BIOS 521 Lab #0: R Tutorial (EPID)

Matt Zawistowski

This tutorial provides an introduction to the R programming software, including basic commands to read in your data file and begin an initial analysis of the data. The tutorial is designed to help you become more familiar with basic commands in R. You will start by simply copy-pasting code to the command line. At the end of the tutorial, you will write your own code by making simple changes to previous examples.

Installation of R/RStudio You first need to install R and RStudio on your personal computer. Go to https://cran.r-project.org/ and choose the version of R appropriate for your operating system (e.g. Windows, Mac). Follow the directions to download and install the most recent version of R. Next you will want to install RStudio, an optional R workspace which provides nice visual features that are particularly helpful for new users. RStudio can be downloaded at: https://rstudio.com/products/rstudio/download/. Click on the (free) RStudio Desktop version and then choose the appropriate version for your operating system. This website provides additional details on installation: https://rstudio-education.github.io/hopr/starting.html.

Once you have installed both programs, launch RStudio and you are ready to begin coding.

Installing tidyverse The first step is to install the `tidyverse` package which contains nearly all functions needed to complete the labs this semester. R packages contain code for functions that are not included with basic R installation. Packages only need to be installed one time and then simply be re-loaded each time you begin a new R session. To download and install the `tidyverse` package, copy and paste the following code into the R console command line. You will need to choose a mirror for download. Just choose anything in the US.

```
install.packages("tidyverse")
```

In the previous step, you downloaded and installed R code to your local computer. Now you need to actually run that code in the current R session using the `library` function. You will need to run this command every time you start a new R session.

```
library(tidyverse)
```

Loading Data Next we will load the data file for the *EPID* project into the current R session. First you need to tell R where the file is located. It would be a good idea to create a specific directory on your local computer where you store the dataset, corresponding code and result files. You can set the current working directory using the function `setwd` and passing the path of the directory. Once the working directory is set, R will now look for files in this directory and print output to this directory. You will need to replace the text `PATH OF FILE` with the actual path where you have placed the data file (the path needs to be in double quotes).

```
# Set the working directory containing the dataset
setwd("PATH OF FILE")
```

NOTE: The pound sign (now known as a hashtag) is the comment character in the R language, meaning the line is ignored by the computer and simply there to help humans reading the code. In the code above, you can see that I added a comment to remind myself what the following line of code does. Adding comments to your code makes it much more interpretable both to others and to yourself. It can be very frustrating to return to code you wrote in the past and try to remember what exactly it does. Use comments as a favor to your future self!

The dataset is in csv format (columns separated by commas) so we will use the `read_csv` function to read it into the R session. The dataset is read in and stored as an object that you can name. You can call it `DATA`, or `epid_data` or `Fred`. It does not matter what you name it, you just need to be consistent throughout your code. The `read_csv` function actually stores the data in something called a *tibble* which has some nice properties for data manipulation that we will use later. For now, you don't need to worry about that. I have included the option `na="NA"` to tell R that missing data in this file is coded with "NA". Other common options for missing data are `-9` or simply a blank entry.

```
# Read in the dataset using read_csv.  Note that my data file is stored in a
# subfolder called 'Data' off of my working directory
DATA = read_csv("Data/EPID_521_lab_data.csv", na = "NA")
```

```
## Rows: 2033 Columns: 13
```

```
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (5): RIAGENDR, RIDRETH1, DMDEDUC2, DBQ700, DIQ010
## dbl (8): SEQN, LBXHCY, LBXCOT, RIDAGEYR, BMXBMI, BPXSY1, LBXTC, WkAlcDays
```

```
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

When the data is read into R using the `read_csv` function, it will print to the screen the list of variables that were stored and their respective data type (e.g. `chr` indicates categorical/character variables, and `dbl` indicates variables stored as a number).

If you are unsure about how to use a function or what the potential options are, you can always learn more about that function and even see example usage by typing a question mark (`?`) followed by the function name. For example: `?read_csv`.

Working with Data Once the data has been loaded, the first thing you probably want to do is take a peek to ensure it looks proper. To do that, use the `view` function. This will open a separate window in RStudio that allows you to scroll through the dataset.

```
# Take a peek at your data
view(DATA)
```

You should notice that that there are many `NA` values present in the data, these indicate missing measurements for an observation.

Now that you have peeked at the dataset, let's learn some functions for working with data. First, let's see how many observations and variables are contained in the dataset. The `dim` function will return two numbers: the first is the number of rows (samples/observations) and the second is the number of columns (variables).

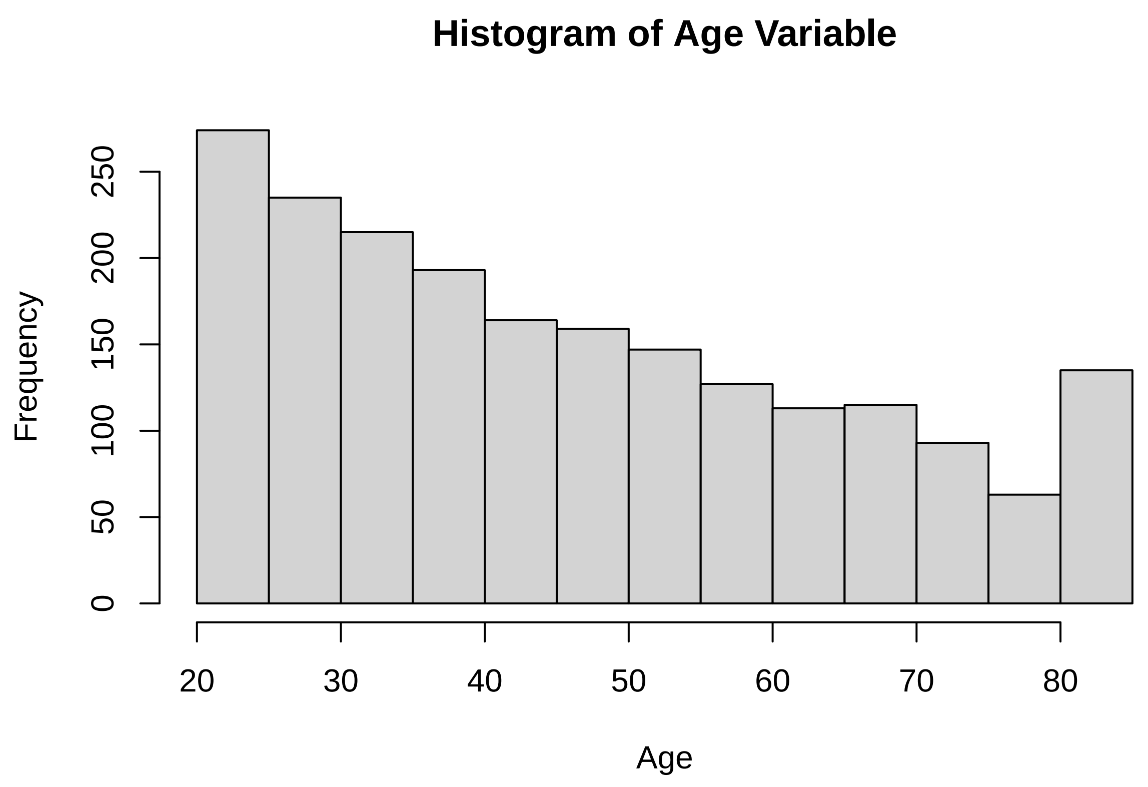
```
dim(DATA)
```

```
## [1] 2033   13
```

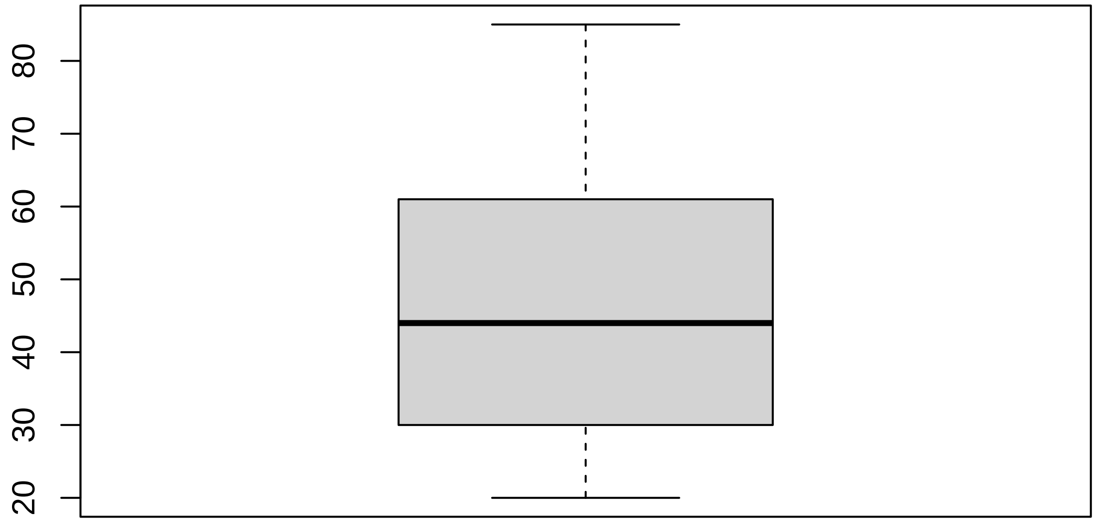
In this case there are 2033 samples and 13 variables.

Numerical Variable Next we'll look deeper into the variables in the dataset by performing an exploratory analysis consisting of graphical plots and computing descriptive statistics. Let's start with the Age variable (called RIDAGEYR in the datafile). Since Age is numeric, either a histogram or boxplot are good ways to visualize the distribution.

```
# Plot the Age variable using a histogram, use the main and xlab options to
# provide names for the figure and the x-axis, respectively
hist(DATA$RIDAGEYR, main = "Histogram of Age Variable", xlab = "Age")
```



```r
# Plot the Age variable using a Boxplot
boxplot(DATA$RIDAGEYR)
```



In the code above, we specified the variable we wanted to examine using the `$` notation. When R sees `DATA$RIDAGEYR` it knows that you are referring to the `RIDAGEYR` variable in the `DATA` data object.

The distribution of Age can be summarized using descriptive statistics such as the mean and standard deviation, or the Five-Number Summary. The `na.rm=TRUE` option tells R to ignore any missing data when computing the quantity of interest. If there are missing data in the variable and this option is not included, R will return `NA` rather than the mean of all available numbers.

```r
mean(DATA$RIDAGEYR, na.rm = TRUE)
```

```
## [1] 46.71864
```

```r
sd(DATA$RIDAGEYR, na.rm = TRUE)
```

```
## [1] 18.78938
```

```r
summary(DATA$RIDAGEYR)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
##   20.00   30.00   44.00   46.72   61.00   85.00 
```

Categorical Variable Now, let's look at a plots and descriptive statistics for the categorical variable *gender*. Create a table with counts for each level of the categorical variable using the table function:

```
# create a table of counts
table(DATA$RIAGENDR)
```

```
##
## Female   Male
##   1291    742
```

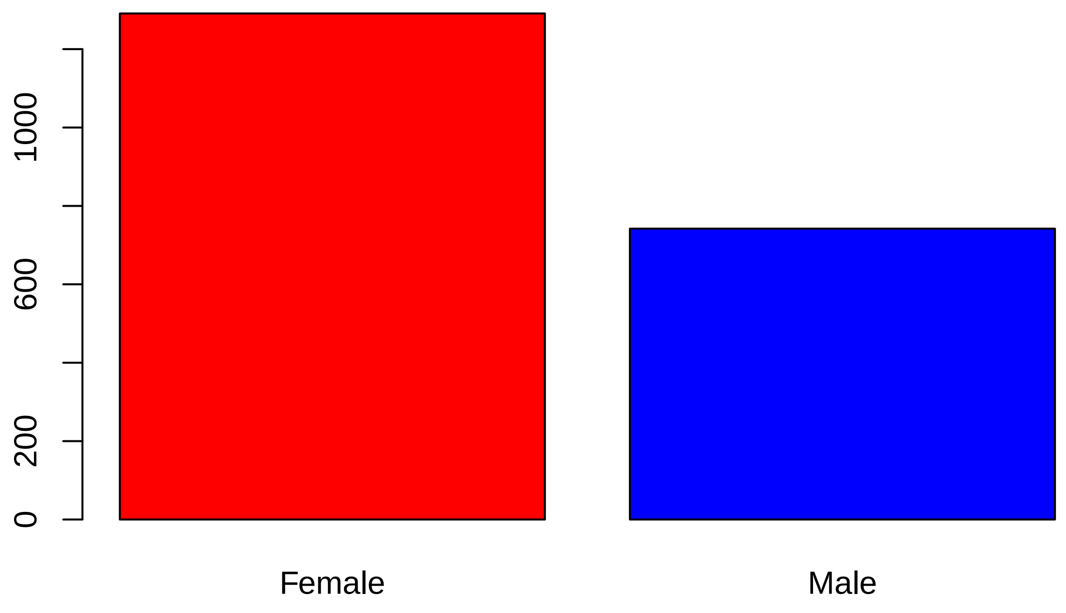
This table now tells us that the *gender* variable has two levels (Female and Male) and gives the corresponding counts in the dataset. If you are interested in the proportions (rather than counts), those can be computed by passing the counts to the `prop.table` command as follows:

```
# create a table of proportions
prop.table(table(DATA$RIAGENDR))
```

```
##
##    Female      Male
## 0.6350221 0.3649779
```

A barchart is used to visualize counts for categorical variable. R provides fine-tune control over graphics, for example the colors for the bars, by passing options to the function. Here the `col=` argument lets you specify the colors of each bar.

```
# create a barplot
barplot(table(DATA$RIAGENDR), col = c("red", "blue"))
```

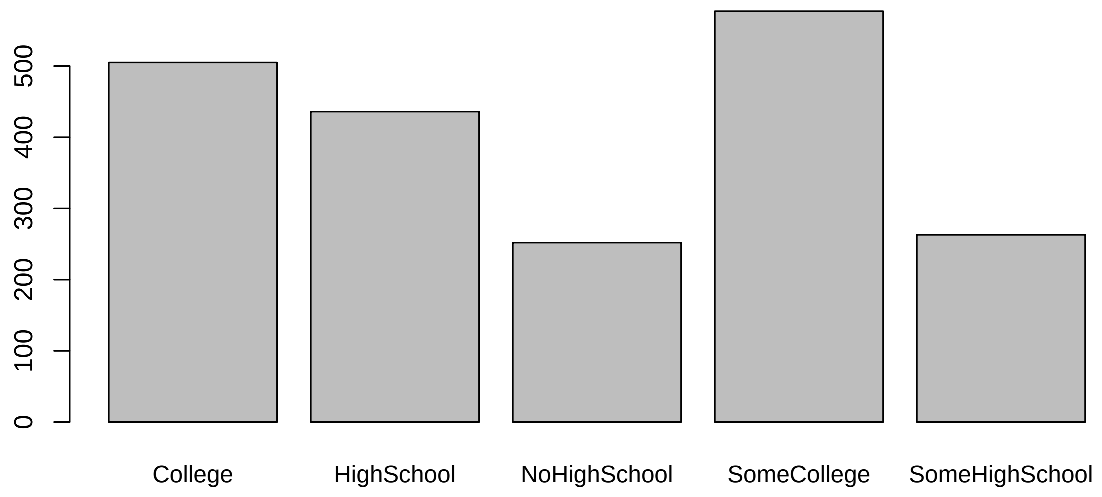


Next, lets look at a different categorical variable, *education*. Make the count table and barplot in the same fashion as we did for the *gender* variable.

```r
table(DATA$DMDEDUC2)
```

```
##
##       College     HighSchool   NoHighSchool    SomeCollege SomeHighSchool
##           505            436            252            577            263
```

```r
barplot(table(DATA$DMDEDUC2), cex.names = 0.9)
```



```r
# The `cex.names` argument in the `barplot` function controls how large to
# print the names of the levels under the bars. I have made them a little
# smaller than the default so they all fit nicely.
```

Something should look off about the histogram. The order of the bars does not make sense. The *education* variable is an *ordinal* categorical variable since there is a natural ordering to different education levels. R does not know this so it simply plotted the levels in alphabetical order in the barplot, making it difficult to see patterns in the plot. We can use the `ordered` function to tell R the correct order for the levels of *education*.

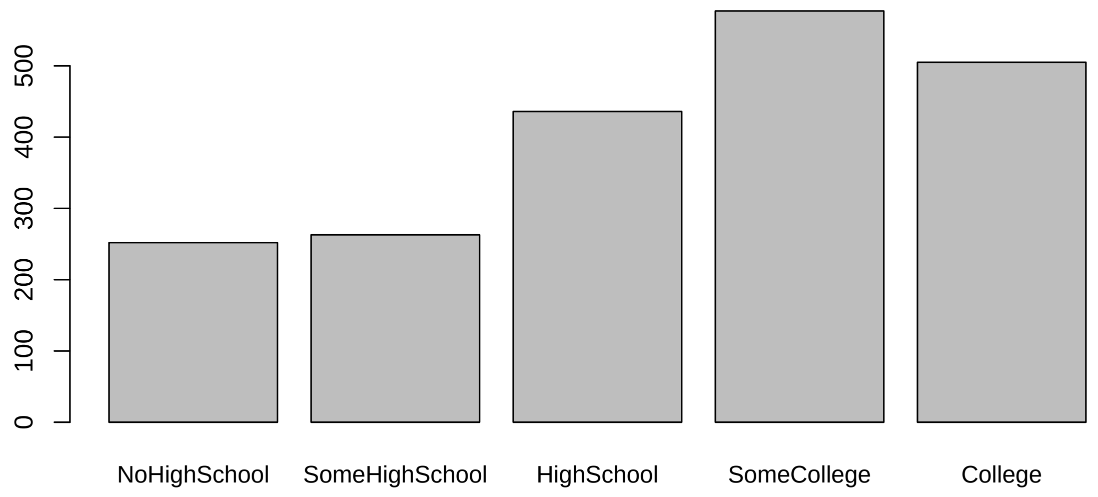
```r
# order the levels of the education variable
DATA$DMDEDUC2 = ordered(DATA$DMDEDUC2, levels = c("NoHighSchool", "SomeHighSchool",
    "HighSchool", "SomeCollege", "College"))
```

Now, re-run the table and barplot commands. Notice that the levels for education are in the natural order. The barplot in particular becomes much easier to interpret.

```
table(DATA$DMDEDUC2)
```

```
##
##   NoHighSchool SomeHighSchool     HighSchool    SomeCollege        College
##            252            263            436            577            505
```

```
barplot(table(DATA$DMDEDUC2), cex.names = 0.9)
```



Basic Data Manipulation We will next look at two basic data manipulation functions. First we will rename a variable. It can be difficult to look at the variable names that come with the dataset, for example *RIDAGEYR*, and remember exactly what it is. Let's just rename it something simpler, like *Age*. To do this we will use the *rename* function. You should feel free to rename variables in your data to something that is more informative. As a bonus, your updated variable names will appear in any plots and tables, making them much more readable.

```
# rename the Age variable
DATA = rename(DATA, Age = RIDAGEYR)
```

Finally, we will create a new variable. We can create a new variable based on existing variables in the dataset using the `mutate` function. Imagine that we would like to indicate samples that are over age 50. We will create a new variable called `Age50` that is `1` if the sample is 50 or over and `0` if the sample is under 50. We will also make use of the `ifelse` function which takes a logical condition (e.g. Age>=50) and returns `1` if that condition is true for a sample and `0` otherwise. (Remember you can use `?ifelse` for additional details on the usage.)

```
# create a new variable Age50 to indicate which samples are aged 50 and above
DATA = mutate(DATA, Age50 = ifelse(Age >= 50, 1, 0))
```

We can see count the number of samples aged 50 and above by creating a table:

```
# count the number of sample 50 and above
table(DATA$Age50)
```

```
##
##    0    1
## 1206  827
```

Lab #0 Questions: Now, it's your turn to try writing R code on your own. Answer the following questions by writing code based on the examples from above. Submit both the code you wrote and the answer for each question.

1. Each dataset contains a variable for self-reported race. Rename that variable to simply *race*. Is there any need to order the levels of the race variable? Why?
2. How many levels are included in the *race* variable? What are the proportions for each group in your dataset?
3. Compute the mean, standard deviation and Five-Number Summary for the BMI variable in your dataset.
4. Based on the descriptive statistics for BMI, do you have any concerns about potential outlier values?
5. Based on the descriptive statistics for BMI, do you think the distribution for BMI is most likely to be left skewed, right skewed or symmetric? Why?
6. Confirm your answer to the above question by creating an appropriate plot to visualize the distribution of BMI.
7. Create a new variable called LogAge containing the natural logarithm of the Age variable. (HINT: The `log` function computes the natural log, `logAge=log(Age)`)
8. What does the distribution of your new variable logAge? (HINT: Create a histogram or boxplot of the variable you created in the previous step.) Compare this to the shape of the original Age variable? That is, how did applying the natural log function change the shape of the distribution of ages in the dataset.
9. Suppose that you are interested in designing a study with individuals aged 65 and above. How many such samples in your dataset? (HINT: create a new variable to identify samples 65+ and use a table to count.)