able to do in any stats program is to import your data. In R, there are several methods for doing so, and we will focus on just a few of them. Variables in R are very flexible and can take many different forms. Variable names must begin with a letter, but then can be any length and can include numerals. In R, an arrow, composed of a "less than" symbol and a dash ("<-") is used to assign variables. Try it with this simple example:

```
## Store a simple variable, then display it
testVariable <- 3
testVariable
```

To execute the commands, either highlight all the lines and type `Ctrl+Enter`, or place the cursor on the first line and press `Ctrl+Enter` for each line. The first command creates a variable, and the last line tells the console to display it. This variable can now be used just like a number. Try the code below, typing it in the script, then using `Ctrl+Enter` to execute it:

```
## Use the variable
testVariable + 6
testVariable * 4
```

Now, go back and change the value of the variable by changing the line assigning the variable. Re-execute the assignment and all of the lines using it. Note that this can allow you to re-do an analysis if your initial values change (e.g., because of a typo, or just adding observations) without having to re-type all of your previous commands.

1.6. Entering Longer Data

A single value is not usually very useful for statistics. You will often have multiple observations, and will want to keep them all together, rather than saving each as a different variable. R may have already given you a hint that it can do this. When you displayed the value of `testVariable`, you likely noticed a "[1]" next to the value. (For those with programming experience, note that the index starts at 1, not 0.) This is an "index" telling us that the value displayed is the first element of that variable. That is because, in R, most simple variables are stored as "vectors." This allows us to store many values in a single variable. We need to tell R that we are giving it several values to store together using the command "`c()`". Try it out with this code:

```
## Make a test vector, then display it
testVector <- c(1,3,4,6,8,10,15)
testVector
```

Vectors in R have some special properties, and these properties will become very useful. First, lets see how vectors handle a little bit of arithmetic:

```
## See how vectors respond to arithmetic
testVector + 3
testVector * 2
testVector + testVector
```

What did you notice about the outputs? For arguments that are only one number long, the same procedure is applied to each element (item) of `testVector`. When the arguments are the same length (have the same number of stored values) as `testVector` (e.g., itself), then the argument is applied element-for-element.

1.7. Playing with Data

Now that we have some sense about how these things work, we can begin to look at the data a little more closely. First, lets create a new vector with each element being 3 greater than the elements in `testVector`. Go back to the the first arithmetic line with `testVector`, change it to match the line below, and run it:

```
testVectorB <- testVector + 3
```

All outputs in R can be saved in this fashion, which allows you to go back and use them later or to perform other manipulations on them. For example, what if you only want to know what the 3rd element of `testVectorB` is? R allows us to do that using brackets "[ ]" to tell it which element we want to see. Try it:

```
## Display only the 3rd element of a vector
testVectorB[3]
```

We can also use this to display only elements that meet certain criteria. Let's display only the elements that are greater than 8:

```
## Display only elements greater than 8
testVectorB[testVectorB > 8]

## Show how it is done:
testVectorB > 8
```

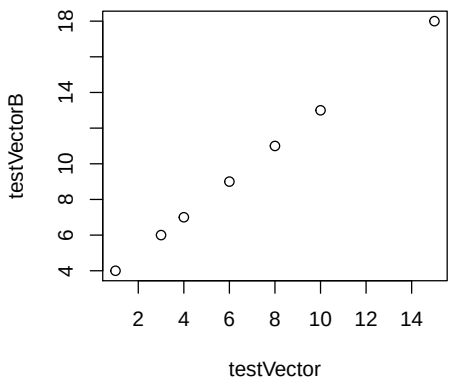
As you can see from the last line, R produces a type of value called a "logical" that says whether or not a condition (here, being greater than 8) is met. When placed within brackets, only those elements that meet the condition are displayed. This is useful for displaying certain data, but is especially important when looking at two different variables. For example:

```
## Display the value of testVectorB when testVector equals 3
## Note that two '=' are used for logical tests
testVectorB[testVector == 3]
```

As we begin to analyze the data you have generated in class, we will have grouping variables (e.g., categories, such as a treatment) that can be used to split the data just like this.

Finally, we are going to generate a plot using these two vectors. In R, functions (such as `c()` and `plot()`) accept arguements inside a pair of parentheses. Once you open a parenthesis with a "(", R will keep expecting that what you have provided is a part of the function, until you close it with a ")". Functions accept arguments either in the order they are expected, as we use below (see the help for each function to see the order), or by naming each argument followed by an equal sign and the value (e.g. "`x = 1`"). The general format for plotting is to give the x-coordinates, followed by the y-coordinates, as follows:

```
## Make a plot of two vectors
plot(testVector, testVectorB)
```



There are several additional options that we can use to change the plot, and we will explore those as we move forward with analyzing data.

1.8. Importing spreadsheet style data

Typing in data, as described above, works for simple datasets. What about when you have 100 observations, each with multiple measurements and properties? You could go through and type them all as individual vectors, but that would get cumbersome rapidly, and leads to typos and other errors affecting your results. Instead, we can use a type of variable called a data frame (data.frame in R) that will keep each vector together, and reduce errors. Try this:

```
## Make a data frame out of the two testVector's, then display it
testDataFrame <- data.frame(testVector,testVectorB)
testDataFrame
```

As you can see, the rows now show which elements of each vector correspond with each other. In this way, you can see the relationship between the two vectors, and, if one were a grouping factor (e.g., treatment), use that relationship in your analyses. However, this still involved typing in all of your data, which is still a pain and error-prone. For any large dataset, you probably already have a spreadsheet, and would like to be able to just use that. A safe, universally compatible approach, is to use a text based format for saving your data, such as comma-separated values (csv).

For today, the data are available from the zipped "data" directory associtated with this tutorial. Make sure to save the zip file in the same directory as your script, and then decompress it before continuing. We will start by looking at the fastPlant data, which gives the height of plants following growth in either high or low fertilizer and high or low light exposure.

To import data, click on "Tools" then "Import Dataset" and "From text File." Select the file "fastPlant.csv" and follow the on-screen prompts. Make sure to change the separator parameters if needed (e.g., to a comma), and tell R that the dataset has a header row. Use the preview of the data frame (in the bottom right of the window) to make sure that the dataset is reading in correctly. If necessary, name the resulting data.frame "fastPlant" in the top left. Click import, then copy the resulting command into your script. It should look something like:

```
## How you would load a data set from the command line
## Note that this can be relative to your working directory.
## Your path may be different, and should be copied
##   from the Console after going through the menus.
fastPlant <- read.csv("data/fastPlant.csv")
```

Note that, as long as your data directory is in your working directory (set above), you can include just the "data/fastPlant.csv" portion of the path, as R will know to start in your working directory. This is known as a "relative" path, as it is relative to your working directory (as opposed to an "absolute" path that includes the full location), and is generally a best practice, for several reasons. First, it means that if you move your data directory (for example, if you move this directory into a directory for past courses), all you need to change to be able to run this script is the `setwd` line at the top. Second, and more importantly, using relative directories makes it easier to share scripts with classmates and collaborators. You'll likely notice your name in the absolute path, and the people you share your script with will have a different name there (or maybe a completely different path, espcially if they use a different OS). By using relative directories, only the `setwd` line at the very top of the script needs to be changed, not every line reading in data.

There are several ways to access the vectors in this data.frame, and we will explore a couple. First, you can use the brackets (" [" and "] ") just like for vectors. Here the format is "[row, column]." An empty argument means "include all" as you can see below:

```
## Display the third column, then the 4th row
fastPlant[,3]
fastPlant[4,]

## Display just the 4th element of the 3rd column
fastPlant[4,3]
```

This format works for several data types in R, but data.frames allow the use of column names, which makes the code much easier to follow. To call a single column by name, we can either use brackets or the "$" symbol. Either approach produces a vector, which can then be manipulated in the same manner as the vectors we used earlier. Remember that "tab" can autocomplete a partially typed variable (or show available options); this trick also works for column names after typing a "$".

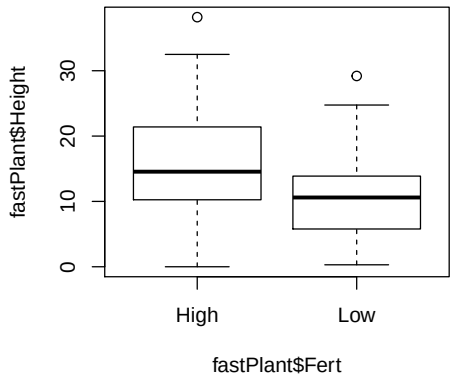
```
## Display a column by name
fastPlant$Height
fastPlant[,'Height']

## Display the heights of only fertilized plants
fastPlant$Height[fastPlant$Fert == 'High']
```

This is a great way to work with multiple datasets, and keeps things very clean.

1.9. Plotting data using fast plant results

One of R's most powerful features is its plotting ability and options. The default options are a great place to start, and they can be modified readily to produce exactly the plots that you envision. To start, let's plot the height of each plant, split by whether or not it was fertilized. To do this we will use the operator "~" (a tilde, usually above the tab key) to denote that we are entering a formula. That is we are telling R to consider what we input as "Response Variable ~ Predicting Variable(s)." Try the below:

```
## Plot the heights of plants based on fertilization
plot(fastPlant$Height ~ fastPlant$Fert)
```

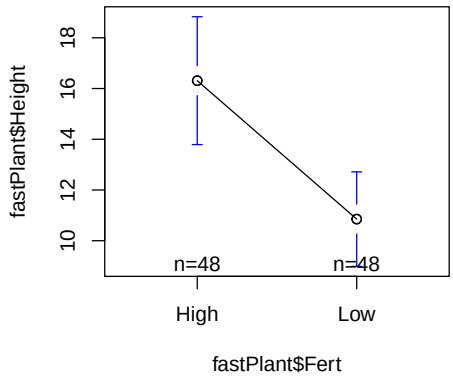


Now, look at the graph that was produced. It looks rather different than our earlier plot. For categorical data (called factors), R's default is to produce box plots. What if you want to plot the means and confidence interval instead? You could go through and compute the values yourself and plot those, but here we encounter another of R's powers: user created packages. These packages cover a huge range of things, and exist for nearly anything you might like to do. Here, we would like to use the function "`plotmeans`" from the package "`gplots`" so we need to install the additional package. Copy and run the following:

```
## Install a new package, make sure to follow on screen prompts.
install.packages('gplots')

## Load the package for use:
library(gplots)
```

Packages only need to be installed once on each computer, so you can now add a "#" before the installation line, to prevent it from running again (this is called "commenting out" a line). In the future, you will only have to run the `library` line to load the package. Now, use one of the packages functions to make a new plot:

```
## Use the plotmeans function from the new package
plotmeans(fastPlant$Height ~ fastPlant$Fert)
```

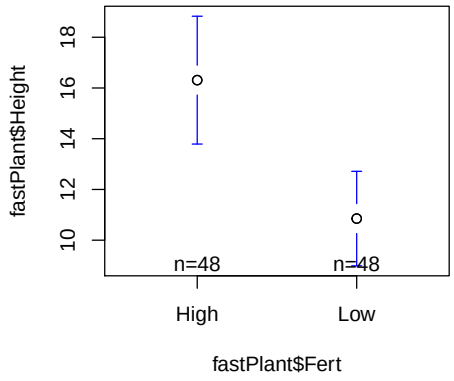


This plots the mean for each group, along with the confidence intervals. What if we don't want the line connecting the two groups? First, we can look at the help page for `plotmeans`, which will tell us what arguments we can use (and their default values).

```
## Open help for plotmeans
?plotmeans
```

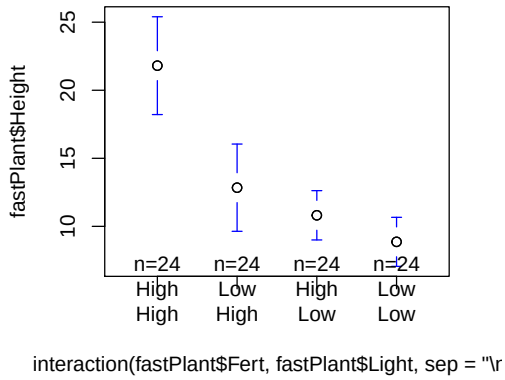
Then, we can change an argument to avoid adding those lines.

```
## Re run with the parameter changed
plotmeans(fastPlant$Height ~ fastPlant$Fert, connect=FALSE)
```



This output looks a lot better and suggests something about the effect of fertilizer on growth. Add a comment ("`## comment`") that says what this figure suggests. However, there is still more that we can learn from plots like this. We actually have 4 treatment groups, and we can display them all at once, even though there is not currently a variable naming them explicitly. We could add such a variable, but there is actually a simpler workaround: the function "`interaction()`". Try the below code to plot the full experimental design:

```
## Plot the full experiment
plotmeans(fastPlant$Height ~ interaction(fastPlant$Fert,fastPlant$Light, sep='\n'),
          connect=FALSE)
```



This plots the four groups (a 2x2 interaction) separately, again with mean and confidence interval bars. What does this plot suggest about the treatment? Add a comment with your answer. A few notes are useful here. First, notice that we are able to use a function within a function to provide us with a different input - this is an important feature of R that can really help keep code readable. Second, note the "sep= '\n' " in the function. This tells the program to separate the two factors (`Fert` and `Light`) with a newline. You can use any separator you would like, but the newline tends to work best for plotting like this. Finally, the plot is a bit messy still. You can control a number of parameters to clean this up. Open the help, then add a couple parameters to improve the plot. I will leave the specifics up to you:

```
## Open the help, then make the plot prettier
?plotmeans
plotmeans(fastPlant$Height ~ interaction(fastPlant$Fert,fastPlant$Light,sep='\n'),
          connect=FALSE, main= 'TITLE FOR YOUR PLOT',
          xlab= 'Treatment Groups (Fert; Light)', ylab= 'Height (UNITS)')
```

Note: in this document, continued lines will be indented as above. In RStudio, the line can continue to the right effectively indefinitely. If you would like to, you can add line breaks manually, and RStudio should indent appropriately. If you use multiple lines, remember that you need to run the full command, not just individual lines. Feel free to modify any other parameters that you would like as well. Just make sure that the plot looks nice when it is done.

Before we move to the next dataset, lets save this plot. There are two general approaches to doing this. First, in the plot window, click "Export" then "Save Plot as Image ...", name the file with your last name, and save it wherever you would likely (probably in your working directory). Alternatively, you can use the function `png()` or any of the similar functions to directly create an image file. Look at the help for `png` to see which file types are available, and what options you can set. For a simple plot, the only required arguement is a file name, though you can do things like settng the background color, changing the size of the output file, and setting resolution. (If you prefer pdf files, there is also a `pdf()` function.) However, you must always remember to close the connection to the file (known as a "device") to stop adding things to that plot, using `dev.off()`. Try it here:

```
## Save a plot directly
## This will automatically save the plot if you run it again
png('fileName.png')
plotmeans(fastPlant$Height ~ interaction(fastPlant$Fert,fastPlant$Light,sep='\n'),
          connect=FALSE, main= 'TITLE FOR YOUR PLOT',
          xlab= 'Treatment Groups (Fert; Light)', ylab= 'Height (UNITS)')
## Always turn off the device, so R knows you are done
dev.off()
```

1.10. Looking at normality using penny ages

Now, we are going to switch to a new data set, and play with a couple of other functions in R. This data set has the age of 1000 pennies, measured in years prior to the day they were collected by students. For the sake of being able to find things, I like to separate out the sections of a script with something like:

```
#####################################################################
#################### PLAY WITH PENNY DATA ###########################
#####################################################################
```

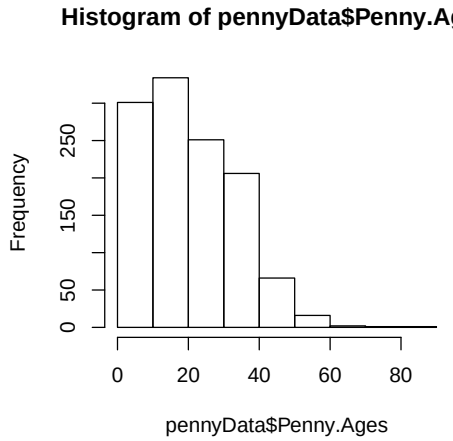
This way, it is easy to search back through your code to find a particular section. You could, alternatively, create a new script. However, to keep the lab all together, let's not. Now, we are going to read in the penny data, just like we did with the fast plant data. Either, go to "Tools" then "Import Dataset" and "From text File" or try typing it directly, as listed below. Note that in RStudio, hitting 'tab' will auto-complete many commands, variable name, and even the path to files. If you use the menus, make sure to change parameters (including setting the "heading" to "yes"), and name the resulting variable, this time "pennyData" Don't forget to copy the resulting command into your script. From here, we can begin using the data.

```
## Load pennyData
pennyData <- read.csv("data/pennyData.csv")

## Display some of the data
pennyData$Penny.Ages
```

As you can see, there are 1000 penny ages in the data set, but it is difficult to get a sense for what the data look like from 1000 numbers, particularly if they weren't ordered. A histogram is a great tool to display the shape of a dataset. Here, we are going to search for the function within RStudio, but you could easily type "make histogram in R" into Google for some more great examples. Search directly in R by using "`??histogram`" at the command line.

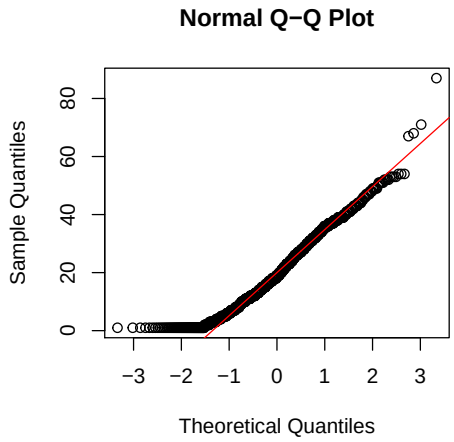
Look through the resulting list, to see if anything looks promising. It seems that the function "`plot.histogram`" might be useful. Click on that and look at the resulting help document. This doesn't look quite right, but does point us to a function called "`hist`." Click on one of the links to "`hist`" and browse through that help file. That seems more like it. Now, let's try using `hist()`:

```
## Plot a histogram
hist(pennyData$Penny.Ages)
```



Does the data look normally distributed to your eye? Add a comment to your script describing the graph and what it suggests about the distribution. Now, let's try a plot that will give us a better idea about the normality: the `qqnorm()` function creates a plot with the observed data (y-axis, by default) against the expected values from a perfect normal distribution.

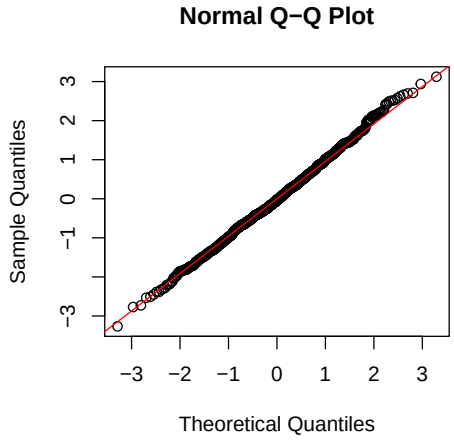
```
## Plot the qq-plot for penny ages
qqnorm(pennyData$Penny.Ages)

# Add a line to observe deviation
qqline(pennyData$Penny.Ages, col='red')
```



Does the data align very closely to the line? Add a comment suggesting whether or not you think the data are normal based on this plot. Note here that the argument "`col=`" lets you set the color of the output. This argument works for many plot functions, and there are 657 named colors (type "`colors()`" for a list), or can be specified with hexadecimal notation (e.g '#FF0000').

For comparison, we can generate a random data set from the normal distribution, and see what the plot looks like. Try this:

```
## Generate random data to see what a QQ-plot 'should' look like
## generates 1000 random data points from the normal distribution
randData <- rnorm(1000)

qqnorm(randData)
qqline(randData,col='red')
```



Use that graph as a baseline against which to assess your data. You can use the "Back" button (left facing blue arrow) above the plots to view previously created plots. Update your comment as needed, given your new information. In many cases, this eye-ball approach is a great first approximation of the normality of your data. However, R also has several built in normality tests, which we will explore in the next chapter.

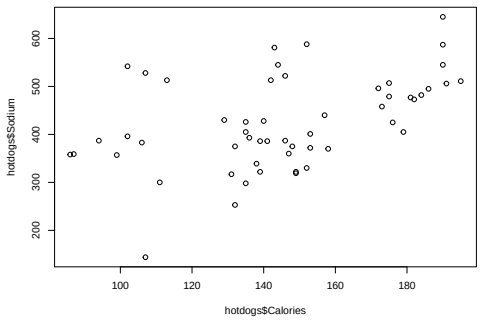
1.11. Plotting correlations using hot dog data

We are now switching to another data set again, this time a data set on the calories and sodium in various types of hotdogs, originally collected by Consumer Reports in June 1986 (pp.

366-367). These data are in the file "hotdogs.csv". As before, put in a new separator (e.g. lots of "#"s) to make it easy to find this section if needed. Load the data as we have above (look back if you need to), and name the data.frame "hotdogs". Don't forget to copy the command into your script if you don't type it directly.

Next, lets play with a few more plotting options. This time, we want to look at the relationship between sodium and calories. So, try:

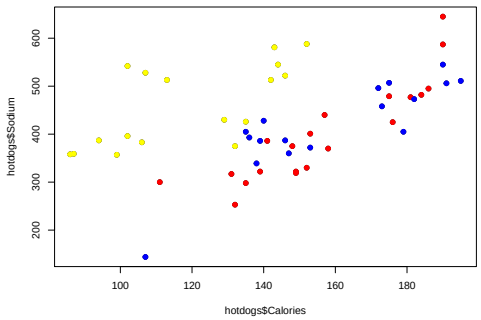
```
## Plot relationship between Calories and Sodium
plot(hotdogs$Calories, hotdogs$Sodium)
```



This plot works, but it is missing a title and information about units. Use the plotting parameters from above (e.g., `main, xlab, ylab`, etc) to modify the plot to make it clearer. Does it appear that sodium is related to calories? Add a comment to your script.

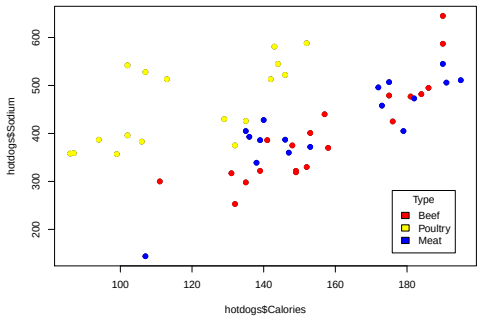
What if, however, only some types of hot dogs show such a relationship? We can color code the types of hot dogs to look at them separately. The simplest way to do this is with the function `points()`, which adds points to a graph. Recall from earlier that "[ ]" calls specific elements from a vector, and can be used with a logical test to limit which points are displayed. Here, we will also use a different shape for the points that will be easier to see. Look at the help for `par` (`?par`) for more information about all graphing parameters, including `pch`.

```
## Add color coded points for each type of hot dog
## Feel free to use different colors, or use '?par' to look at other 'pch' options
## the '==' tests to see if the values are identical
plot(hotdogs$Calories, hotdogs$Sodium)

points(hotdogs$Calories[hotdogs$Type=='Beef'],
       hotdogs$Sodium[hotdogs$Type=='Beef'], pch=19,col='red')

points(hotdogs$Calories[hotdogs$Type=='Poultry'],
       hotdogs$Sodium[hotdogs$Type=='Poultry'], pch=19,col='yellow')

points(hotdogs$Calories[hotdogs$Type=='Meat'],
       hotdogs$Sodium[hotdogs$Type=='Meat'], pch=19,col='blue')
```



Now, each of the types is color-coded, and we can interpret each. Has your conclusion changed at all? Add a comment to your script here. Finally, we can add a legend to the plot with the function `legend()`. Not that a legend can either be added by naming a position (e.g., top), or by passing explicit x-y coordinates. Here, we will use a named position, but for finer control, the coordinates may work better for you.

```
## Add a legend
legend("bottomright", inset=.05,
       legend=c('Beef','Poultry','Meat'),
       fill=c('red', 'yellow','blue'),title='Type')
```



1.12. Bar plots and tables using violations data

Finally, we will work on a new type of plot in R: the bar plot. For this, we will use the violations data set. These data are a sample of the fines assessed to college students on a large college campus, including the Year of the student (e.g., Freshman or Senior), and the amount of the fine. The data are in the file "violations.csv"

Load it as before. Don't forget to copy the commands into your script, if you use the file menu. The year data is of the class (aka type) "factor." This is a variable type that R uses to recognize that the data are grouping variables – categories that separate data. The values of these groups are by default stored alphabetically as the "levels" of the variable. The first thing we want to do is change that order to something more useful for our purposes:

```
## See what kind of variable 'Year' is
class(violations$Year)

## View the levels of the variable 'Year'
levels(violations$Year)
```

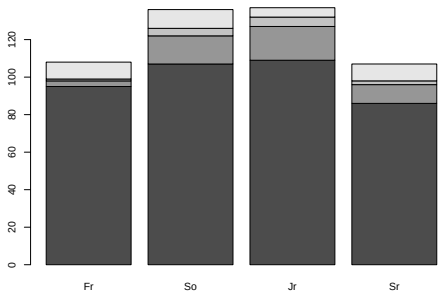
```
## Set the levels of Year to the normal order
violations$Year <- factor(violations$Year,c('Fr','So','Jr','Sr'))
```

Now, lets look at a helpful summary of the data. For this, we will use the function `table()` in R. `table()` counts the number of times a given value occurs, and produces and output that shows those counts.

```
## Produce a simple table, here showing the number from each class
table(violations$Year)

## Save and display a more useful table
violationsTable <- table(violations$Fine,violations$Year)
violationsTable
```

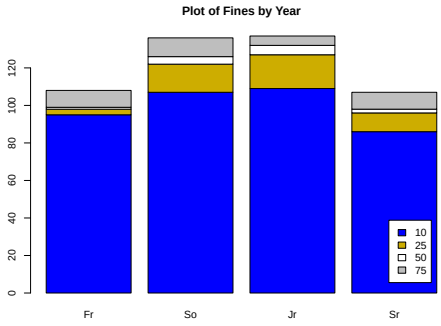
This table shows the number of fines of each value given to each class in the dataset. Such a table can then be used for a large number of other R functions, including plotting using barplot(). A simple example is just to call:

```
## Make a simple bar plot of vioations
barplot(violationsTable)
```



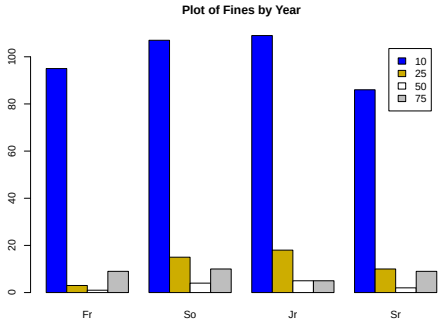
This simple plot can give us a good first look at the data, but we can make it more beautiful (one of the powers of R).

```
## Set some colors to use, Juniata colors seemed fitting
barColors <- c('blue','gold3','white','grey')

## Use the bar colors, and add a title to the graph
barplot(violationsTable, col=barColors, main='Plot of Fines by Year')

## Add a legend
legend("bottomright",inset=0.05,
       legend=rownames(violationsTable),fill=barColors,bg="white")
```



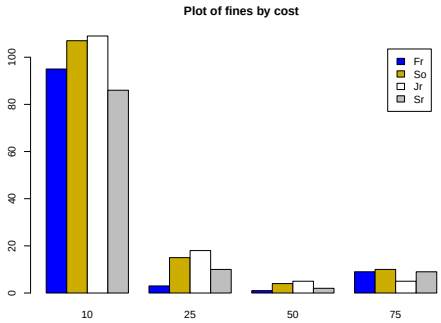
In addition, the `barplot()` function allows us to show the data next to each other, rather than stacked. This allows each of the fine values to be compared more easily, and may be more useful, depending on your purposes.

```
## Create a plot with the values side-by-side
barplot(violationsTable, beside=TRUE, col=barColors, main='Plot of Fines by Year')

## Add a legend
legend("topright",inset=0.05,
       legend=rownames(violationsTable),fill=barColors,bg="white")
```



Finally, tables can be rotated to allow us to look at the data in a different way. This uses the function `t()`, which transposes any table or matrix.

```
## Display the rotated table
t(violationsTable)

## Make a plot of the transposed table
barplot(t(violationsTable),col=barColors,
        beside=TRUE,main='Plot of fines by cost')

## Add a legend
legend("topright",inset=0.05,
       legend=rownames(violationsTable),fill=barColors,bg="white")
```



As you have hopefully seen, R has an incredible amount of power. In this chapter, we have focused largely on importing data and displaying it. In the future, use these notes, and the script you generated as a starting point for your analyses. Google is your friend (GIYF) as you move forward and add additional approaches and statistical tests.

1.13. Assignment

For your assignment, create an html output from the script to confirm that your code worked. Even if you are not doing this tutorial for a class, generating this file will allow you to check your own work, and will catch most common mistakes. RStudio has a built in tool to generate these files, but it requires an R package called knitr. Install the knitr package from the command line, not in your script using:

```
install.packages('knitr')
```

. . . and follow the on screen prompts.

Now, click on the small notebook icon just above your script. It should say "Compile an HTML notebook from the current R script" when you mouse over it. Assign an appropriate title, and put your name as the author. Click "Compile" and an html file will be created. Submit these as instructed for the course

Chapter 2:
Statistical Tests in R

2.1. Background

Hypothesis testing lies at the heart of statistics. Whether it is determining whether or not two samples differ, whether or not a mean is different than zero, or testing the effect of many variables simultaneously, statistical tests help us to understand the world around us. All of the tests you will use in class were developed by hand and, for much of their time, have been calculated by hand. Today we will be learning how to use a computer program to compute the statistics for us.

2.1.1. Functions and test in R

We will again be using the R statistical environment and RStudio to conduct these analyses. The community based nature of R ensures that all of the common statistical tests are available to us and that there is extensive help and documentation available online. For our statistical tests today, we will be calling built in functions in R, using the syntax: "`functionName(argument1=value, argument2=value, \ldots)`" to generate a result. The help "`?functionName`" is great for these tests, but is sometimes not clear on how a function should be interpreted. So, you may want to supplement this help (and this tutorial) with other resources, such as a statistics text and/or Wikipedia.

2.1.2. Updating an old script

Today, you will be relying (rather heavily) on the script that you generated in the last chapter. Refer to your notes (comments) in the script to help you, and remember how grateful you are for those as you comment today. Focus on writing your scripts well the first time, and you will save yourself massive amounts of time in the future.

2.1.3. Reminders

Below are a few notes to remember from the last chapter. For more details, refer back to the previous manual or to your notes in the old script.

- Comment and save frequently.
- Highlighted commands, or the current line, are sent from your script to the console with either `Ctrl+Enter`, or by clicking "run."
- Run every line of your script, including comments, as you add them. This makes catching errors much easier, since you will know exactly which line contains the error.
- Always set your working directory near the top of your script.
- Reading in data works relative to your working directory, which may help you identify errors.

2.2. Chapter Goals

- How to run basic statistical tests
- How to utilize previous scripts to save time
- The basic syntax needed for lots of other tests

2.3. Getting Started

2.3.1. What you need for today

Today, you will need:

- Your old script
- Your computer with R and RStudio installed (see previous chapter for details)
- The "data" folder from the last chapter

Make sure that you have all of these available before we start. Refer back to your old code and lab manual frequntly for more information on topics we covered before.

2.3.2. Setting up your script(s)

First, open RStudio and open your previous script from "File → Open File . . . " Then, select "File → New → R Script" to open a new document. This document will hold all of our commands for today's lesson, so save it with a meaningful name, in the same directory as your old script. Make sure that this name includes your last name along with something to distinguish it from the previous script. As we did last time, you will be turning in the html output we create from the script at the end of the lab.

Now, lets add some information to the top of the script. From your old script, copy over the header information; modify it with today's date and description. Remember that, in R, anything that is preceded by a "#" is ignored by the console, which allows us to add very useful comments to ourselves. It should look something like this:

```
## Script written by FIRSTNAME LASTNAME
## R lab 2- learning tests
## 2014 Jan 27
```

These comments will help you when, like today, you go back to old code for help. Make sure to add comments throughout your script today, particularly to answer questions posed in the chapter. If you are using this tutorial for class, you will be graded on these answers, so make sure to be clear in the script.

Next, set your working directory. Go to your old script and copy the "`setwd()`" command, it should like something like:

```
## Manually set working directory
## Make sure that this is set for your computer, not mine
setwd('~/Documents/learningR/')
```

2.4. The t-test

One of the most common statistical test is the "t-test." By now, you have likely calculated t-tests by hand at some point. For larger data sets, however, you may find such hand calculations cumbersome. Luckily, R has a built in function to calculate t-tests for us. Take a look at the help for the t-test in R using "`?t.test`" and see what options are available.

To see what the output of a t-test looks like, let's try running it with a set of random data. This approach allows us to control a number of factors, including the difference between samples and the standard deviation. Use the following code to generate two random variables, then run a t-test to compare them. If you are unsure of the options in "`rnorm()`" use the help, or hit tab to see what is available.

```
## Generate two random samples with different means
randA <- rnorm(n=20,mean=10,sd=5)
randB <- rnorm(n=20,mean=15,sd=5)

## Run a test to compare them
t.test(randA,randB)
```

This output shows a lot of information, each bit of which may be useful for different applications. The line beginning with "`t=`" gives us all of the test statistics, including the p-value. Here, we can see that the two samples are indeed different, as we expected them to be. Now, go back and change the mean of randB to 10 and rerun the t-test. Note the change in p-value, your result is likely not significant any more. If it is, compare with your neighbors and consider why about 1 of the 20 students in the class may get a significant result.

2.4.1. Power and the t-test

Now, let's try running the t-test with some of the data we used in the last chapter, starting with the `fastPlant` data. Go to your previous script and copy over the lines to read in the `fastPlant` data. Then, add the code for the final plot (using `plotmeans`) to give you a visual guide to these data. Don't forget that you will need to load the `gplots` library for this plot to display. (It is generally best practice to load all required libraries at the top of a script, so that it is easy to make sure that they are all installed if you share the script with someone else.)

Because there are two grouping variables, we will focus on just one subset: the plants in low-light. Later in this lesson, we will explore how to analyze more of this data at once. The t.test function provides a simple way to limit the data analyzed with the "`subset=`" argument. The subset option works with the "`formula=`" approach to the `t.test`, which is similar to the formulas we have used in the past. Note that we can either include "`fastPlant$`" before each variable, or include "`data = fastPlant`" in the function call. Many, but not all, functions include this option, and it can be a great time saver and make the outputs a little cleaner. The code to run this test is either:

```
## Run a t test on a portion of the fastPlant data
t.test(fastPlant$Height ~ fastPlant$Fert,subset= (fastPlant$Light=="Low") )
## Or
t.test(Height ~ Fert,data=fastPlant, subset= (Light=="Low") )
```

Based on this output: does fertilizer amount affect plant growth when lighting is low? Add a comment to your script about this. However, one concern we have about this analysis is that power can often be an issue for t-tests, which may make you reconsider your conclusion. To see how much confidence we should place in this finding, we'll utilize R's built in power test:

"`power.t.test()`" Open the help for the power function to see the details of which options we need to include. For this test, we need to supply 4 of these 5 parameters:

- `n` = sample size per group
- `delta` = the difference between the groups
- `sd` = the standard deviation of the data
- `sig.level` = the significance level desired, and
- `power` = the portion of tests in which a significant result will occur

...and R will calculate the fifth parameter. First, we are interested in the power we have to detect an effect of fertilizer, assuming that our data are an accurate reflection of the real population. We could calculate each of the values directly, but we have already generated some of them. Our `plotmeans()` graph tells us how many samples are in each group. The `t.test()` output gives us the two means, from which we can calculate the difference. However, we need to calculate the standard deviation, which we can calculate in R using the `sd()` function. Add the numbers you found in place of the X's below:

```
## Calculate the standard deviation
sd(fastPlant$Height[fastPlant$Light=="Low"])

#Calculate the power of the T test
power.t.test(n=X,delta=X,sd=X,sig.level=0.05)
```

Based on this, what proportion of the time would we have been able to identify a significant result? Does this change your confidence in the p-value you calculated above? Add comments to your script addressing these questions.

Next, we can calculate the number of samples we would need to collect, given the data, to get a significant result 90% of the time. Copy your power test to a new line, and remove the "`n =`" option, then add the option "`power = 0.9`" to the command. This tells R to calculate the n (now missing) necessary to achieve a significant result in 90% of the experiments testing this. How many samples would you need to collect to reliably detect your difference? This is an important point to remember when evaluating the output of statistical tests.

2.4.2. Normality and the t-test

T-tests rely on the assumption that the data being analyzed are normally distributed. When this assumption is violated, the results of the test may not be informative. In this section, we will use the hotdogs data to illustrate this point. First, copy (from your old script) the lines that read in the `hotdogs` data. Then, copy over and run your final plot (with colored points) to help you visualize the data.

Now, lets see if there is a significant difference in Sodium between the Poultry and Beef hotdogs. We can use the formula input of `t.test()` again, though note that we need to omit the data we are not analyzing. To do this, we will utilize the "does not equal" logical operator, which in R is implemented as "`!=`". Because the t-test compares two groups, R will throw an error if all three Types are included. A later test in this chapter will allow us to analyze all of the groups simultaneously. The command should look like this:

```
## Run a t-test on sodium
t.test(Sodium~Type,subset= (Type!="Meat"),data=hotdogs )
```

Based on this result, is there a significant difference? Include a comment in your script here. It is possible that power is again an issue here, but there is actually another possible culprit: normality. Looking at the Poultry dots, do the Sodium values appear normally distributed? We can look at this more formally with a series of plots and tests. First, lets look at a histogram of the Poultry Sodium data:

```
## Make a histogram of the Poultry sodium
hist(hotdogs$Sodium[hotdogs$Type=="Poultry"],)
```

Does this look normally distributed? Next, let make a QQ-plot for the same data. Refer to the help or the previous chapter for more information on the `qqnorm()` function.

```
## Make a QQ-plot and add a line
qqnorm(hotdogs$Sodium[hotdogs$Type=="Poultry"])
qqline(hotdogs$Sodium[hotdogs$Type=="Poultry"],col='red')
```

What can you see now? Finally, let's run a normality test on the data:

```
## Test normality
shapiro.test(hotdogs$Sodium[hotdogs$Type=="Poultry"])
```

Summarize your conclusions in comments to the script.

Now that we suspect that the data are not normally distributed, we should likely use a non-parametric test. Non-parametric tests generally utilize rank-order data instead of raw values. This makes them less sensitive to the distribution of the data, and a good choice for non-normal data. One of the non-parametric equivalents of the t-test is the Mann-Whitney-Wilcoxon test, available in R as wilcox.test(). This test runs very similarly to the t.test, so copy your `t.test()` line and change "`t.test`" to "`wilcox.test`". Run the test, and interpret the result as a comment in your script.

```
## Run a non-parametric test on sodium
wilcox.test(Sodium~Type,subset= (Type!="Meat"),data=hotdogs )
```

Note here that R gives a warning when running the `wilcox.test`. Generally, a warning means "be aware of this" or "take this with a grain of salt" or even "are you sure you meant to do this?" However, in most cases, as long as you understand the source of the warning, you can safely use the result, just be cautious. In this case, the warning is caused by the fact that some of Calorie and Sodium values are the same for two different hotdogs. Because non-parametric tests use rank orders, this means that the values are "tied" for a particular rank. Thus, the p-value calculated may not be completely accurate because the test assumes no ties. Here, that is ok, as the p-value is still going to be close enough for our purposes.

It is important to note that non-normal data can cause spurious results in both directions. False negatives, like we saw here, or false positives, which would incorrectly reject the null-hypothesis. When running statistical tests, it is a good idea to check your assumptions before running the tests to avoid erroneous conclusions.

2.5. Chi-squared Test

Another useful statistical test is the chi-squared test for independence of data, available in R as "`chisq.test()`". This is usually, though not always, count data, similar to the `violationsData` set from last lab. From your old script, copy the lines to import the violations data, attach it, change the variable order, and make the `violationsTable`.

Looking at the `violationsTable`, we can see that the high fine levels are very rare. Such low values cause problems for the chi-squared test, so first lets see what happens when we run a `chisq.test()` on the first two rows of the table:

```
## chi-squared test on low level fines
chisq.test(violationsTable[1:2,])
```

What does this result suggest? Add a comment to your script interpreting the table. Now, copy the line and re-run the test with the full table, and look at the resulting warning:

```
chisq.test(violationsTable)

## Warning: Chi-squared approximation may be incorrect
```

Note that the approximation may not be correct, in this case because the sample sizes are too small in some of the cells. Taken with our previous result, this may suggest that a larger sample may reveal a significant relationship between Year and Fine amount at all levels.

2.6. ANOVA and linear models

Finally, today we will cover another important statistical test: Analysis of variance, typically called ANOVA. This test allows us to compare several groups at once, and to analyze more than one variable. There are several flavors of ANOVA that you may encounter; today, we will work through how to run three of them in R: One-way and two-way ANOVA, and using linear models for blocked designs.

2.6.1. One-way ANOVA

The simplest kind of ANOVA is the one-way ANOVA, which compares the means of groups separated by a single factor. In principle, any number of groups can be analyzed; however, for only two groups, the test is equivalent to the t-test. To illustrate this, lets analyze some of the hotdog data, specifically: do different types of hotdog have different calories? Make sure the hotdog data is loaded and pull up the plot to help you visualize as you go. Now, let's run an ANOVA using the built-in R function `aov()`:

```
## Run an ANOVA
aov(Calories ~ Type,data=hotdogs)
```

This gives us much of the information that we need, though not all of it. Importantly, it also provides us with a warning and a reminder: these data are unbalanced, violating one of the assumptions of ANOVA. For a one-way ANOVA, this is not a major problem, but it can be a concern for more complicated designs.

What we would really like from R, however, is an F-statistic, and a calculated p-value. Each could be calculated by hand from this output, but that seems like work. Luckily for us ~~lazy~~ efficient programmers, R already has a method to do this for us: `summary()`. The function `summary()` works differently on different classes of data, but for most statistical tests, it will output the sort of final information you are likely to want. So, first we will save the result of the `aov()` call, which works just like other variables, and can store the results of most R functions.

```
## Run and save an ANOVA
anovaResult <- aov(Calories ~ Type,data=hotdogs)
## Display the results
summary(anovaResult)
```

What does this result suggest about the relationship between Type and Calories? Knowing whether or not there is an effect of Type on Calories still doesn't tell us *what* that effect is. Below, we use two R functions to show more detailed information. The `model.tables()` function calculates the effects (or the means) of each group, while the function `TukeyHSD()` computes the post-hoc Tukey test and the adjusted p-values for multiple-testing. Use the R code below to run these functions, then add comments to your code describing the results.

```
## Generate the table of effects
model.tables(anovaResult)

## Generate a table of means instead (more intuitive)
model.tables(anovaResult,type="means")

## Run a statistical test to determine which groups differ
TukeyHSD(anovaResult)
```

Note the relationship between these outputs. The `model.tables()` function outputs the effect sizes, which can be interpreted as the difference between the group mean and the grand mean. Adding the "`type='means'` " portion gives you the actual group means, along with the calculated grand mean. Finally, `TukeyHSD()` yields the confidence intervals for the difference between each pair of groups, adjusted for multiple testing. That is, the output of `TukeyHSD()` tells you the difference in the means, the lower and upper limit of the confidence interval, and a p-value for the probability of being that much (or more) different from zero, given the null hypothesis. Each of these outputs provide a useful insight into the data.

2.6.2. Two-way ANOVA

Next, we are going to introduce the two-way factorial ANOVA. Plot the full `plotmeans()` plot for your fastPlant data again. What do you notice about the effects of Light and Fert in relation to each other? We can use the `interaction.plot` function to make this even more clear:

```
## Make an interaction plot for the fastPlant data
interaction.plot(x.factor=fastPlant$Fert, trace.factor=fastPlant$Light,
                 response=fastPlant$Height, main="TITLE")
```

It appears that the two treatments may be affecting one another, but we would like to know more about exactly how. First, let's run an ANOVA describing the effects of both treatments on Height. The syntax is similar to the one-way sANOVA, but we are also interested in analyzing a second variable, using the "+" to tell R to anlyze both.

```
plantAnova <- aov(Height ~ Light + Fert,data=fastPlant)
summary(plantAnova)
```

What does this finding suggest? However, there is still more that we can analyze, particularly the interaction between the two treatments. To do this, we replace the "+" with an asterisks (*) to tell R to analyze the interaction between the two, like this:

```
plantAnovaInt <- aov(Height ~ Light * Fert,data=fastPlant)
summary(plantAnovaInt)
```

What additional information does this provide?

2.6.3. Using linear models

In R, we have been calling the `aov()` function directly; however, R is internally using linear models to run these tests, and these linear models can be accessed directly. This can be especially important for analyzing data that are linear, instead of grouped neatly, or when there are random factors, such as individual, that can add information to our analysis. We have one such data set in the scale squeeze data collected in a biostats class. Each student pressed on a bathroom scale with their dominant arm, with their elbow resting on a book so their arm was parallel to the surface of the scale. Scale readings were recorded in 5-second intervals, with the first reading taken once the scale initially reaches a maximum.

We will analyze these data looking at the effect of fatigue on pressing, taking into account the differences in who was pressing. Load in the data set, either using `read.csv()`, or the menus, but don't forget to paste the command into your script. It is good practice to look at what is in the dataset to ensure it read properly.

```
## Read in the data
scaleSqueeze <- read.csv("data/scaleSqueeze.csv")

## See what is there
head(scaleSqueeze)
```

Now plot the data:

```
plot(scaleSqueeze$pounds ~ scaleSqueeze$time, main="TITLE")
```

Look at the data and add a comment to your script describing the pattern(s) you see. From this, we might be interested in an ANOVA to tell us about the effect of time (fatigue) on the strength of the press. Run (and save and display) a simple ANOVA doing this with the command:

```
## Anova test and output
scaleAnova <- aov(pounds ~ time,data=scaleSqueeze)
summary(scaleAnova)
```

What conclusion can you draw from this test? What does it suggest is happening, and does this match your previous prediction?

However, this only tells us *that* time has an effect, not what that effect is. For that, we can use linear models via the `lm()` function. This function constructs a linear model, giving us both an intercept and the effect of the variables. Let's look at this and compare it to our `aov()` output.

```
## Anova test and output
scaleLM <- lm(pounds ~ time,data=scaleSqueeze)
summary(scaleLM)
anova(scaleLM)
summary(scaleAnova)
```

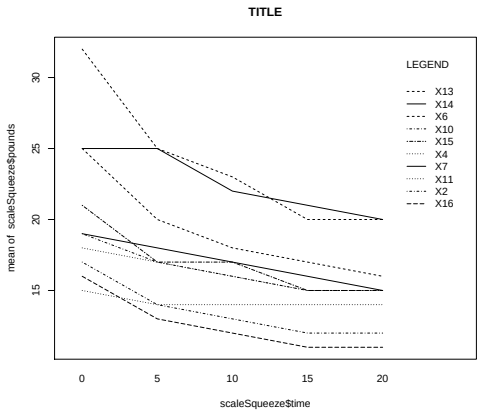
Note first that the `summary()` of the `lm()` output tells us the effect of time on pushing: it appears that each unit of time (seconds) reduces pressing by 0.264 pounds. Further, what do you notice about the p-values for the three outputs? How about the differences between the two ways of getting an anova output?

The result looks fairly clear, but we have additional information that we can use. It seems like the pattern may be stronger even than the test calculated, but it may be hampered by the large

amount of individual variation. Luckily, we know which individual is measured at each time, and we can give this information to the computer.

First, let's plot the data using the individual information to give ourselves a better sense of what we have. We could color code each individual, like we did for the hotdog data by Type, but that is a bit cumbersome, and we are ~~lazier~~ more efficient than that. There is, as is usually the case, an R function that will do the work for us: `interaction.plot()`. Use the help or tab for more details on this command, but the below syntax should produce a nice plot for you:

```
## Add individual information to the plot
interaction.plot(x.factor=scaleSqueeze$time, trace.factor=scaleSqueeze$Ind,
                 response=scaleSqueeze$pounds, main="TITLE",
                 trace.label='LEGEND')
```



What information did this add? Now, we want to add this same information into our `lm()` and `anova()` analyses. This syntax will run the analysis for you:

```
scaleLMwithInd <- lm(pounds ~ time + Ind,data=scaleSqueeze )
summary(scaleLMwithInd)
anova(scaleLMwithInd)
```

What additional information is provided by this test? Add a comment to your script to interpret your findings, specifically the difference found when including individual in the model.

2.7. Assignment

Click on the small notebook icon just above your script. It should say "Compile an HTML notebook from the current R script" when you mouse over it. Assign an appropriate title, and put your name as the author. Click "Compile" and an html file will be created. Submit these as instructed for the course. Refer to the previous chapter for more details.

Chapter 3:
Further Reading

3.1. Background

This R tutorial is, intentionally, a very basic introduction to using the software package. It introduces the basic syntax and some of the most basic uses of the functions in R. This is a good place to start, and has hopefully built you a strong base of understanding of the language, syntax, and general enviroment. However, it is likely not quite enough to get you going for using R to analyze your own data if the structure or question are even slightly different from what is presented here.

There are a number of topics still left unaddressed in this tutorial, including writing your own functions, advanced control of plotting, additional statistical tests, ways of manipulating data, and many, many others. The intention is to continue expanding this tutorial, so please check back to our website (https://bitbucket.org/petersmp/publishedtutorials) if you are looking for something more . However, no amount of expansion will ever encompass all of the usage cases you might encounter. To help you find more answers, below we offer two strategies: searching for a specific question and a list of other tutorials.

3.2. Specific searches

No matter how long you have used R for, there is always something new to learn. In writing this tutorial, to match an earlier version using different software, we had to figure out how to generate a fairly specific version of a barplot – something we hadn't done before. The solution was simple: run to Google! A search for a few variants of "Make side by side barplot in R" got us started. By reading through a few answers, we were able to put together the plot we wanted.

These answers can often be a bit intimidating to novice users, as it seems that there are a dozen different solutions. How can you be expected to pick the right one? The real answer is: all of the solutions are usually "right." The flexibility of R means that the same problem can usually be solved in a lot of different ways. Find one that you understand, implement it, then modify it to fit your exact needs.

This is a good point to point out a great quote: "This is R. There is no if. Only how." (Simon 'Yoda' Blomberg; R-help; April 2005; retreived from the R package "fortunes"). Anything you want to do in R is possible, it is just a matter of finding a way that you can implement.

There are a number of sites that offer R specific searches that may be easier to get started with. Each of these is geared specifically to R, and each handles this in a slightly different way. None is inherently better than the others (or than Google), but rather each has a different purpose and flavor. You may find that one is the most comfortable for you of that which you choose to use varies by your current question.

- http://www.rseek.org/ A search engine speicific to R
- https://stat.ethz.ch/mailman/listinfo/r-help An R help mailing list
- http://stackoverflow.com/questions/tagged/r A great general purpose help site here linked directly to the "r" tag to help you get straight to R specific help
- http://stats.stackexchange.com/questions/tagged/r A related general help site but with a slightly broader focus on the issues related to programming.

As always, don't forget that searching the help functions in R directly (e.g., by using "?" or "??" as shown in this tutorial) can often get you started, and may at the very least help you refine your search terms

3.3. Additional tutorials

If you want to go further in R, it is likely necessary to get a bit of guidance, rather than just focus on solving your own specific problems. To that end, there are a lot of additional tutorials, and you are bound to find one written at the level you want and with a focus on the issues you are interested in. Some are available in print format (I think these are called "books") and may be especially helpful as a bigger picture overview. There are also a lot of different tutorials available online, that may help you.

The official "Introduction to R" offers a very detailed look at many R processes and is available at http://cran.r-project.org/doc/manuals/r-release/R-intro.html .

Alternatively, there is a long list of resources aimed at everything from complete novices to advanced users posted at http://stackoverflow.com/tags/r/info which links to many, many resources. This list is more exhaustive than any we could compile, and will be kept up to date as new resource emerge.