Introduction to R

Yu Ouyang

QIPSR Fall Software Festival

September 20, 2013

1 Introduction

The goal of this workshop is to introduce you to the programming language/statistical software R and Rstudio, a popular graphical user-interface (GUI) for R users.

Since we'll be using RStudio for this workshop, let's begin by examining what the RStudio GUI looks like when first initialized.

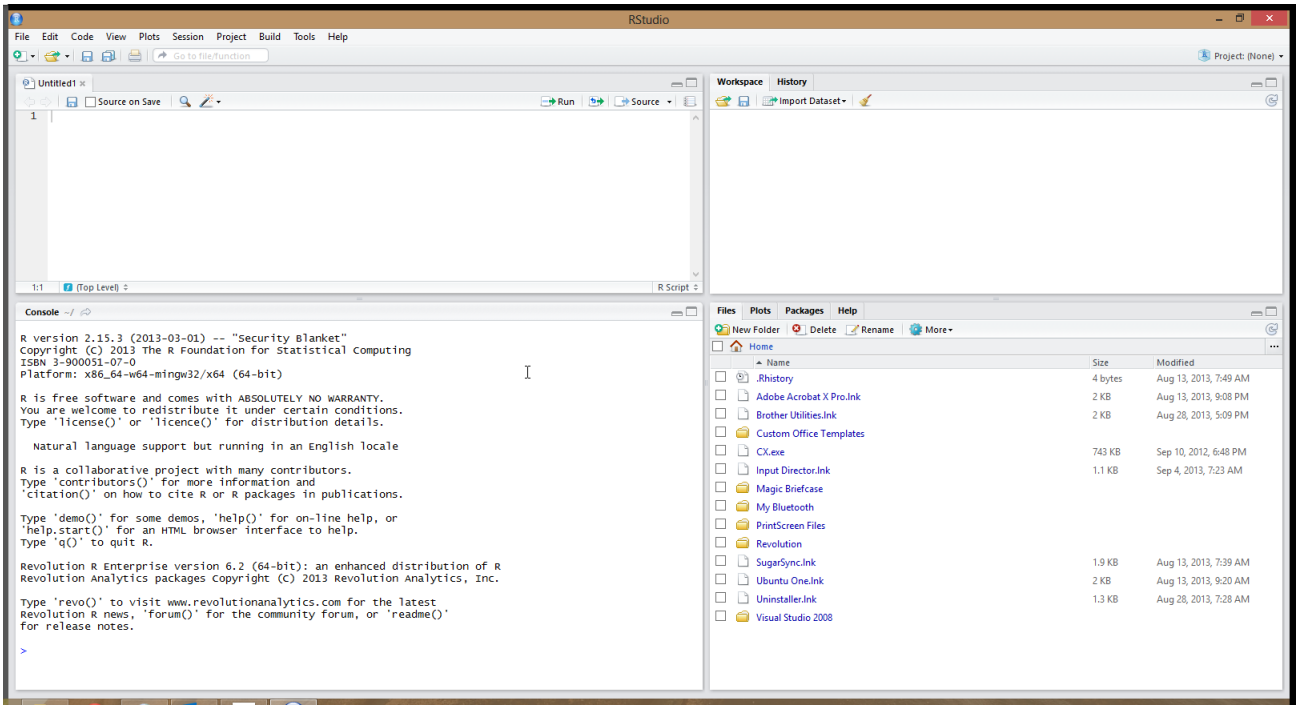


Figure 1: RStudio Interface

Figure 1 shows the four panels that you will be working with in RStudio. The upper left panel contains your R script. Similar to a Do file in Stata or an M-file in MATLAB, an R script allows you to keep a record of your analyses in R, assisting in reproducing your work. The panel in the upper right contains your *workspace*, as well as a history of the commands that you've previously entered. The lower right panel is a multi-purpose window that allows you: (1) to view all files in your current working directory; (2) to view all plots generated thus far in your analyses; (3) to see what packages are currently loaded in R; and (4) to access the help menu. The lower left panel is where the action happens. It's called a *console*. Everything you launch RStudio, it will have the same text at the top of the console telling you the version of R that you're running, as well as some basic R commands. Below that information is the *prompt*. Designated by the > symbol, the prompt is a request for command; this is where you can type in some command, hit the enter button, and immediately see some results. Since we generally want to keep track of our analyses, we will work mostly with writing our commands in the R script and then transfer them to the console.

2 Clear Workspace

By default, RStudio will save everything you generated to the workspace from the previous session. What this means is that you may want to clear the workspace for the current project. **NOTE**: Make sure this is what you want to do since the following command will delete everything in the workspace.

For Stata users, the following command would be the equivalent of the `clear all` command.

```
rm(list=ls()) # Delete everything in workspace
```

3 Set Working Directory

Similar to many other statistical packages, R allows you to set working directory. By setting the working directory in R to a given folder on your computer, you are directing to look in that folder for files, datasets, etc. The R command to set working directory is `setwd()`.

```
setwd("/home/yuouyang/Ubuntu One/Teaching/Misc/Introduction to R Workshop")
```

4 Arithmetic

What can you do using R? One basic way that you can use R is as a calculator, albeit a pretty powerful one. Let's do a couple of examples using addition, subtraction, multiplication, and division.

4.1 Addition

```
2 + 2 # two plus two
```

```
## [1] 4
```

As expected, R returns the value 4 as the result of 2 + 2.

Here is an opportunity to demonstrate a feature in R. The # symbol is what you would use to include an inline comment in R. What it does is that R will ignore everything to the right of the # symbol.

4.2 Subtraction, Multiplication, and Division

```
3 - 1 # three minus one
```

```
## [1] 2
```

```
3 * 4 # three multiplied by 4
```

```
## [1] 12
```

```
10 / 2 # ten divided by two
```

```
## [1] 5
```

4.3 Order of Operation

The order of operation is as you learned in school. R will first do everything within the parentheses first, then exponentiation, then muliplication and division, and last addition and subtraction.

```
3 + 2 * 5 # multiply first
```

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

```
(3 + 2) * 5 # add first
```

```
## [1] 25
```

5 Objects and Data Types

If you read descriptions of R before, you will probably seen it referred to as an "object-oriented language." Objects are things, like a vector of numbers, a word, a dataset, etc. In this section, we wil use objects to illustrate the different data types in R.

Numeric: Numeric is simply numbers. It encompasses both integers and floating-point numbers. You can do math with these types of objects.

Let's create an object called two and assign to it the value 2, which is an integer.

```
two = 2 #Assign value 2 to the object named two
two
```

```
## [1] 2
```

Here's is another opportunity to illustrate a feature in R. The equal sign, =, symbol is known as the assignment operator. It assigns what is on the right-hand side of the = sign to what is on the left-hand side of the = sign. The equal sign can be used interchangably with <-, the more traditional assignment operator. In the example above, we assigned the value 2 to the object named "two." To view the object, simply type the name of the object.

We can do the same with floating-point numbers and even scientific notations:

```
two.five = 2.5 #floating-point number halfway between 2 and 3
two.five
```

```
## [1] 2.5
```

```
twoThousands = 2e3 # the scientific notation for 2 * 10^3 (or two thousands)
twoThousands
```

```
## [1] 2000
```

String: Strings are simply a collection of characters or words that begins and ends with quotes (""). Used mostly for graphs, labels, and other descriptive texts.

```
string = "This is a bunch of words"  # string object
string
```

```
## [1] "This is a bunch of words"
```

Logical: The Boolean values true or false, used mostly for programming logic.

```
t = TRUE
t
```

```
## [1] TRUE
```

```
f = FALSE
f
```

```
## [1] FALSE
```

Factor: This is a special data type for storing categorical data. It combines the integer and character types; we will talk more about this later.

NA: NA is the "missing data" placeholder in R.

```
missing = NA
```

5.1 Mutability

Objects in R are mutable. In other words, objects can be changed or replace. For example,

```
a = 2  #Assign 2 to a
a  #Object a equals 2
```

```
## [1] 2
```

```
a = a + 3  #Replace object a with a + 3
a  #Object a now equals 5
```

```
## [1] 5
```

Note the sequence of what we just did. Initially, the object `a` was assigned a value of 2. Next, we replaced `a` with `a` plus 3. Now, the object `a` is equal to 5.

6 Functions

Most R tutorials will use the word function, instead of command. Functions are what we use to carry out analysis on the data. For example, we would use the `sqrt()` function to find the square root of some value or some object. `sqrt(4)` will give us the square root of 4, which is 2. I will use the words "function" and "command" interchangably throughout.

7 Help Menu

R has a very extensive built-in help and documentation system. To access the help file for, for example, the `plot` command, enter a question mark followed by the command name.

```
?plot
```

Note that the help file for the `plot` command appears in the lower right panel in RStudio. Most help files will provide a description of what the command does and how it's generally used. For now, the help files will most likely be little use to you as they can be a little intimidating. However, as you becomes more familiar with R, you will find these help files more and more "helpful."

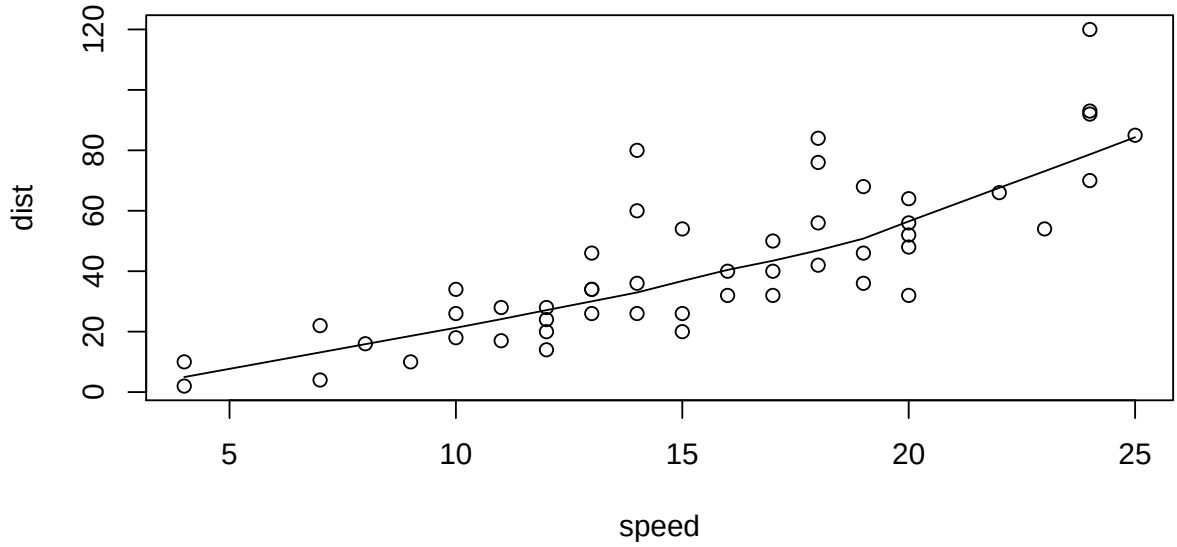
7.1 Alternatives to R's Help Menu

If you're an R novice, and find the help files very "unhelpful" at this point, what should you do? In this case, Google is your friend. Chances are, we can search Google, or any other search engine, for whatever that we wanted to do. For instance, let's say that we want to know how to create a scatterplot in R. We would simply type "create scatterplot in R" into our favorite search engine and find a ton of useful information on how to create a scatterplot in R.

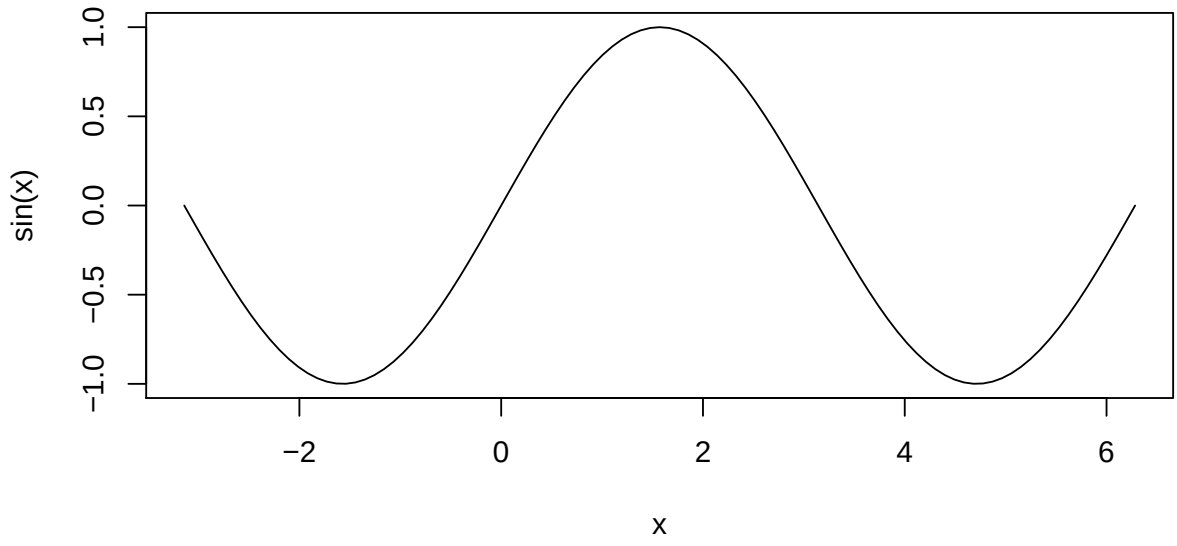
Another method to get help on R commands is to see an example. Most R commands have a series of examples on how to use the command. For example, let's ask R to show us a couple of examples using the `plot` command. We would type:

```
example(plot)
```

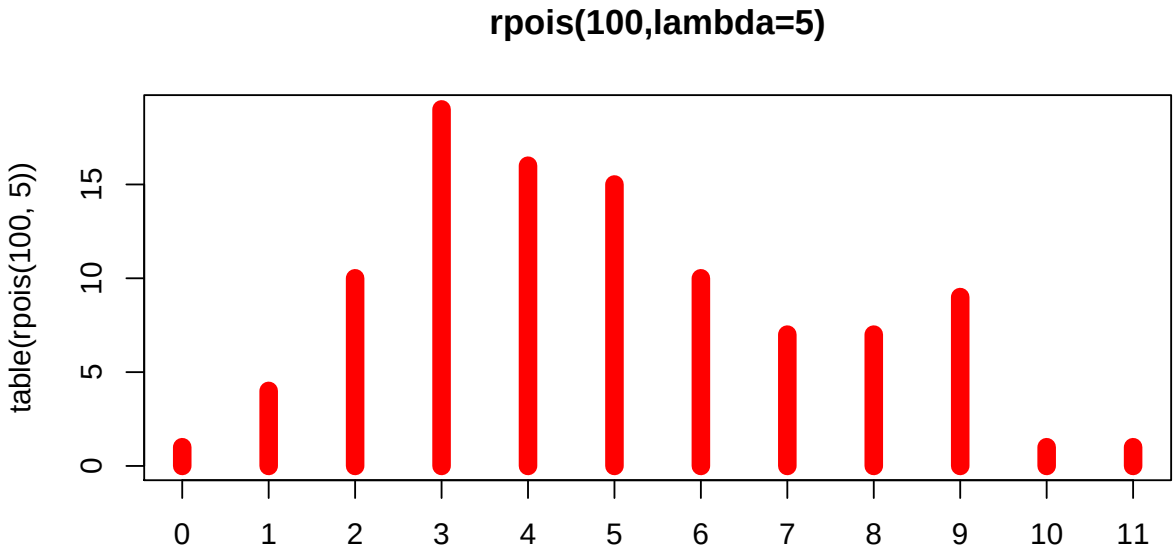
```
##
## plot> require(stats)
##
## plot> plot(cars)
```



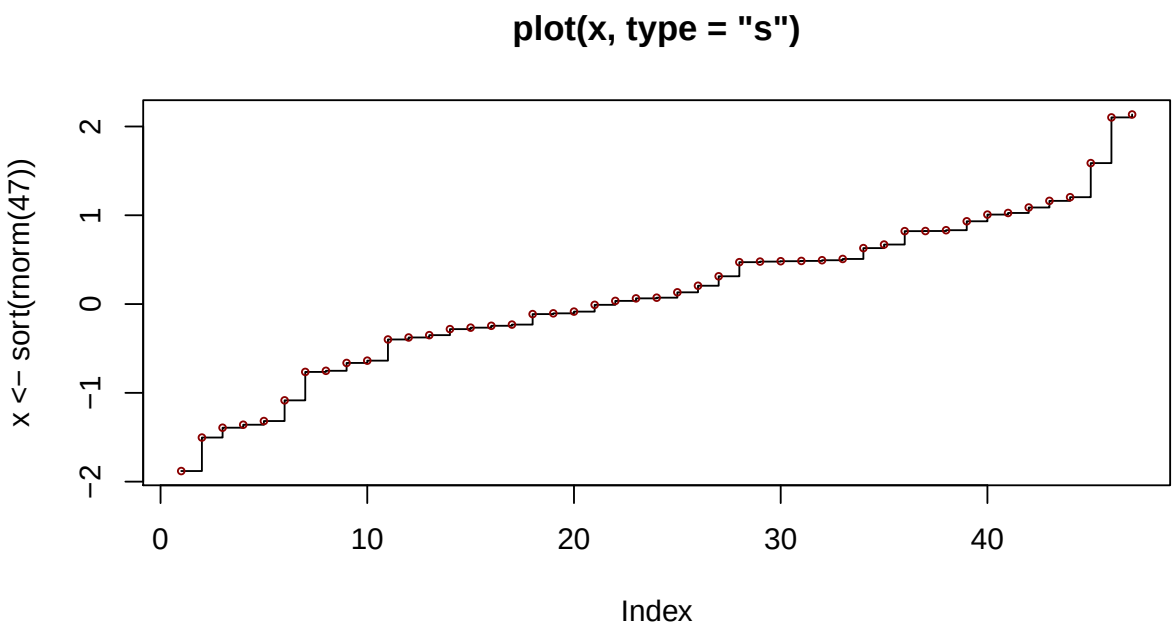

```
##
## plot> lines(lowess(cars))
##
## plot> plot(sin, -pi, 2*pi) # see ?plot.function
```



```
##
## plot> ## Discrete Distribution Plot:
## plot> plot(table(rpois(100,5)), type = "h", col = "red", lwd=10,
## plot+      main="rpois(100,lambda=5)")
```




```
##
## plot> ## Simple quantiles/ECDF, see ecdf() {library(stats)} for a better one:
## plot> plot(x <- sort(rnorm(47)), type = "s", main = "plot(x, type = \"s\")")
```




```
##
## plot> points(x, cex = .5, col = "dark red")
```

At this point, it's not really important to explain what each plot, and the associated commands, is. For now, just know that R can provide examples of how to use a given commands properly if we want it to.

8 Loading Data

Enough description, let's do something more interesting in R. Since we work a dataset most of the time, let's try and load a dataset into R.

8.1 URLs

For datasets that are located on a website, all we need is the URL to the dataset. For example, let's grab a comma-separated dataset (or a CSV file) from the ATS website at UCLA. We will use the `read.csv()` function to load the dataset. We will then use the `head()` command to view the first few rows of the dataset.

```
#Load CSV from website
csv = read.csv("http://www.ats.ucla.edu/stat/data/test.csv", header = TRUE)
head(csv) #View first few rows
```

```
##    prgtype gender  id ses schtyp level
## 1  general      0  70   4      1     1
## 2   vocati      1 121   4      2     1
## 3  general      0  86   4      3     1
## 4   vocati      0 141   4      3     1
## 5 academic      0 172   4      2     1
## 6 academic      0 113   4      2     1
```

Let's review what just happened here. We loaded the CSV file from the website into an object called `csv`, using R's `read.csv()` function. In addition to the URL of the dataset, we also told R that the first row of the CSV file contains the variable names (`header = TRUE`). Putting the object's name, `csv`, into the function `head()` allows us to see the first few rows of the dataset stored in the object called `csv`.

8.2 CSV (on disk)

Let's say that, instead of locating on a website, our CSV file is in our working directory. Since we have already set working directory with the `setwd()` command, we can just load the CSV file in the working directory using the `read.csv` function.

```
#Load CSV from disk using read.csv
csv1 = read.csv("test.csv", header = TRUE)
```

We can verify that the two datasets—one from the website (`csv`) and one from the working directory (`csv1`)—are indentical by using the `head()` command.

```
head(csv)
```

```
##    prgtype gender  id ses schtyp level
## 1  general      0  70   4      1     1
## 2   vocati      1 121   4      2     1
## 3  general      0  86   4      3     1
## 4   vocati      0 141   4      3     1
## 5 academic      0 172   4      2     1
## 6 academic      0 113   4      2     1
```

```
head(csv1)
```

```
##    prgtype gender  id ses schtyp level
## 1  general      0  70   4      1     1
## 2   vocati      1 121   4      2     1
## 3  general      0  86   4      3     1
## 4   vocati      0 141   4      3     1
## 5 academic      0 172   4      2     1
## 6 academic      0 113   4      2     1
```

An alternative way to load the CSV file is to use the `read.table` command. This is to illustrate that there are often multiple ways to do something in R.

```
#Load CSV from disk using read.table
csv2 = read.table("test.csv", header = TRUE, sep = ",")
head(csv2) #View first few rows
```

```
##    prgtype gender  id ses schtyp level
## 1  general      0  70   4      1     1
## 2   vocati      1 121   4      2     1
## 3  general      0  86   4      3     1
## 4   vocati      0 141   4      3     1
## 5 academic      0 172   4      2     1
## 6 academic      0 113   4      2     1
```

As you can see in the example here, in addition to telling R the dataset's name and that the first row contains variable names, we also told R that the dataset is comma-separated, or comma-delimited. Thus, the `read.table()` function is a generic command that allows us to load in files with other delimiters. For instance, we can use this function to load files with: (1) semicolon delimiters; (2) space delimiters; (3) character delimiters, and etc.

8.3 Foreign Datasets

While most people that use R as their primary statistical package will use datasets saved in CSV format, R is also compatible with dataset formats created by other statistical packages. In the example below, we will load the exact same dataset, but now in Stata's .dta format.

To load datasets in formats created by other statistical softwares, we will use the a package in R called `foreign`. Packages are collections of R functions, data, and compiled code in a well-defined format. The `foreign` package is compatible with datasets stored in Minitab, S, SAS, SPSS, Stata, Systat, or dBase format.

```
require(foreign) #Load foreign package
dta = read.dta("test.dta") #Load dataset in Stata's .dta format
head(dta) #View first few rows
```

```
##    prgtype gender  id ses schtyp level
## 1  general      0  70   4      1     1
## 2   vocati      1 121   4      2     1
## 3  general      0  86   4      3     1
## 4   vocati      0 141   4      3     1
## 5 academic      0 172   4      2     1
## 6 academic      0 113   4      2     1
```

Again, we can verify that the Stata .dta dataset is the same as the dataset in CSV.

```
head(dta) #First few rows from .dta dataset
```

```
##    prgtype gender  id ses schtyp level
## 1  general      0  70   4      1     1
## 2   vocati      1 121   4      2     1
## 3  general      0  86   4      3     1
## 4   vocati      0 141   4      3     1
## 5 academic      0 172   4      2     1
## 6 academic      0 113   4      2     1
```

```
head(csv) #First few rows from .csv dataset
```

```
##    prgtype gender  id ses schtyp level
## 1  general      0  70   4      1     1
## 2   vocati      1 121   4      2     1
## 3  general      0  86   4      3     1
## 4   vocati      0 141   4      3     1
## 5 academic      0 172   4      2     1
## 6 academic      0 113   4      2     1
```

9 Data Management

In this section, let's do some basic things with data. We will use the cognitive test scores dataset from Gelman and Hill (2007). [1] First, let's load the dataset and look at the first few rows of the data.

```
require(foreign) #Load foreign package
kidiq = read.dta("kidiq.dta") #Load Stata dataset "kidiq.dta"
head(kidiq)
```

```
##   kid_score mom_hs mom_iq mom_work mom_age
## 1        65      1 121.12        4      27
## 2        98      1  89.36        4      25
## 3        85      1 115.44        4      27
## 4        83      1  99.45        3      25
## 5       115      1  92.75        4      27
## 6        98      0 107.90        1      18
```

9.1 Examine Variables in a Dataset

Since this is a small dataset, we can easily see that there are only five variables. For larger datasets, we can use the `colnames()` function to list all of the variables.

```
colnames(kidiq)
```

```
## [1] "kid_score" "mom_hs"    "mom_iq"    "mom_work"  "mom_age"
```

Using the `colnames()` function, we can see that the five variables are: (1) kid_score, (2) mom_hs, (3) mom_iq, (4) mom_work, and (5) mom_age.

9.2 Dimension of the Dataset

How large is the dataset? To see the size of the dataset, we will use the `dim()` command.

```
dim(kidiq)
```

```
## [1] 434   5
```

The dimension of the dataset is 434, 5. In other words, there are 434 rows and 5 columns in the dataset.

[1] Gelman, Andrew, and Jennifer Hill. 2007. *Data Analysis Using Regression and Multilevel/Hierarchical Models.* New York, NY: Cambridge University Press.

9.3 Rename Variable

If we want to rename a variable in a dataset, an easy way to do this is to use the `rename` function from the `reshape` package.[2] Generically, the structure of the command to rename a variable looks like this:

```
mydata = rename(mydata, c(oldname = "newname")
```

Let's focus on everything to the right of the equal sign first. Our original data is stored in an object called `mydata` and we want to rename a variable from `oldname` to `newname`. Since R objects are mutable, the left side of the equal sign says that we want to replace the "old" `mydata` with the "new" `mydata`.

Now that we know how to use the `rename` command, let's rename the variable `mom_hs` to `mom_HighSchool`

```
colnames(kidiq) #View variable names
```

```
## [1] "kid_score" "mom_hs"    "mom_iq"    "mom_work"  "mom_age"
```

```
require(reshape) #Load reshape package
kidiq = rename(kidiq, c(mom_hs = "mom_HighSchool")) #Rename variable
colnames(kidiq) #View variable names, again
```

```
## [1] "kid_score"      "mom_HighSchool" "mom_iq"         "mom_work"
## [5] "mom_age"
```

[2]The **reshape** package may not be available. You may need to install it. Use the **install.packages()** function. Type **install.packages("reshape")** into the R console.

9.4 Descriptive Stats

Commonly, one of the first things we do with a new dataset is to look at some summary statistics. The `summary()` command is a generic function that works with many types of R objects. Let's try the `summary()` with our dataset (`kidiq`) and see what happens.

```
summary(kidiq) #Summarize dataset object
```

```
##    kid_score      momCompletedHighSchool     mom_iq          mom_work
##  Min.   : 20.0   Min.   :0.000          Min.   : 71.0   Min.   :1.0
##  1st Qu.: 74.0   1st Qu.:1.000          1st Qu.: 88.7   1st Qu.:2.0
##  Median : 90.0   Median :1.000          Median : 97.9   Median :3.0
##  Mean   : 86.8   Mean   :0.786          Mean   :100.0   Mean   :2.9
##  3rd Qu.:102.0   3rd Qu.:1.000          3rd Qu.:110.3   3rd Qu.:4.0
##  Max.   :144.0   Max.   :1.000          Max.   :138.9   Max.   :4.0
##     mom_age
##  Min.   :17.0
##  1st Qu.:21.0
##  Median :23.0
##  Mean   :22.8
##  3rd Qu.:25.0
##  Max.   :29.0
```

As you can see, the above command returns some basic information on each of the five variables. What if we just want the summary statistics on one of the variables, say, `mom_HighSchool`. To select the variable `mom_HighSchool` from the object `kidiq`, we would type `kidiq$mom_HighSchool` in R.

```
summary(kidiq$mom_HighSchool) #Summary Stats on mom_HighSchool
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.000   1.000   1.000   0.786   1.000   1.000
```

Why do we have a dollar sign (`$`) between the name of our data object and the variable name? The dollar sign is called the component selector; it basically extracts a part—component—of an object.

9.4.1 Factor Variables

Here's another opportunity to demonstrate an R feature: factor variables. Let's create a factor variable called `momCompletedHighSchool` based on `mom_HighSchool` and put it into the dataset object `kidiq`. In the following command, the first label, "no," correspond to `mom_HighSchool` = 0 and the second label, "yes," will correspond to `mom_HighSchool` = 1 because the order of the labels will follow the numeric order of the data.

```
#Create a new factor variable based on mom_HighSchool in kidiq
kidiq$momCompletedHighSchool = factor(kidiq$mom_HighSchool, labels = c("no", "yes"))
colnames(kidiq)
```

```
## [1] "kid_score"              "mom_HighSchool"
## [3] "mom_iq"                 "mom_work"
## [5] "mom_age"                "momCompletedHighSchool"
```

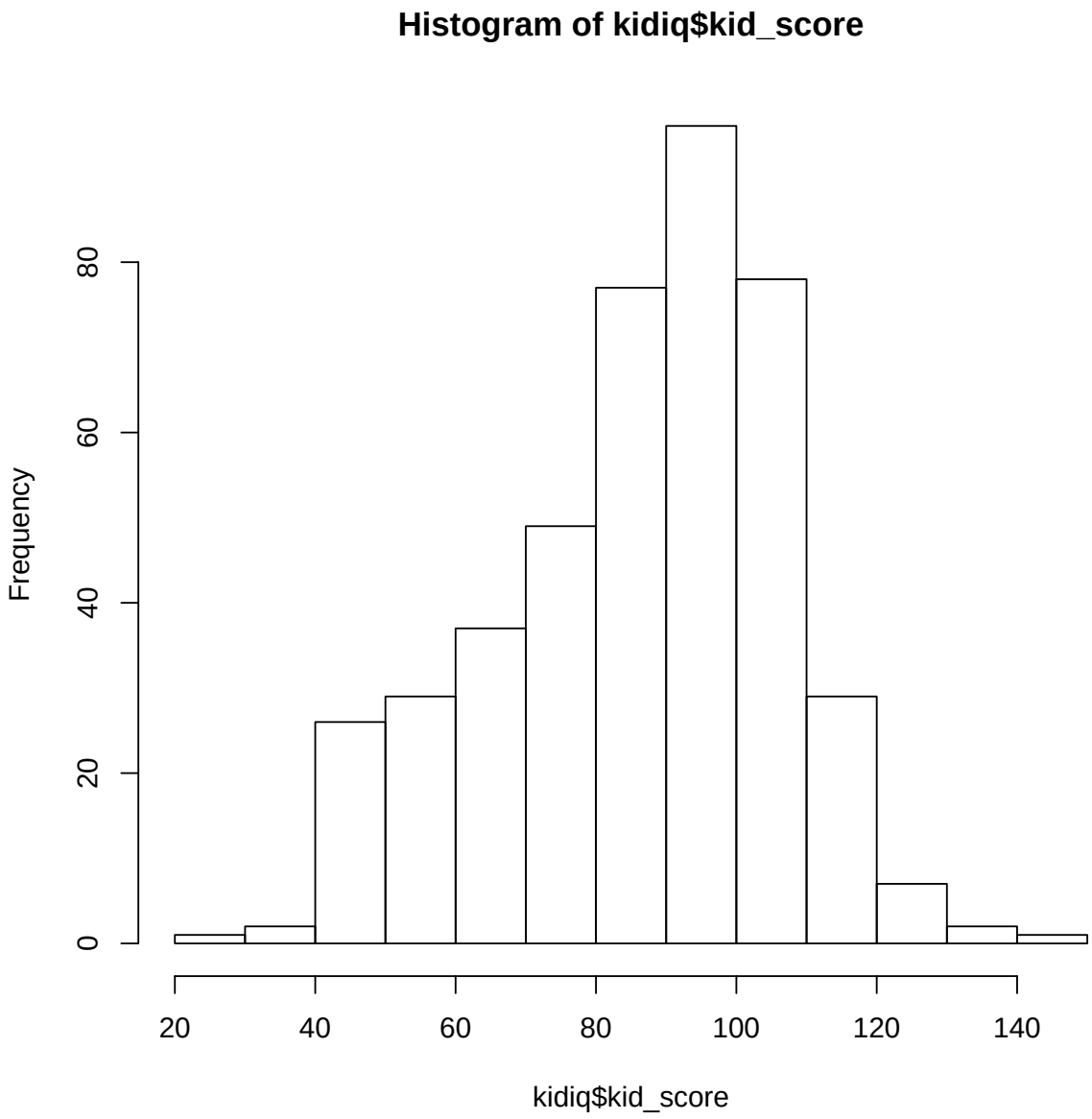
```
head(kidiq)
```

```
##   kid_score mom_HighSchool mom_iq mom_work mom_age momCompletedHighSchool
## 1        65              1 121.12        4      27                    yes
## 2        98              1  89.36        4      25                    yes
## 3        85              1 115.44        4      27                    yes
## 4        83              1  99.45        3      25                    yes
## 5       115              1  92.75        4      27                    yes
## 6        98              0 107.90        1      18                     no
```

10 Basic Graphs

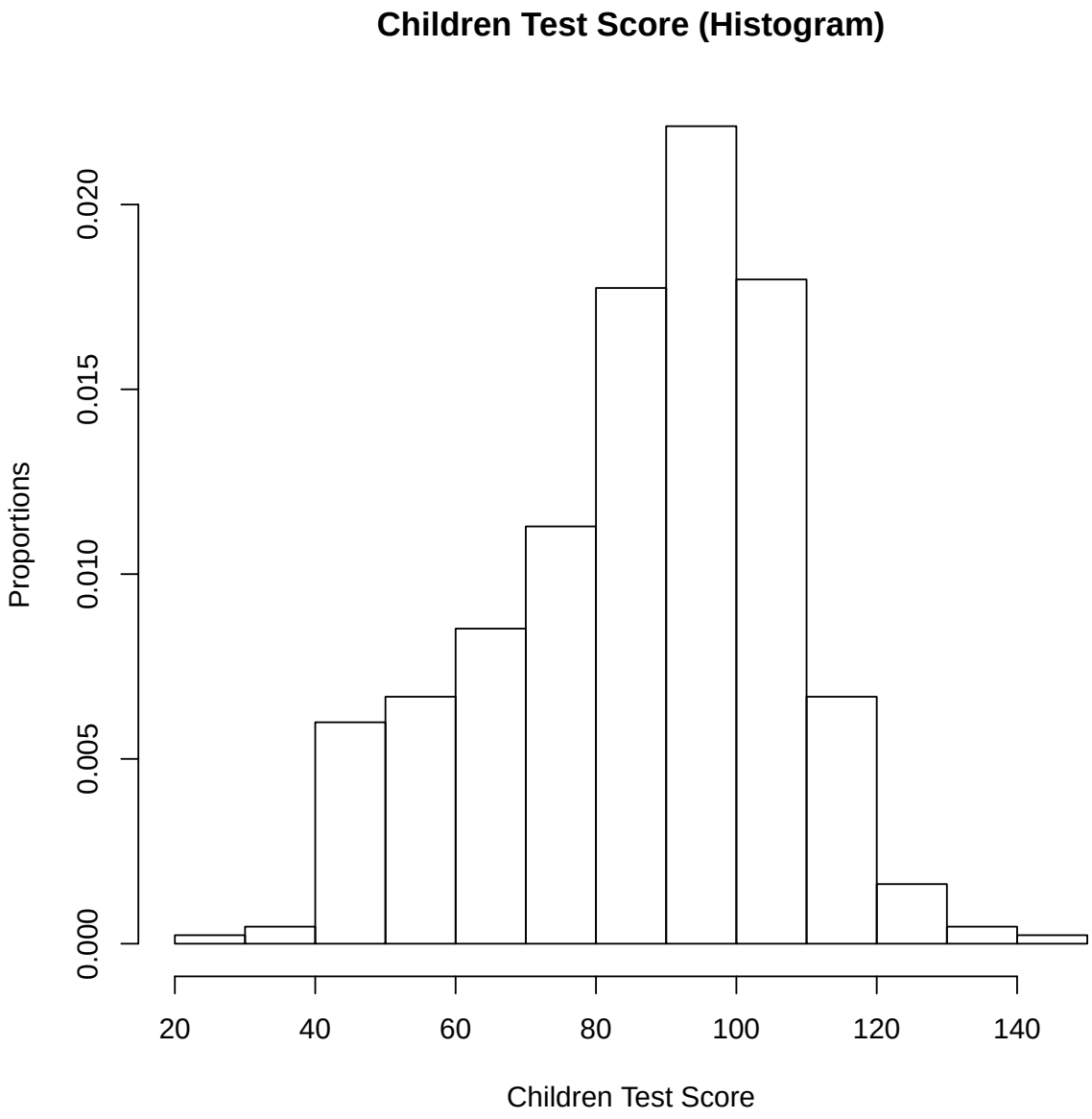
In this section, we will examine a couple of ways to generate basic graphs in R. Using the `kidiq` dataset, we will begin with a histogram. A histogram is very useful for displaying the distribution of a continuous variable. In R, you can create histograms with the `hist()` function, where `x` is a numeric vector of values. Let's create a histogram for the variable `kid_score` from the `kidiq` dataset.

```
hist(kidiq$kid_score) #Histogram of kid_score
```



We can alter the default histogram by adding: (1) a title; (2) a y-axis label; (3) a x-axis label; and change the y-axis to proportions.

```
hist(kidiq$kid_score, #Histogram of kid_score
main = "Children Test Score (Histogram)", #Add Title
ylab = "Proportions", #Add label (y-axis)
xlab = "Children Test Score", #Add label (x-axis)
freq = FALSE #Change y-axis to percentages
)
```

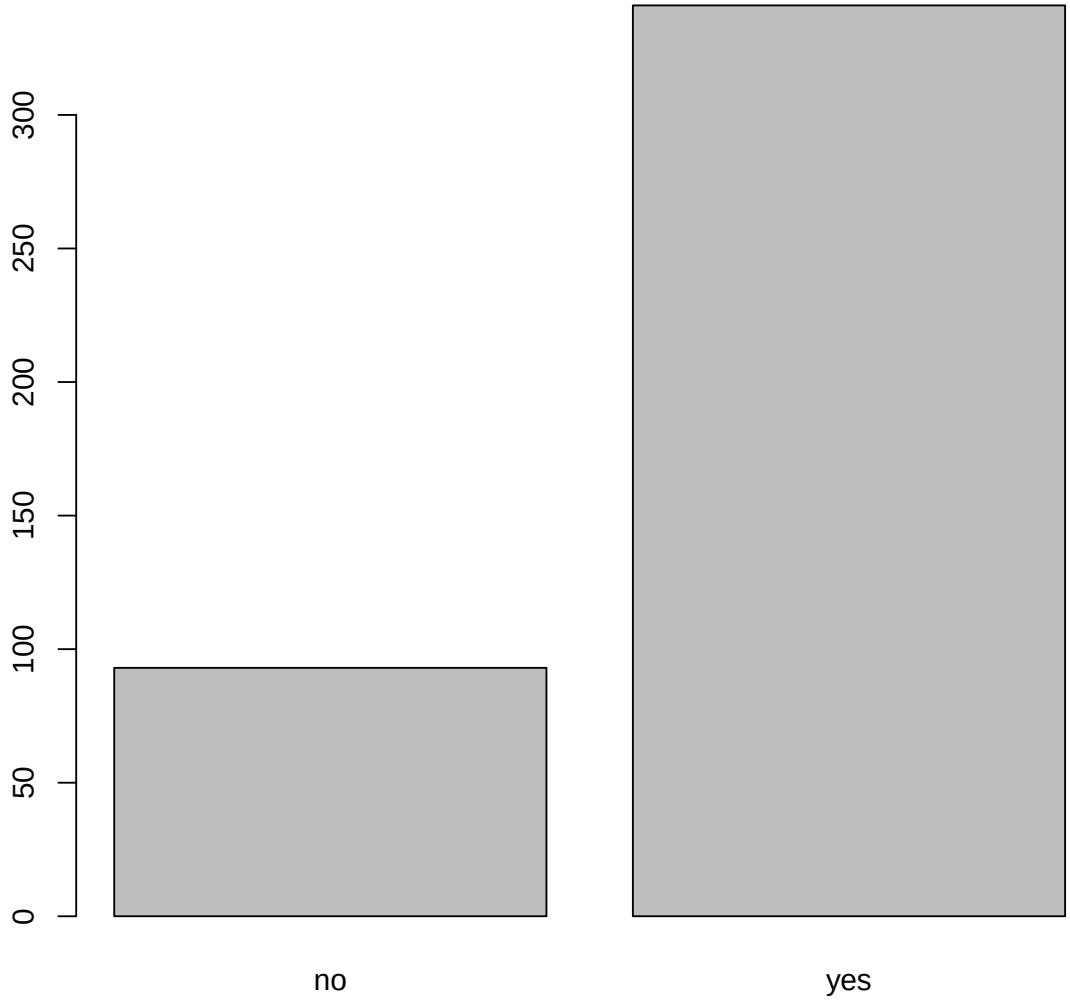


10.1 The plot() function

More generically, R offers the `plot()` function to create a variety of graphs. Depending on the type of data inserted, the `plot()` function may returns different plots.

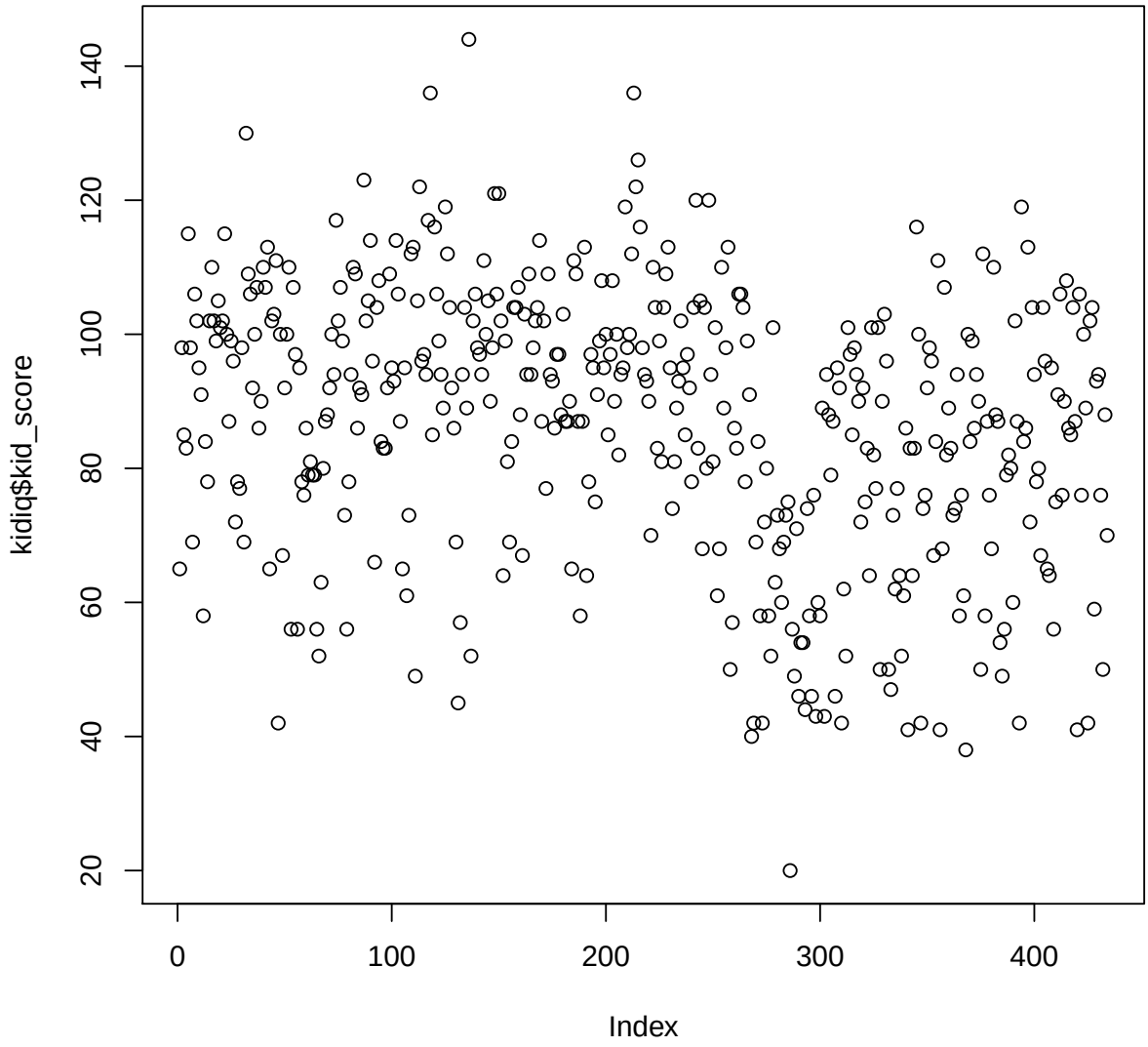
For example, the following command will return a barplot for a categorical variable.

```
plot(kidiq$momCompletedHighSchool)
```



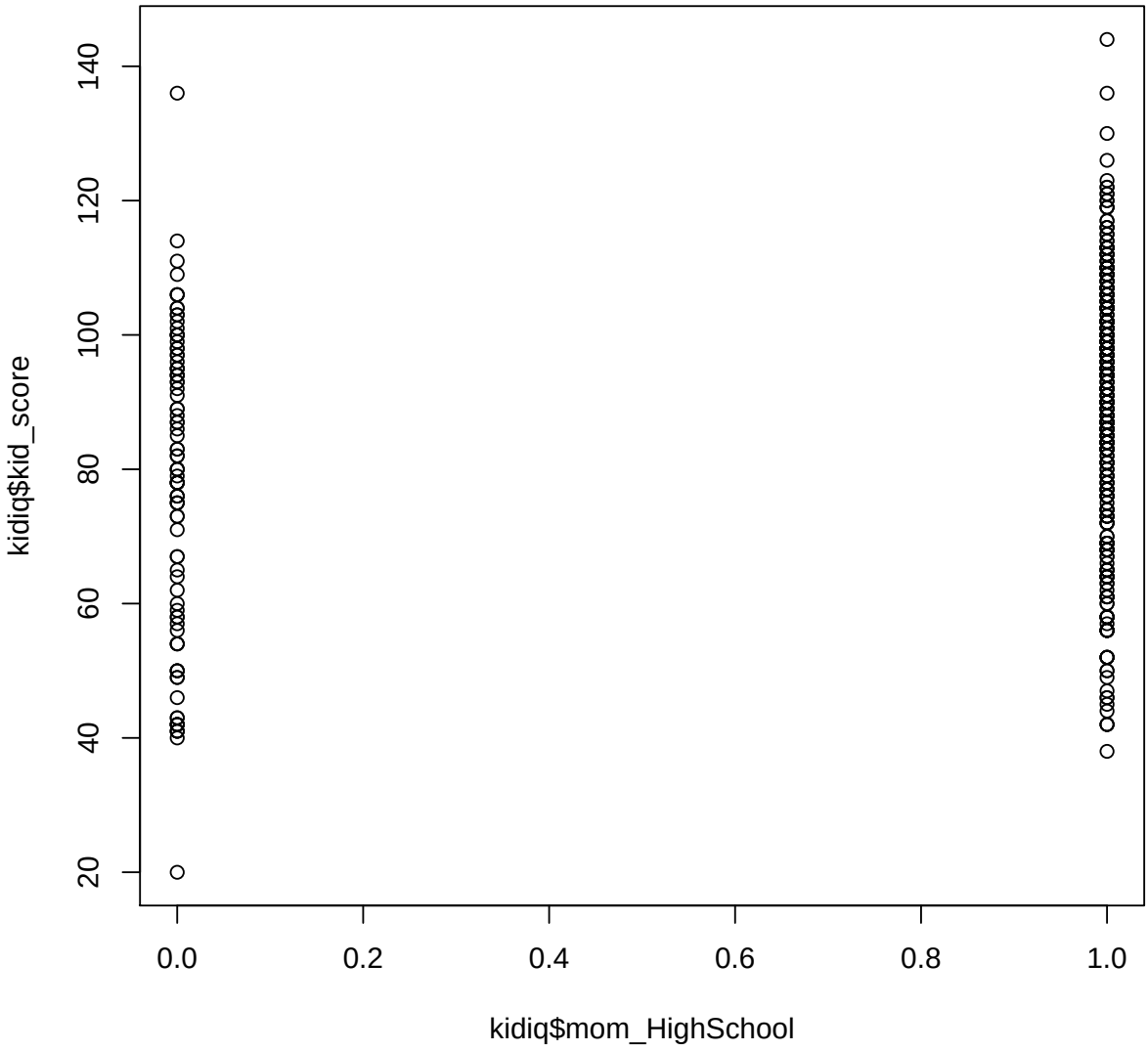
For a continuous variable, the `plot()` function will return a scatterplot, as the following command demonstrates.

```
plot(kidiq$kid_score)
```



With a continuous variable, the `plot()` function also allows us to specify another variable on the x-axis.

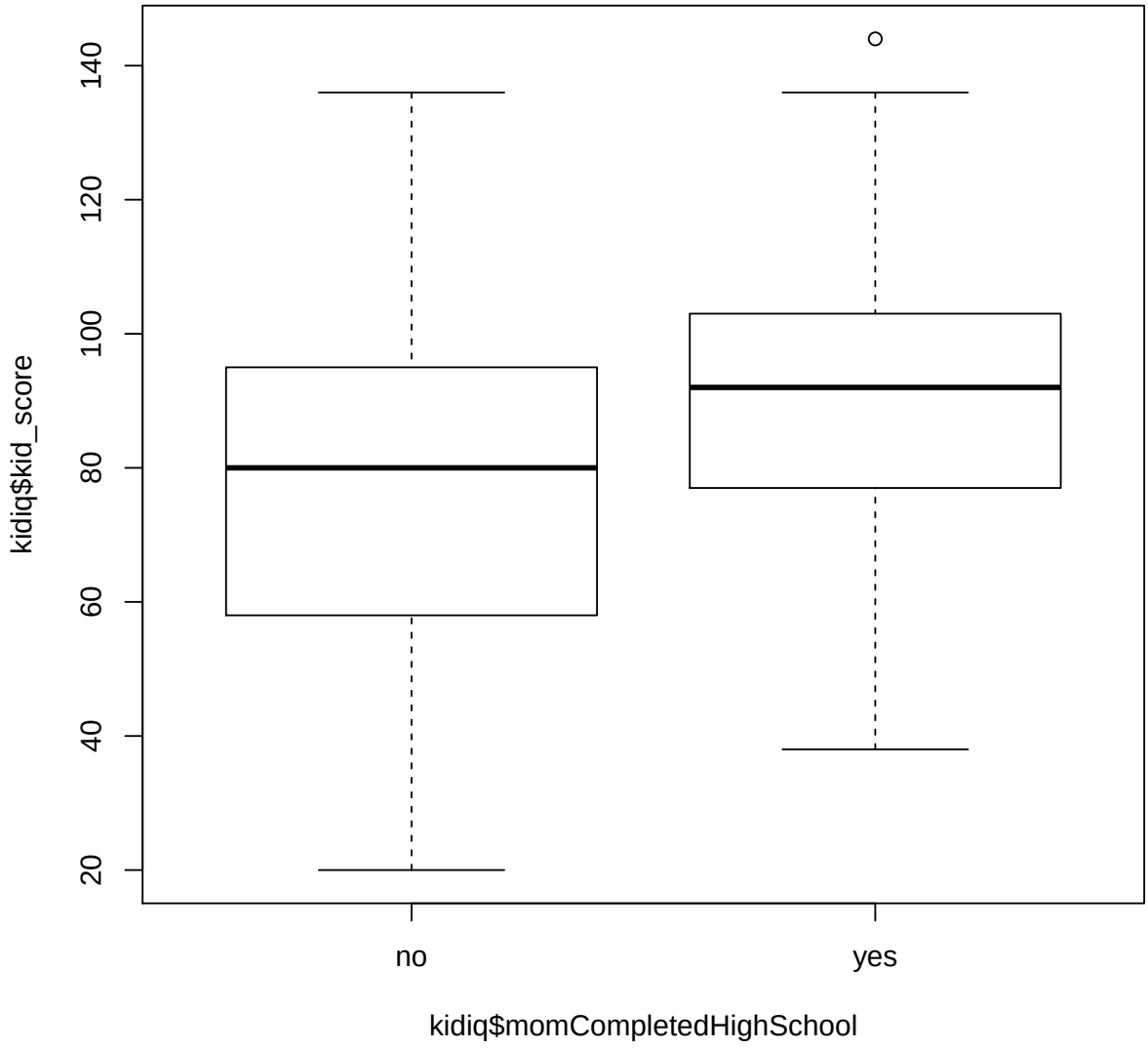
```
plot(kidiq$kid_score ~ kidiq$mom_HighSchool)
```



In the command above, the tilde symbol, `~`, delineates the variable on the y-axis and the variable on the x-axis.

Note what happens when we specify a factor variable on the x-axis.

```
plot(kidiq$kid_score ~ kidiq$momCompletedHighSchool)
```



11 Basic Analysis

In this section, we will use linear regression to illustrate how to conduct analyses in R. Again, we will use the `kidiq` dataset.

11.1 Linear Regression - Single Predictor

Let's fit a linear model with the child's test score (`kid_score`) as the dependent variable and the mother's score on an IQ test—a continuous variable—as the independent variable. We fit the model and view the results using the following commands:

```
single = lm(kidiq$kid_score ~ kidiq$mom_iq) #Fit linear  regresion model
summary(single) #View results of the model
```

```
##
## Call:
## lm(formula = kidiq$kid_score ~ kidiq$mom_iq)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -56.75 -12.07   2.22  11.71  47.69
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)   25.7998     5.9174    4.36  1.6e-05 ***
## kidiq$mom_iq   0.6100     0.0585   10.42  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 18.3 on 432 degrees of freedom
## Multiple R-squared: 0.201,Adjusted R-squared: 0.199
## F-statistic:  109 on 1 and 432 DF,  p-value: <2e-16
```

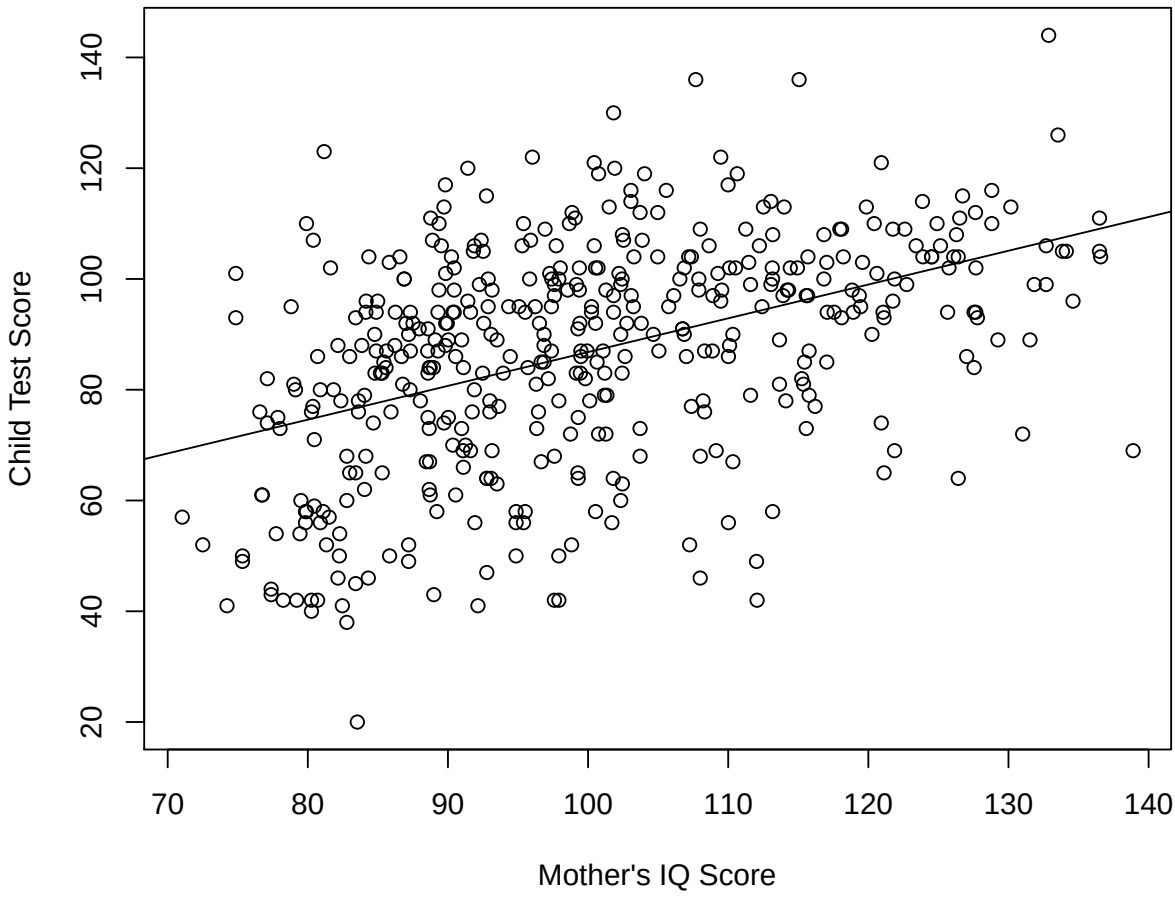
Let's review what just happened. We fitted the following linear model using the `lm()` function and saved the results in the object `single`:

$$\text{kid\_score} = \text{mom\_iq} + \varepsilon$$

We then used the `summary()` command to view the results of the linear model.

Let's now look at the linear model on a plot.

```
plot(kidiq$kid_score ~ kidiq$mom_iq, #Scatterplot
xlab = "Mother's IQ Score", #Add x-label
ylab = "Child Test Score" #Add y-label
)
abline(single) #Add linear fit line
```



The fitted model is:

$$\widehat{\text{kid\_score}} = 25.80 + .61 * \text{mom\_iq} + \varepsilon$$

11.2 Linear Regression - Multiple Predictors

Let's make the previous linear model simply more complicated by adding another independent variable, `mom_HighSchool`.

```
#Fit multiple regression model
multiple = lm(kidiq$kid_score ~ kidiq$mom_iq + kidiq$mom_HighSchool)
summary(multiple) #View results
```

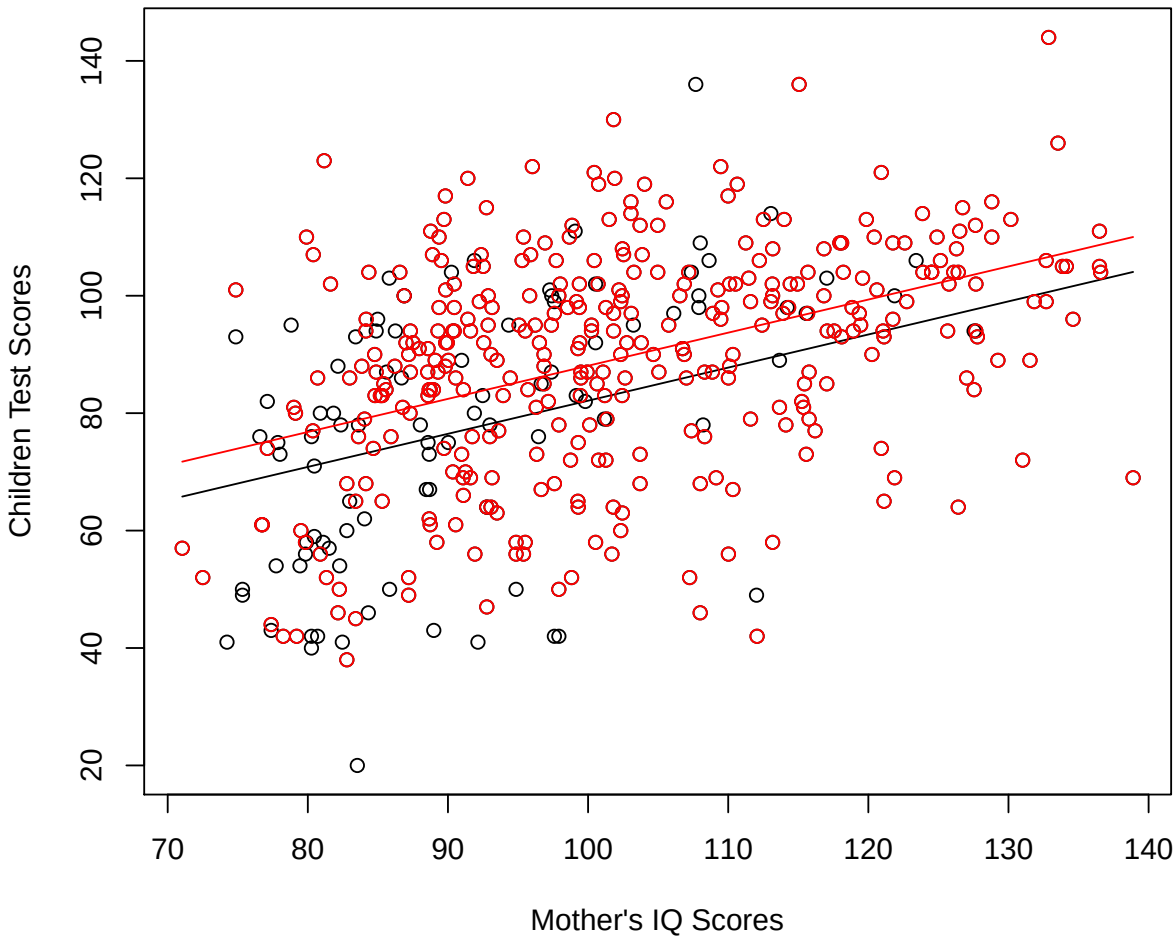
```
##
## Call:
## lm(formula = kidiq$kid_score ~ kidiq$mom_iq + kidiq$mom_HighSchool)
##
## Residuals:
##    Min     1Q Median     3Q    Max
##  -52.9  -12.7    2.4   11.4   49.5
##
## Coefficients:
##                       Estimate Std. Error t value Pr(>|t|)
## (Intercept)            25.7315     5.8752    4.38  1.5e-05 ***
## kidiq$mom_iq            0.5639     0.0606    9.31  < 2e-16 ***
## kidiq$mom_HighSchool    5.9501     2.2118    2.69   0.0074 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 18.1 on 431 degrees of freedom
## Multiple R-squared: 0.214,Adjusted R-squared: 0.21
## F-statistic: 58.7 on 2 and 431 DF,  p-value: <2e-16
```

The fitted model is:

$$\widehat{\text{kid\_score}} = 25.73 + 5.95 * \text{mom\_HighSchool} + 0.56 * \text{mom\_iq} + \varepsilon$$

Let's plot this model.

```
#Fit multiple regression model
multiple = lm(kidiq$kid_score ~ kidiq$mom_HighSchool + kidiq$mom_iq)
plot(kidiq$kid_score ~ kidiq$mom_iq, #Scatterplot
xlab = "Mother's IQ Scores", #Add x-label
ylab = "Children Test Scores", #Add y-label
)
curve(coef(multiple)[1] + coef(multiple)[2] + coef(multiple)[3]*x, #Generate fit line
add=TRUE, col="red")  #Add fit line to previous plot and color it red
curve(coef(multiple)[1] + coef(multiple)[3]*x, add=TRUE)
points(kidiq$mom_iq[kidiq$mom_HighSchool==1],
kidiq$kid_score[kidiq$mom_HighSchool==1], col="red")
```



A ton of stuff just happened to generate the previous graph. Let's break it down piece by piece. First, we fitted the model using the `lm()` function and saved the results to an object called `multiple`. Next, using the `plot()` command, we generated a scatterplot with Children Test Scores (`kid_score`) on the y-axis and Mother's IQ Scores (`mom_iq`) on the x-axis. Of course, we made the scatterplot prettier by adding an x-label and a y-label. Third, using the `curve()` command, we added two linear fit lines, one for those children whose mother completed high school (red line) and one for those children whose mother did not complete high school (black line). The last command, using the `points()` function, colors those observations whose mother completed high school red.

11.3 Linear Model - Interactions

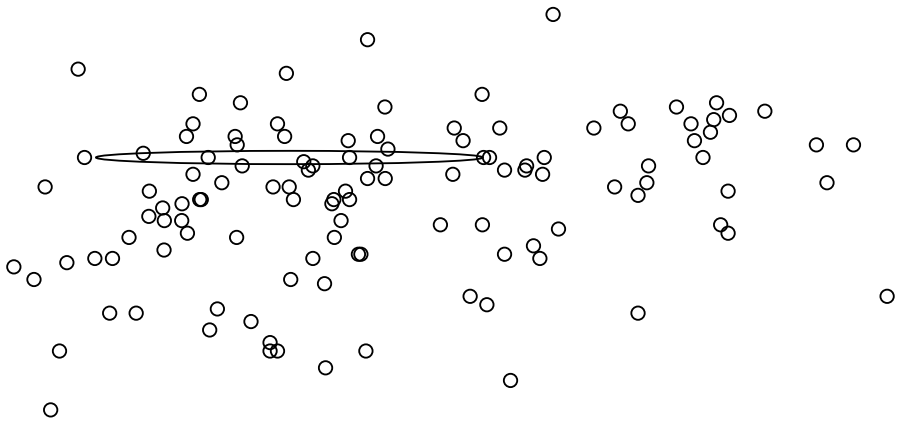
What if we want to add an interaction term between `mom_iq` and `mom_HighSchool` into the previous model?

```
#Fit multiple regression model with interactions
interaction = lm(kidiq$kid_score ~ kidiq$mom_HighSchool + kidiq$mom_iq
+ kidiq$mom_iq*kidiq$mom_HighSchool)
summary(interaction)
```

```
##
## Call:
## lm(formula = kidiq$kid_score ~ kidiq$mom_HighSchool + kidiq$mom_iq +
##     kidiq$mom_iq * kidiq$mom_HighSchool)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -52.09 -11.33   2.07  11.66  43.88
##
## Coefficients:
##                                    Estimate Std. Error t value Pr(>|t|)
## (Intercept)                         -11.482     13.758   -0.83    4e-01
## kidiq$mom_HighSchool                 51.268     15.338    3.34    9e-04 ***
## kidiq$mom_iq                          0.969      0.148    6.53  1.8e-10 ***
## kidiq$mom_HighSchool:kidiq$mom_iq    -0.484      0.162   -2.99    3e-03 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 18 on 430 degrees of freedom
## Multiple R-squared: 0.23,Adjusted R-squared: 0.225
## F-statistic: 42.8 on 3 and 430 DF,  p-value: <2e-16
```

To see how this model looks graphically, we would do the same thing as the previous graph.

```
#Fit multiple regression model with interactions
interaction = lm(kidiq$kid_score ~ kidiq$mom_HighSchool + kidiq$mom_iq
+ kidiq$mom_HighSchool*kidiq$mom_iq)
plot(kidiq$kid_score ~ kidiq$mom_iq,
xlab="Mother IQ score",
ylab="Child test score"
)
curve(coef(interaction)[1] + coef(interaction)[2] + (coef(interaction)[3]
+ coef(interaction)[4])*x,
add=TRUE, col="red")
curve (coef(interaction)[1] + coef(interaction)[3]*x,
add=TRUE)
points(kidiq$mom_iq[kidiq$mom_HighSchool==0],
kidiq$kid_score[kidiq$mom_HighSchool==0])
points(kidiq$mom_iq[kidiq$mom_HighSchool==1],
kidiq$kid_score[kidiq$mom_HighSchool==1], col="red")
```



12 Display Model Results in Caterpillar Plot (Advance)

Many packages that estimate statistical models have built in plotting capabilities. For example, the `survivial` package has a `plot.survfit()` command for plotting survivial curves created using event history analysis.

However, sometimes either a package doesn't have built in commands for plotting model results the way you want to and/or you wish to improve the aesthic quality of the plots they do create by default. In either case, you can almost always create the plot that you want by first breaking into the model results object, extracting what you want, then plotting it with the `ggplot2` package.

To illustrate, let's create a caterpillar plot showing the coefficient estimates and the uncertainty surrounding them. Further, let's fit one of the previous models using a Bayesian normal linear regression framework. **NOTE:** The following commands will require that the `Zelig`, `ggplot2`, `MCMCpack`, and `coda` packages to be already installed in R.

```
#Load Zelig package
require(Zelig)

#Estimate model
bayes.model = zelig(Examination ~ Education + Agriculture +
Catholic + Infant.Mortality,
model = "normal.bayes",
data = swiss)
```

```
##
##
##  How to cite this model in Zelig:
##   Ben Goodrich, and Ying Lu. 2013.
##   "normal.bayes: Bayesian Normal Linear Regression"
##   in Kosuke Imai, Gary King, and Olivia Lau, "Zelig: Everyone's Statistical Software,
##   http://gking.harvard.edu/zelig
##
```

```
#Create summary object
bayes.modelSum = summary(bayes.model)

#Create summary data frame
bayes.modelDF = data.frame(bayes.modelSum$summary)

#Show data frame
bayes.modelDF
```

```
##                         Mean      SD   X2.5.      X50.   X97.5.
## (Intercept)       18.646074 5.92866  7.0305 18.650330 30.42487
## Education          0.424876 0.09097  0.2425  0.425473  0.60040
## Agriculture       -0.067269 0.04251 -0.1515 -0.067515  0.01549
## Catholic          -0.079682 0.01807 -0.1153 -0.079837 -0.04404
## Infant.Mortality -0.007321 0.23554 -0.4688 -0.009075  0.45927
## sigma2            19.895381 4.56803 12.8441 19.254808 30.69691
```

We want to use the `ggplot2` package to create credibility intervals for each variable with `X2.5.` as the minimum value and `X97.5.` as the maximum value. These are the lower and uppoer bounds of the middle 95 percent of the estimates' marginal posterior distributions, i.e. the 95 percent credibility intervals.[3] We will create a point in the `mean` of each estimate. To do this, we will use `ggplot2`'s `geom_pointrange` command.

First, we need to do a little cleaning up.

```
#Convert row.names to normal variable
bayes.modelDF$Variables = row.names(bayes.modelDF)

#Keep only coefficient estimates
## This allows for a more interpretable scale
bayes.modelDF = subset(bayes.modelDF, Variables != "(Intercept)")
bayes.modelDF = subset(bayes.modelDF, Variables != "sigma2")
```

The first line of the executable code creates a proper variable out of the data frame's `row.names` attribute. In this case, `row.names` contains the names of the variables included in the regression. The second and third executable lines remove the estimates `(Intercept)` and `sigma2`. This allows the variable's coefficient estimates to be plotted on a scale that enables easier interpretation.

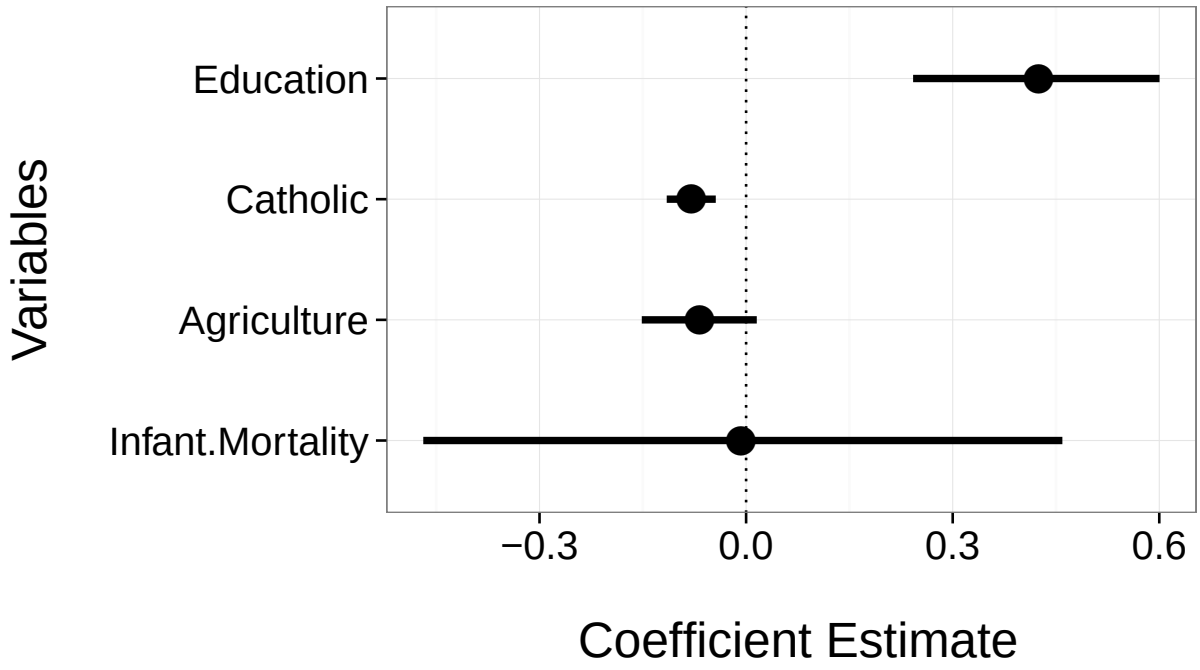
Now, we can create our caterpillar plot.

```
#Load ggplot2 package
require(ggplot2)

#Make caterpillar plot
ggplot(data = bayes.modelDF,
aes(x = reorder(Variables, X2.5.),
y = Mean,
ymin = X2.5., ymax = X97.5.)) +
geom_pointrange(size = 1.4) +
geom_hline(aes(intercept = 0), linetype = "dotted") +
```

[3]The procedures used here are also generally applicable for graphing frequentist confidence intervals once you have calculated the confidence intervals. One useful command for that purpose is **confint**.

```
xlab("Variables\n") +
ylab("\n Coefficient Estimate") +
coord_flip() +
theme_bw(base_size = 20)
```



There are some new pieces of code in here, so let's take a look. First, the data frame is reordered from the highest to lowest value of `X2.5.` using the `reorder` command. This makes the plot easier to read. The middle point of the point range is set with `y` and the lower and upper bounds with `ymin` and `ymax`. The `geom_hline` command used here creates a dotted horizontal line at 0, i.e. no effect. `coord_flip` flips the plot's coordinates so that the variable names are on the y axis.