Introduction to R - Lab Practice

This lab on the Introduction to R comes from "Introduction to Statistical Learning with Applications in R" http://faculty.marshall.usc.edu/gareth-james/ISL/index.html by Gareth James, Daniela Witten, Trevor Hastie and Robert Tibshirani.

The `tidyverse`

The textbook was written using `base R` functions, which can be unintuitive. For our labs, we will be using the so-called 'tidyverse,' which is based on the idea of tidy data and functions. Tidy data has observations as rows and variables as columns. All the observations must be at the same observational level. Tidy functions are pure, pipeable, and predictable. We'll begin to understand all of this more fully as we start to work with data.

Because the tidyverse is outside the `base R` functions, we must use packages (sometimes called libraries) to get the additional functionality. Packages are discussed further in Chapter 3. For now, lets just load a bunch of packages we'll need for our work.

```
library(ISLR)
library(dplyr)
library(readr)
library(ggplot2)
library(GGally)
library(rlang)
library(shiny)
#library(mosaic)
library(manipulate)
```

Getting error messages? You may need to install the packages first using `install.packages()`.

Loading Data

For most analyses, the first step involves importing a data set into `R`. For this class, a lot of the data comes from the package itself, so we can use the `data()` function to import it.

We begin by loading in the `Auto` data set. This data is part of the `ISLR` package. Install ISLR package (install.packages("ISLR"))

```
Auto=read.table("Auto.data")
fix(Auto)
Auto=read.table("Auto.data",header=T,na.strings="?")
fix(Auto)
Auto=read.csv("Auto.csv",header=T,na.strings="?")
fix(Auto)
dim(Auto)
Auto[1:4,]
Auto=na.omit(Auto)
```

```
dim(Auto)
names(Auto)
```

Nothing happens when you run this, but now the data is available in your environment. (In RStudio, you would see the name of the data in your Environment tab).

To view the data, we can either print the entire dataset by typing its name, or we can "slice" some of the data off to look at just a subset by piping data using the `%>%` operator into the `slice` function:

```
library(dplyr)
head(Auto)
tail(Auto)
# "The first row":
Auto[1, ]

# "All rows in second and fifth columns":
Auto[, c(2, 5)]

# "The first row in second to fifth columns":
Auto[1, 2:5]
```

If data isn't internal to packages, .csv files are the most common way for data to be formatted. CSV files are simply text files, which you could alternatively open on your computer using a standard text editor. While it is often tempting to view a data set using Excel before loading it into `R`, it can introduce problems in the data later.

We can use the `read_csv()` function in the `readr` package to load in a CSV file. The help file contains details about how to use this function.

Before attempting to load a data set, we must make sure that `R` knows to search for the data in the proper directory. If you are using internal package data, this isn't a concern– just make sure you have run `library()` on the package containing the data. For external data, the easiest way make sure the data is accessible to R is to put the data in the same folder as your Jupyter notebook. We have a file called `Auto.csv` in this directory, so we can read it in without worrying about the file path.

When we read in data that is external, we must give it a name. We'll call it `Auto` and use the assignment operator `=`. The assignment operator `<-` is equivalent, and using one or the other is a matter of personal preference.

```
Auto=read.csv("Auto.csv", na="?")
head(Auto)
tail(Auto)
# "The first row":
Auto[1, ]

# "All rows in second and fifth columns":
Auto[, c(2, 5)]

# "The first row in second to fifth columns":
Auto[1, 2:5]
```

Notice that the data looks just the same as when we loaded it from the package. Now that we have the data, we can begin to learn things about it.

```
dim(Auto)
```

```
str(Auto)
```

```
names(Auto)
```

The `dim()` function tells us that the data has 392 observations, or rows, and nine variables, or columns. The original data had some empty rows, but when we read the data in `R` knew to ignore them.

The `str()` function tells us that most of the variables are numeric or integer, although the `name` variable is a character vector.

`names()` lets us check the variable names.

Summary statistics

Often, we want to know some basic things about variables in our data. `summary()` on an entire dataset will give you an idea of some of the distributions of your variables.

The `summary()` function produces a numerical summary of each variable in a particular data set.

```
 summary(Auto)
```

The summary suggests that `origin` might be better thought of as a factor. It only seems to have three possible values, `1`, `2` and `3`. If we read the documentation about the data (using `?Auto`) we will learn that these numbers correspond to where the car is from: 1. American, 2. European, 3. Japanese.

```
?Auto
```

So, lets `as.factor` that variable into a factor (categorical) variable.

```
library(sf)
library(dbplyr)
library(magrittr)
require(magrittr)

Auto$origin = as.factor(Auto$origin)
summary(Auto)
```

We can also look at statistics for a particular variable with the function `favstats()`

```
#library(mosaic)
#favstats(~mpg, data=Auto)
```

Or, just look at one particular statistic using `mean()`, `sd()`, `median()`, `IQR()`, and more.

```
mean(Auto$displacement, data=Auto)
```

Plotting

We can use the `ggplot2` package to produce simple graphics. `ggplot2` has a particular syntax, which looks like this

```
ggplot(Auto) + geom_point(aes(x=cylinders, y=mpg))
```

The basic idea is that you need to initialize a plot with `ggplot()` and then add "geoms" (short for geometric objects) to the plot. The `ggplot2` package is based on the Grammar of Graphics, a famous book on data visualization theory. It is a way to map attributes in your data (like variables) to "aesthetics" on the plot. The parameter `aes()` is short for aesthetic.

For more about the ggplot2 syntax, view the help by typing ?ggplot or ?geom_point. There are also great online resources for ggplot2, like the R graphics cookbook.

```
?ggplot
```

The cylinders variable is stored as a numeric vector, so R has treated it as quantitative. However, since there are only a small number of possible values for cylinders, one may prefer to treat it as a qualitative variable. We can turn it into a factor, again using a mutate() call.

```
Auto$cylinders = as.factor(Auto$cylinders)
```

To view the relationship between a categorical and a numeric variable, we might want to produce *boxplots*. As usual, a number of options can be specified in order to customize the plots.

```
ggplot(Auto) + geom_boxplot(aes(x=cylinders, y=mpg)) + xlab("Cylinders") + ylab("MPG")
```

The geom geom_histogram() can be used to plot a histogram.

```
ggplot(Auto) + geom_histogram(aes(x=mpg))
```

The function warns us that it used a default number of bins, so we should think more carefully about what value makes sense.

```
ggplot(Auto) + geom_histogram(aes(x=mpg), binwidth=5)
```

For small datasets, we might want to see all the bivariate relationships between the variables. The GGally package has an extension of the scatterplot matrix that can do just that. (Be patient– it takes a long time!)

```
nncols = 9
ggpairs(Auto, columns = 1:nncols,cardinality_threshold = 304)
```

We can also produce a pairs plot for just a subset of the variables by piping just a selection of the variables into the plot.

```
Auto %>% select(mpg, cylinders) %>%
    ggpairs()
```

Sometimes, we might want to save a plot for use outside of R. To do this, we can use the ggsave() function.

```
ggsave("histogram.png",
       ggplot(Auto) + geom_histogram(aes(x=mpg), binwidth=5)
      )
```

Appendices

These sections are not required in order to understand the tidyverse material we are focusing on, but are part of the lab in the book, so we are including them here. The appendices start with making your own vectors and matrices, something which is almost never required in the tidyverse. Then, the appendices move on to more sophisticated plots (such as those in 3 dimensions) and finish up with some introduction to indexing using numbers. With our tidy tools, we will try to avoid indexing using numbers if at all possible, but you may want to look at this material to see another method for slicing data.

Making vectors and matrices

R uses functions to perform operations. To run a function called funcname, we type funcname(input1, input2), where the inputs (or arguments) input1 and input2 tell R how to run the function. A function can have any number of inputs.

For example, to create a vector of numbers, we use the function `c()` (for concatenate). Any numbers inside the parentheses are joined together. The following command instructs `R` to join together the numbers 1, 3, 2, and 5, and to save them as a vector named `x`. When we type `x`, it gives us back the vector.

```
x <- c(1,3,2,5)
x
```

We can also save things using = rather than < −:

```
x = c(1,6,2)
x
```

Hitting the up arrow multiple times will display the previous commands, which can then be edited. This is useful since one often wishes to repeat a similar command. In addition, typing `?funcname` will always cause `R` to open a new help file window with additional information about the function `funcname`.

```
?c
```

We can tell `R` to add two sets of numbers together. It will then add the first number from `x` to the first number from `y`, and so on. To do this, `x` and `x` should be the same length. We can check the length of each vector using the `length()` function:

```
y = c(1,4,3)
length(x)==length(y)
```

Looks good, so let's sum them up (I'll print the starting value of the `x` and `y` vectors, too):

```
#Value of x
x

#Value of y
y

#Value of x+y
x+y
```

The `ls()` function allows us to look at a list of all of the objects, such as data and functions, that we have saved so far:

```
ls()
```

The `rm()` function can be used to delete any that we don't want:

```
rm(x)
ls()
```

It's also possible to remove all objects at once (effectively clearing the history):

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

The `matrix()` function can be used to create a matrix of numbers. Before we use the `matrix()` function, we can learn more about it:

```
?matrix
```

The help file reveals that the `matrix()` function takes a number of inputs, but for now we focus on the first three: the data (the entries in the matrix), the number of rows, and the number of columns. First, we create a simple matrix.

```
x=matrix(data=c(1,2,3,4,5,6), ncol=2, nrow=3)
x
```

Note that we could just as well omit typing `data=`, `nrow=`, and `ncol=` in the `matrix()` command above: that is, we could just type

```
x=matrix(c(1,2,3,4,5,6),3,2)
x
```

and this would have the same effect. However, it can sometimes be useful to specify the names of the arguments passed in, since otherwise `R` will assume that the function arguments are passed into the function in the same order that is given in the function's help file. As this example illustrates, by default `R` creates matrices by successively filling in columns. Alternatively, the `byrow=TRUE` option can be used to populate the matrix in order of the rows.

```
matrix(c(1,2,3,4),2,2,byrow=TRUE)
matrix(c(1,2,3,4),2,2,byrow=FALSE)
```

(Notice that in the above command we did not assign the matrix to a value such as `x`. In this case the matrix is printed to the screen but is not saved for future calculations.)

The `sqrt()` function returns the square root of each element of a vector or matrix. The command `x^2` raises each element of `x` to the power 2; any powers are possible, including fractional or negative powers.

```
# orginal value of x
x

# square root
sqrt(x)

# values squared
x^2
```

The `rnorm()` function generates a vector of random normal variables, with first argument `n` the sample size. Each time we call this function, we will get a different answer. Here we create two correlated sets of numbers, `x` and `y`, and use the `cor()` function to compute the correlation between them.

```
x=rnorm(50)
y=x+rnorm(50,mean=50,sd=5)
cor(x,y)
```

By default, `rnorm()` creates standard normal random variables with a mean of 0 and a standard deviation of 1. However, the mean and standard deviation can be altered using the `mean` and `sd` arguments, as illustrated above. Sometimes we want our code to reproduce the exact same set of random numbers; we can use the `set.seed()` function to do this. The `set.seed()` function takes an (arbitrary) integer argument.

```
set.seed(13)
rnorm(50)
```

We'll use `set.seed()` throughout the labs whenever we perform calculations involving random quantities. In general this should allow each of you to reproduce the results from the book. Depending on your version of `R` it is possible that some small discrepancies may appear between the book and your output from `R`.

The `mean()` and `var()` functions can be used to compute the mean and variance of a vector of numbers. Applying `sqrt()` to the output of `var()` will give the standard deviation. Or we can simply use the `sd()` function.

```
set.seed(3)
y=rnorm(100)
y[0:10]

# Compute the mean
mean(y)
```

```
# Compute the variance
var(y)

# Standard deviation = square root of variance
sqrt(var(y))

# But there's a function for that
sd(y)
```

Graphics

We will now create some more sophisticated plots. The `contour()` function produces a contour plot in order to represent three-dimensional data; it is like a topographical map. It takes three arguments:

1. A vector of the `x` values (the first dimension),
2. A vector of the `y` values (the second dimension), and
3. A `matrix` whose elements correspond to the `z` value (the third dimension) for each pair of `(x,y)` coordinates.

As with the `plot()` function, there are many other inputs that can be used to fine-tune the output of the `contour()` function. To learn more about these, take a look at the help file by typing `?contour`.

```
sort(x)
```

```
x <- seq(-3, 3, by = 0.1)
y <- x
z <- outer(x, y, function(xi, yj) xi^2+yj^2)
persp(x, y, z)
```

```
y=x
f=outer(x,y,function(x,y)cos(y)/(1+x^2))
z <- outer(x, y, function(xi, yj) xi^2+yj^2)
persp(x, y, z)
contour(x,y,f)
contour(x,y,f,nlevels=45,add=T)
fa=(f-t(f))/2
contour(x,y,fa,nlevels=15)
```

The `image()` function works the same way as `contour()`, except that it produces a color-coded plot whose colors depend on the `z` value. This is known as a heatmap, and is sometimes used to plot temperature in weather forecasts. Alternatively, `persp()` can be used to produce a three-dimensional plot. The arguments `theta` and `phi` control the angles at which the plot is viewed.

```
image(x,y,fa)
persp(x,y,fa,theta=30,phi=40)
```

Indexing Data

We often wish to examine part of a set of data. Suppose that our data is stored in the matrix `A`.

```
A=matrix(1:16,4,4)
A
```

Then, typing:

```
A[2,3]
```

will select the element corresponding to the second row and the third column. The first number after the open-bracket symbol `[` always refers to the row, and the second number always refers to the column. We can also select multiple rows and columns at a time, by providing vectors as the indices.

```
A[c(1,3),c(2,4)]
A[1:3,2:4]
A[1:2,]
A[,1:2]
```

The last two examples include either no index for the columns or no index for the rows. These indicate that `R` should include all columns or all rows, respectively. `R` treats a single row or column of a matrix as a vector.

```
A[,1]
```

The use of a negative sign `-` in the index tells R to keep all rows or columns *except* those indicated in the index.

```
A[-c(1,3),]
A[-c(1,3),-c(1,3,4)]
```

The `dim()` function outputs the dimensions of a matrix, written as the number of rows followed by the number of columns.

```
dim(A)
```