ENV 710 – Applied Statistics
Lab 1: Introduction to R

What is R?

R is powerful software for interacting with data. With **R** you can create sophisticated graphs, you can carryout statistical analyses, and you can create and run simulations. **R** is also a programming language with an extensive set of built-in functions, so you can, with some experience, extend the language and write your own code to build your own statistical tools.

R is an open source implementation of the S language, which has been around for more than twenty years and has been the most widely used statistical software in departments of statistics for most of that time, first as S and then as the commercially available S-PLUS. A core team of statisticians and many other contributors work to update and improve **R** and to make versions that run well under all of the most popular operating systems. Most importantly to you, **R** is free, high-quality statistical software that will be useful as you learn statistics even though it is also a first-rate tool for professional statisticians.

As described on the R project homepage:

> "**R** is a system for statistical computation and graphics. It consists of a language plus a run-time environment with graphics, a debugger, access to certain system functions, and the ability to run programs stored in script files.
>
> "The core of **R** is an interpreted computer language which allows branching and looping as well as modular programming using functions. Most of the user-visible functions in **R** are written in **R**. It is possible for the user to interface to procedures written in the C, C++, or FORTRAN languages for efficiency. The **R** distribution contains functionality for a large number of statistical procedures. Among these are: linear and generalized linear models, nonlinear regression models, time series analysis, classical parametric and nonparametric tests, clustering and smoothing. There is also a large set of functions which provide a flexible graphical environment for creating various kinds of data presentations. Additional modules are available for a variety of specific purposes."

R Resources

There are hundreds of books, online resources and documents, blogs, and webpages on **R**. See the R Shortcuts under the Resources file on the class Sakai webpage for a few useful **R** resources and shortcut reference cards. Or, look under Documentation on the R project homepage for a list of manuals, FAQ's, books, etc. This introductory document scratches the surface of **R**, but is hopefully detailed enough to get your started using it.

Downloading R from the Web

Go the R project homepage at http://cran.us.r-project.org/.

Download **R** for Windows. Click on the link “Download R for Windows”, then click on the link “base”, and finally click on “Download R 3.0.1 for Windows” (or the recent version which could have a larger number). This begins the download of a file whose size is about 52MB. After the download is complete, double click on the downloaded file and follow the on screen installation instructions.

Download **R** for (Mac) OS X. Click on the link “Download R for Mac OS X, then click on “R-3.0.1.pkg (latest version)” (or the most recent version which could have a larger number) to begin the download. Once it is downloaded in your Downloads folder or on your Desktop, double click on the downloaded file. Your Mac will unstuff the downloaded file and create an **R** folder. Inside this folder, there are many files including one with the **R** logo. Drag a copy of this to your Desktop and then drag the whole **R** folder to the Applications folder (located on the hard drive). After completing this, you can drag the original downloaded file to your trash bin.

Getting Started in R

This document will describe some of the basic functionality of **R**. Usually, you interact with **R** through a command-line interface — you type or paste a command in the console and **R** responds. Alternatively, commands can be written in a text editor and submitted to the console using the `source()` command. Once you have mastered a few commands, the command-line interface gives you control of an extremely powerful tool for interacting with data. Gaining mastery of a few **R** commands does take some learning and patience, as **R** is finnicky. You will undoubtedly experience some challenges as you work to learn a new skill that is not wholly intuitive, but it is well worth the effort. Onward!

The command–line version of **R** provides us with two prompts, that is, two signals that it is ready for input. They are: the caret

```
>
```

which means that **R** is ready for a new command, and the plus sign:

```
+
```

which means that R is waiting for you to finish the current command. A principal advantage of the command line interface (CLI) is that it simplifies the development and use of scripts. These allow us to keep a permanent, documented text record of the steps that we have done.

Also note that the number sign # can be used to annotate code within a text editor or text file. Anything after the # will not be run.

The following commands give a glimpse of what **R** can do. You will find it most beneficial to type these commands into **R** to see the results yourself.

Using R as a Calculator

You can use **R** like a calculator. The * symbol stands for multiplication and the ^ symbol stands for exponentiation.

```
2 + 2
12 * 3 - 10/2 + sqrt(16)
3^2
1:10
sum(1:10)
mean(1:10)
sd(1:10)
log(10)
log10(10)
```

The colon operator `:` creates an array of numbers from the first to the second. **R** has a number of built-in functions such as `mean()`, `sum()`, and `sqrt()` that have obvious meanings. The command `sum(1:10)` calculates `1+2+3+···+10`. Note that the `log()` is the logarithm in base e, whereas `log10()` denotes a base of 10.

The symbol `[1]` that precedes the output says that the first row begins with the first number of the output. Long answers are broken across rows, like in this example with similarly useful row labels.

```
1:100
```

R can do arithmetic operations on arrays. If you multiply an array of numbers by a single number, the multiplication happens separately for each number. You can also add or multiply equal-sized arrays of numbers.

```
2 * (1:15)
(1:10) + (10:1)
```

Assigning variables

You can use the = or the key combination <- signs (created to look like an arrow) to create variables. For example, type the below to assign a vector of numbers to `a`.

```
a = 1:10
a
a <- 1:10
a
```

Variable names can consist of letter and numbers, but not symbols.

Combining arrays.

The `c` function in **R** concatenates things. (Because `c` is a reserved function name in **R**, it is preferable not to use `c` as the name of a variable lest you or **R** gets

confused.)

```
a = 1:10
b = 15:20
cc = c(b, a, b)
cc
```

Using functions

R has many built in functions and you can also write your own. To see a function, just type its name. If you want actually to use the function, you need to add parentheses at the end, possibly with arguments in between.

```
mean
?mean
mean(cc)
```

To get information on the way to parameterize a function, type `?` before the function. For example, `?mean` brings up a pop-up box that shows the full usage of `mean` `mean(x, trim = 0, na.rm = FALSE, ...)`.

Here is code to create a function that computes the area of a rectangle. This function is fairly simple and more sophisticated functions are more useful.

```
findArea = function(x, y){x * y}
findArea(13, 4)
```

Changing your workspace.

The workspace is your current **R** working environment and includes any user-defined objects (vectors, matrices, data frames, lists, functions). Use the function, ls() to get a list of objects in your workspace.

```
ls()
```

To delete all the objects in your workspace, type:

```
rm(list=ls())
```

Use *Ctrl-C* or *Esc* to stop processing. This will work most of the time, and you will rarely need to ask your operating system to intercede.

Quit **R** altogether by typing the command `q()`. When you do so, **R** will ask you if you want to save your workspace. This allows you to have variables you have previously defined available without the need to create them again from scratch. Generally, whether or not you choose to save the workspace depends on your workflow. Personally, I almost never do it. I prefer to write scripts, which are a complete record of the analysis performed, and if I need to recreate the analysis, I resource the scripts.

If you choose to save the workspace then a compressed image of the objects, called `.RData`, will be saved into your working directory. To easily access these objects again in a future session, use the load function.

```
load(".RData")
```

Working Directory

The working directory is the location to and from which **R** writes and reads files by default. In Windows, there is a menu item, *Misc*, that allows for selecting the working directory. In CLI, you can use command lines.

```
getwd()                 # Returns the current working directory.
setwd("C:/Temp")        # Set Temp to be the working directory.
```

One easy way to keep your files organized is to have a `setwd` function at the top of your scripts. Then you run the function before running the script so that all the data, etc. can be easily located. For example, at the top of the script file type:

```
setwd("c:/path/to/my/directory/")
```

Here, `setwd()` is a function; we use functions to tell **R** what we want it to do, and `c:/path/to/my/directory/`is an argument for the function. Arguments are the main way that we tell **R** what to do the function upon. In this case, we use the argument to tell **R** that we would like to use `c:/path/to/my/directory/` as the working directory. The forward slashes are used regardless of the underlying operating system. If you want to read or write to a different location, you must explicitly say so. Life is therefore much easier if all the data and scripts for an analysis are kept in a single (frequently backed up!) location.

A hint for Windows users in setting the working directory: open Windows Explorer to your desired working directory, right click in the address bar and click "copy address as text". Now paste it into the `setwd()` command. Make sure to change \'s to /'s and execute `setwd()` function.

Convention

There are many different ways to do things in **R**. There are no official conventions on how the language should be used, but the following thoughts may prove useful in communicating with long-time **R** users.

- Although the equals sign = does work for assignment of variables, it is also used for other things like passing arguments. By contrast, the arrow <– is only used for assignment: most experienced **R** users will use the arrow.

- Spaces are cheap. Use spaces liberally between arguments and between objects and arithmetic operators.

- Call your objects useful names. Don't call your model `model`, or your dataframe `data`. Use names that you will be able to understand at a later date; for example, months later when you need to revise your manuscript for publication.

- You can terminate your lines with semi-colons, but don't.

Helpful Hints and Shortcuts

Remember, **R** is case sensitive.
To **R** run an individual line of code from a text editor, place the cursor at the beginning of the line and hit, C*trl-r*. On a Mac, hit *Command return*. Alternatively, type the command into **R**, or copy and paste the command into **R**.

If you hit the up arrow, the command that you entered previously will appear. *Ctrl-e* moves the cursor to the end of the line. *Ctrl-a* moves the cursor to the beginning of the line. To get help on a command type `help()` or `?()` with the command of interest in the parentheses.

```
help(hist)
?hist
```

Data Manipulation and Storage

There are several different ways to store data in a workspace, but the following four will be essential four our use: vectors, strings, dataframes and matrices.

Vectors are 1-dimensional strings of data. To assign a data vector, use `c()`.

```
emissions <- c(52.134, 82.741, 57.978, 100.0, 93.702)
emissions
```

R also allows character strings (vectors of words and characters).

```
Simpsons <- c("Homer", "Marge", "Bart", "Lisa", "Maggie")
Simpsons
```

Vectors can be combined and sorted. What happens when you change the T (True) to a F (False) after decreasing in the `sort()` function?

```
emissions2 <- c(91.368, 89.111, 72.989, 95.197, 85.318)
emissions.total <- c(emissions, emissions2)
emissions.total

sort(emissions.total)
sort(emissions.total, decreasing=T)
```

You can also do vector math. What happens when you change the value after digits to 3?

```
emissions.total*pi
round(emissions.total, digits=1)
```

You can create vectors of data based on regular sequences using `rep()` and `seq()`. What does each line of code do?

```
rep(0, times = 5)
rep(c(1,2,3), each = 5)
seq(1,10)
seq(-5,5, by=0.1)
rep(seq(1,10), 3)
```

You can use functions on a data vector.

```
sum(emissions.total)        # sums vector of data
length(emissions.total)     # length of data vector
cumsum(emissions.total)     # returns running tally
```

R also has several summary statistics that are extremely useful for data analysis.

```
mean(emissions.total)       # mean of data vector
sort(emissions.total)       # sorts values
min(emissions.total)        # minimum value
max(emissions.total)        # maximum value
range(emissions.total)      # minimum and maximum value
median(emissions.total)     # median value
summary(emissions.total)    # five point summary
sd(emissions.total)         # standard deviation
```

Note that you cannot (and should not) use an **R**-defined function (like `median()`) as a vector name.

Dataframes can store collections of variables of many different types (strings, numbers, etc.). Dataframes are extremely useful and flexible for statistical data analysis.

```
dframe <- data.frame(Number=c(1:20),
           Emissions=emissions.total,
            School=c(rep("Fuqua", 10), rep("NSOE", 10)))
```

Just like it sounds, a matrix is a two-dimensional set of numbers (cannot include strings).

```
matbin <- matrix(c(1,0,0,0,0,1,1,0,1,0,1,1), ncol=3,
           dimnames = list(c("R.1", "R.2", "R.3", "R.4"),
            c("C.1", "C.2", "C.3")))
matbin
```

```
mrandnums <- matrix(rnorm(100, mean=10,sd=100), nrow=20)
mrandnums
```

In the first matrix (`matbin`), we assign the vector of 0's and 1's to a matrix with 3 columns and define the column and row labels using `dimnames`. In the second matrix (`mrandnums`), we create a list of random numbers from a normal distribution and then assign them to a matrix with 20 rows. You may be asking why we would want to create either of those matrices. They are just examples of how matrices can be created and defined.

Graphing

R can create extremely sophisticated and attractive graphs and figures. Below are some simple plots to get started. If you look up R Graph Gallery on the Internet you can get a taste of the diversity of figures that can be created with R.

Some of the built-in functions for creating graphs include:

```
boxplot()
hist()
plot()
barchart()
pie()
```

The following command constructs a boxplot. `ylab` sets the title on the y-axis. `col="green"` is the color of the box plot. `main="Title"` titles the boxplot "Title". `las = 1` changes the orientation of the y-axis labels to be parallel to the axis. `cex.lab, cex.axis, cex.main` change the font.

```
boxplot(emissions.total, col="green", main="Title",
ylab="y axis label", las=1, cex.axis=0.9, cex.lab=1.5)
```

There are many commands to manipulate the graphical parameters of the plot. They can be found by typing: `?par`.

You can save the plot in several ways. By typing `?savePlot()` you can find all the possible formats: png, jpeg, tiff, bmp.

```
savePlot("myHistogram", type = "jpeg")
```

Note that `savePlot()` does not work with Macs. As an alternative, with the plot as an active window, go to *File->Save as->pdf*.

A note on saving graphs with MacOS… It's tempting to just create graphics to the on-screen device (such as X11 on Linux or Quartz on MacOS) and then just use *Save As* from the menu. However, this doesn't allow you to explicitly set the options for the device, and on some platforms, you don't even get to choose the file format. Also, if you resize the graphics window after you create the graph, you can get some unexpected results.

The best practice is to create a script file that begins with a call to the device driver (usually `pdf` or `png`), runs the graphics commands, and then finishes with a call to `dev.off()`. For example:

```
png(file = "mygraphic.png",width = 400,height = 350)
plot(x = rnorm(10), y = rnorm(10), main = "example")
dev.off()
```

Not only will you often get better-looking results, you'll have the means to recreate the graphic file six months down the line, when you've long forgotten how you did it manually.

LOADING DATA INTO R

Now let's load data into R from an existing data file. In this example we will use the `mississippi.csv` data set found on Sakai. This dataset includes yearly water discharge volumes (in cubic kilometers) from the Mississippi River from 1954 to 2001. First, download the data file to your work file for the class and make sure your work directory is set correctly to locate the data.

```
mississippi <- read.table("mississippi.csv",
                header = TRUE, sep = ",")

mississippi <- read.csv("mississippi.csv",
                header = TRUE, sep = ",")

names(mississippi)
```

Note that in the above code, we specified the type of data file (`.csv`) when we called the file name. `header = TRUE` defines the first line of the data file as a header line. `sep = ","` states that the columns are separated by commas. `names()` lists all the variables in the data file.

To list all the objects in your current work environment, type `objects()` or `ls()`.

To remove objects from your current work environment, type `remove()` or `rm()`.

`head()` shows the first several rows of a data.frame, matrix or table.
`tail()` shows the last several rows of a data.frame, matrix or table.

```
head(mississippi)
tail(mississippi, 10)
```

To select individual cells from a vector or dataframe, we define the values we are searching for in closed brackets. For example, the below commands provide (a) the first Discharge data point, (b) the first ten values, and (c) the 1st, 2nd, and 4th values.

```
mississippi$Discharge[1]
```

```
mississippi$Discharge[1:10]
mississippi$Discharge[c(1,2,4)]
```

You can use indexing to select specific rows of data or specific data points. Remember, that when indexing a dataframe or matrix, it is specified as `data.frame[rows, columns]`. For example, below I do the following: (a) index the first row of data, (b) index the 3rd through 9th rows of data, (c) index just the `discharge` column.

```
mississippi[1,]
mississippi[3:9,]
mississippi[,2]
```

You can assign a column of data to a variable of its own:

```
misdis <- mississippi$Discharge
```

The `attach()` command attaches the database to the **R** search path. This means that objects in the database can be accessed by simply typing their names. The `detach()` allows you to reverse the attach command. For example, if you `attach(mississippi)` then you can work directly with the column names as variables.

```
Year
Discharge
```

BE CAREFUL with the `attach()` command: if you have objects with the same name in different files, it will remember the last time the object was named.

We could make a time series plot of water discharge. This entails first defining the object `misdis` as a time series vector

```
discharge.ts = ts(misdis, start=1954, frequency=1)

plot(discharge.ts, ylab="Annual Discharge into the
    Mississippi River (km3 of water)",
     main="Yearly Discharge of the Mississippi River (in
      cubic kilometers of water)")
```

To be honest, this is an ugly plot with poor axis labels. We will improve upon this in the future…

Lab 1: Introduction to Probability in R

The goals of the first lab are to get familiar with **R**, to understand a few of its applications for probability, and to master the basics of visualizing and summarizing data – a skill which will be used in all the other labs.

At the end of each section, there are a few questions to answer. Please type your answers to each problem, including any requested graphs, in a Word document. *Submit your answers and your R-code to the class Sakai site under the folder Assignments before 5 pm on either Mon., Sep. 1 or Wed., Sep. 3.*

A Few Applications Dealing with Probability…

Rolling dice in R

```
roll <- rbinom(n=10, size=1, prob=0.5)
```

This function makes use of the binomial distribution, which we will talk about next week. The binomial distribution yields the number of successes in a sequences of yes, no experiments. `rbinom()` chooses a random deviates from the distribution. Here `prob` assigns the probability of success on each trial: it's like assigning a 50% probability of getting a heads rather than a tails on a coin flip.

We can also flip a coin using the function `sample()`. Note, however, that we have no control over the probability of getting heads. Each element has an equal probability of being chosen.

```
x <- sample(c("H","T"), 10, replace=T)
table(x)/10
```

Problem 1: Using the above function, flip a fair coin 50, 500, and 5000 times. Then flip a coin 50, 500, and 5000 times with a probability of 57% of getting a heads. What are the observed proportions of heads that you got for each flip? How would you code a coin flipping experiment of 50 flips of a fair coin that is replicated 100 times? `?rbinom` might be helpful.

Show the code for each coin flip. Write a paragraph describing your results.

Conditional probability in R

Let's review conditional probability, starting with the "easy" example that we used in class of the number of long months and number of months that contain an "r". Remember that we ran into a friend on the street that we know has a birthday during one of the long months, and we want to derive the probability that her birthday falls in a month with an "r".

```
Months <- c("Jan", "Feb", "Mar", "Apr", "May", "Jun",
            "Jul", "Aug", "Sep", "Oct", "Nov", "Dec")

r <- c("Jan", "Feb", "Mar", "Apr", "Sep", "Oct", "Nov",
     "Dec")
```

```
L <- c("Jan", "Mar", "May", "Jul", "Aug", "Oct", "Dec")

RintersectL <- intersect(r,L)
```

What does the `intersect()` function do?

Remember, the equation for conditional probability is: $P(A|B) = \frac{P(A \cap B)}{P(B)}$. What is the conditional probability statement if we want to know P(r|L)?

We can calculate P(L) by:

```
P_L <- length(L)/length(Months)
```

What does `length()` do here?

Add the *VennDiagram* package to your **R** library. Hint: Using the toolbar in **R** you need to go to <*Package Installer*> and follow the directions. Or, you can type `install.packages()` to get a list of packages, and go from there. Remember that once you install the package in your **R** library, you still have to load it into your workspace with `require()`. Here is an example.

```
require(VennDiagram)

venn.diagram(list(B = 1:1800, A = 1571:2020),
             filename = "trial1.tiff")

venn.diagram(list(B = 1:1800, A = 1571:2020,
             C = 1600:1700),
              fill = c("red", "green", "blue"),
               alpha = c(0.5, 0.5, 0.5), cex = 2,
               cat.fontface = 4,lty =2,
                filename = "trial2.tiff")
```

In the `venn.diagram` function there is a lot of information that defines how the diagram will look (e.g., `fill, alpha, cex, cat.fontface, lty`). Type `?venn.diagram` to see what these do exactly. Note also that the Venn diagram figure does not show up in our workspace, but is being saved in your directory.

Problem 2: Answer the following questions using **R**: (a) What is the probability of r? (b) What is the probability of RintersectL? (c) What is P(r|L)? Make sure to show how you did this in **R**. (d) Finally, make a Venn diagram of this problem.

Problem 3: Last season, Duke forward Jabari Parker had a free throw percentage of 0.748, a three-point percentage of 0.358, and a field goal percentage of 0.473. Would Parker have the best chances of making three points by (a) shooting a single three-point shot, (b) shooting three free throws, or (c) shooting a two-point field goal and a free throw (forgetting that he would have to get fouled first)?

Problem 4: ***Summarizing data in R - CO_2***

Download the data file ("Country CO2.csv") from Sakai. The data file lists carbon dioxide (CO2) emissions per person for countries with populations of at least 20 million people. Use this data set for problem 4.

(a) Create a boxplot of the data. What is the five-number summary (minimum, first quartile, median, third quartile, maximum). What is the standard deviation of the data set? Discuss the shape of the distribution. (b) Which countries are outliers according to the 1.5 x IQR rule? Develop and describe a histogram of the data. Do you agree with the rule's suggestions about which countries are and are not outliers?