R: A Brief Introduction

Andrew Q. Philips[*]

August 16, 2016

[*]Ph.D Candidate, Department of Political Science, Texas A&M University, Contact: aphilips@tamu.edu. Copyright 2016. This paper was prepared for the Summer “Bootcamp” Course at Texas A&M University. Please do not redistribute without permission from the author.

Contents

1 Introduction

The purpose of this document to serve as a brief introduction to R, an application that can be used for data management, statistics, graphics, and a lot of other things. R grew out of the S and S-Plus coding languages in the early 1990s. It is quickly growing in popularity and has already surpassed older languages.

One of the strongest advantages of R is that it is *free*. As such, users are very active in creating and maintaining packages, or stand-alone collections of programs to use in R. There are a set of standard base packages, but many users will install additional packages as needed. The R community is also fairly helpful on online forums.

R is very popular in political science, and seems to be becoming more common due to its cost and flexibility. In fact, it is one of the fastest growing languages.[1] Moreover, there are many complex methods that have not been written for Stata or SPSS but exist in R. If you ever replicate the analysis in an article, there is a good chance that it was written and saved as a .R file.

1.1 Assignments and Working Through This Document

This document discusses various topics in R. In addition it shows actual R output. You should open up your own R window and follow along. Be sure to answer all of the **Assignments** scattered throughout the document.

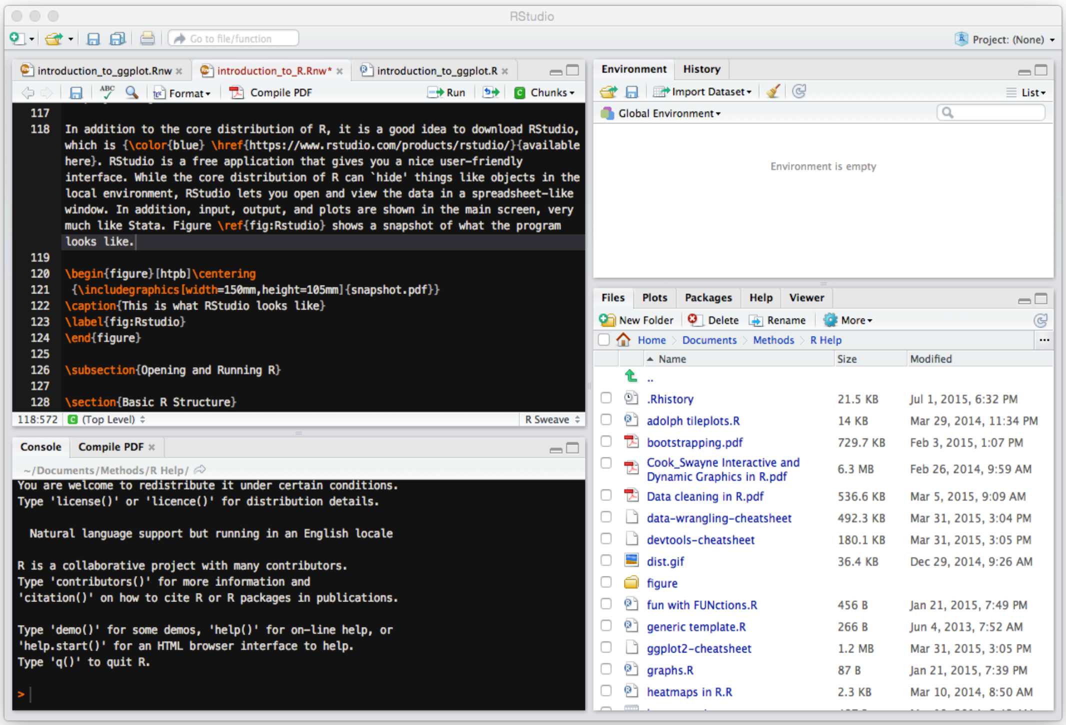
1.2 Obtaining R

The best place to obtain a copy of R for either Mac, Windows, or Linux is on the Comprehensive R Archive Network (CRAN) website, available here. It should also be on the POLS shared drive on your office computers, but this is likely to be a depreciated version. In case R downloads in some obscure place, you probably want to place it in the

[1]In 2015 alone more functions were written for R than in the entire history of SAS: http://r4stats.com/2016/04/19/rs-growth-continues-to-accelerate/.

Applications folder (on Mac) or Program Files (on Windows), or right on the desktop.

In addition to the core distribution of R, it is a good idea to download RStudio, which is available here. RStudio is a free application that gives you a nice user-friendly interface. While the core distribution of R can 'hide' things like objects in the local environment, RStudio lets you open and view the data in a spreadsheet-like window. In addition, input, output, and plots are shown in the main screen, very much like Stata. Current versions even autofill commands as you type. In other words, it substantially decrease the learning curve needed to master R. Figure 1 shows a snapshot of what the program looks like.



```

In addition to the core distribution of R, it is a good idea to download RStudio,
which is {\color{blue} \href{https://www.rstudio.com/products/rstudio/}{available
here}. RStudio is a free application that gives you a nice user-friendly
interface. While the core distribution of R can `hide' things like objects in the
local environment, RStudio lets you open and view the data in a spreadsheet-like
window. In addition, input, output, and plots are shown in the main screen, very
much like Stata. Figure \ref{fig:Rstudio} shows a snapshot of what the program
looks like.

\begin{figure}[htpb]\centering
 {\includegraphics[width=150mm,height=105mm]{snapshot.pdf}}
\caption{This is what RStudio looks like}
\label{fig:Rstudio}
\end{figure}

\subsection{Opening and Running R}

\section{Basic R Structure}
```

```
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

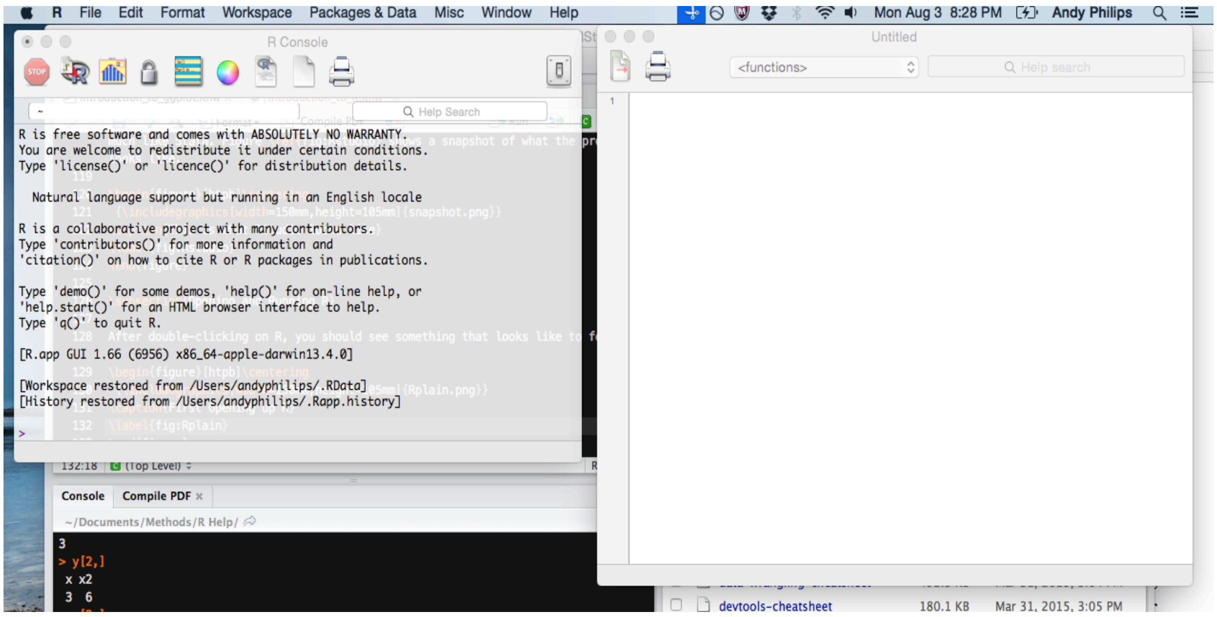
Name	Size	Modified
..		
.Rhistory	21.5 KB	Jul 1, 2015, 6:32 PM
adolph tileplots.R	14 KB	Mar 29, 2014, 11:34 PM
bootstrapping.pdf	729.7 KB	Feb 3, 2015, 1:07 PM
Cook_Swayne Interactive and Dynamic Graphics in R.pdf	6.3 MB	Feb 26, 2014, 9:59 AM
Data cleaning in R.pdf	536.6 KB	Mar 5, 2015, 9:09 AM
data-wrangling-cheatsheet	492.3 KB	Mar 31, 2015, 3:04 PM
devtools-cheatsheet	180.1 KB	Mar 31, 2015, 3:05 PM
dist.gif	36.4 KB	Dec 29, 2014, 9:26 AM
figure		
fun with FUNctions.R	456 B	Jan 21, 2015, 7:49 PM
generic template.R	266 B	Jun 4, 2013, 7:52 AM
ggplot2-cheatsheet	1.2 MB	Mar 31, 2015, 3:05 PM
graphs.R	87 B	Jan 21, 2015, 7:39 PM
heatmaps in R.R	2.3 KB	Mar 10, 2014, 8:50 AM

Figure 1: This is what RStudio looks like

1.3 Opening and Running R

After double-clicking on R, you should see a main console pop up. Then, either select "File - New Document" or use the keyboard shortcut. A blank document should open up, so everything should look like Figure 2.

If instead you are working in RStudio, you just select "File - New R Script" once you



```
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[R.app GUI 1.66 (6956) x86_64-apple-darwin13.4.0]

[Workspace restored from /Users/andyphilips/.RData]
[History restored from /Users/andyphilips/.Rapp.history]

>
```

Figure 2: First Opening up R

open the application.

There two places in which we could type commands. The console, which gives us our output, can handle commands being input. However, for this document we will only type in the Script/Document, and then run the batch (the line we want to run), which gets transferrred over to the console. The next subsection explains why this is a good idea.

Assignment: *Open R or RStudio.*

1.4 Importance of (detailed) R Scripts

Just like Stata, R-Scripts are basically a text document of commands that can be saved and returned to. It is *always* a good idea to do all of your work first in the script, then run it by either selecting the line(s) or using the keyboard shortcut.[2] There are several reasons for this. First, you can add comments using the # symbol (analogous to the $*$ in Stata or % in LaTeX). When you return in a day, week, month, or year, these can be massive timesavers since it lets you (or your co-authors or fellow grad students) see what your thought process was at the time.

[2]There are other GUI programs to call-into R to run the lines from a text editor such as RWinEdt for Windows or TextMate for Mac.

Second, having scripts helps you stay organized. If you are like me, you may find that, despite your best intentions, you end up making lots of datasets, graphs, and scripts. Having everything written down, and more importantly, well documented with dates and where you are pulling data/saving data can help to stay organized. It is always a good idea to create a header in each of your scripts that looks like the following commented-out section below. I tend to have simple preambles to documents that appear as the following:

```
> # --------------------------------#
> #        Andy Philips
> # Texas A&M University
> # aphilips@pols.tamu.edu
> #
> # 8/3/15
> # The purpose of this document is to
> # get familiarized with R
> # --------------------------------#
>
```

but others have even more detailed ones

Third, good replication habits require others to be able to replicate your analysis *exactly*. That is nearly impossible unless you document what you have done.

Finally, borrowing code from yourself/others is a great idea. It is far easier just going back to an existing script and copying and pasting than trying to remember the exact lines you typed. With detailed code, you can just send someone a script you wrote without having to explain all the details.

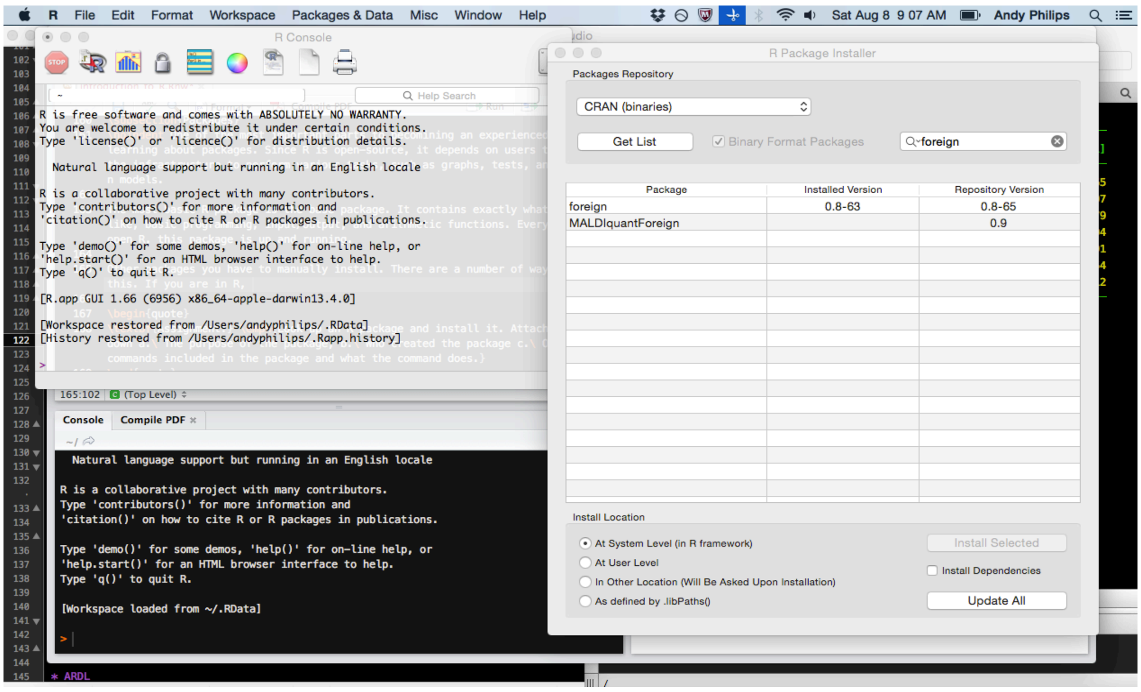
Assignment: *Open up a script in R. Create your own preamble, and save the script (this is done either in the "File" tab, or by pressing "command + s" on a Mac or "control + s" on Windows.*

1.5 Packages

One of the most important parts to becomining an experienced R user is learning about packages. Since R is open-source, it depends on users to create the infrastructure to perform various tasks, such as graphs, tests, and regression models.

The most basic R package is the base package. It contains exactly what it sounds like, basic programming, input/output, and arithmetic functions. Everytime you open R, this package is up and running.

Other packages you have to manually install. There are a number of ways to do this. If you are in R, you have two options. You can click on "Packages & Data" in the menu bar; this opens up a new window where you can search for packages. Let's search for the foreign package, which lets us use other file formats (such as Stata's .dta, .csv, .dat, etc.). This is shown in Figure 3. First type "foreign" into the search bar and press Get List. Then you can press Install Selected.



```
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[R.app GUI 1.66 (6956) x86_64-apple-darwin13.4.0]

[Workspace restored from /Users/andyphilips/.RData]
[History restored from /Users/andyphilips/.Rapp.history]

>
```

Packages Repository: CRAN (binaries)
Binary Format Packages ✓
foreign

Package	Installed Version	Repository Version
foreign	0.8-63	0.8-65
MALDIquantForeign		0.9

Install Location
- [✓ At System Level (in R framework)]
- At User Level
- In Other Location (Will Be Asked Upon Installation)
- As defined by .libPaths()

Install Dependencies

```
 Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[Workspace loaded from ~/.RData]

>
```

Figure 3: Packages

It is now downloaded and stored in one of R's directories. Once you have downloaded a package you do not need to re-install it everytime you open up R.[3] However, in order

[3]It is a good idea however to update your library from time to time.

to place the package in our current R environment, you will need to type:

```
> library(foreign)
```

Now the package is called to the current environment and we can use all of the functions it has.

The other way to install and load packages is through the following typed directly into the console:

```
> install.packages("foreign", dependencies = TRUE)
> library(foreign)
```

This installs the foreign package as well as any other packages that it is dependent to run on. You only need to download a package once, but you need to load it into the current R environment everytime you start a new one. RStudio is similar, except that when you open it up it starts using the previous environment so you do not have to re-load a package. Last, RStudio has a "Packages" tab in one of the consoles that makes searching and loading packages easy.

There are a *lot* of packages in R; as of August 8, 2015, there were 6980 in the CRAN repository. However, there are a few common ones that most people download and use. These include:

- foreign: To load up non-R data formats
- ggplot2: To create great graphs
- lattice: Another graph-maker
- car: Basic statistics (ANOVA)
- zoo: Time series
- plyr/dplyr: Data manipulation

Those who are more interested in learning about packages should check out this page.

Assignment: *Look up three R packages and install them. Then attach them. For each, write down a. The purpose of the package, b. Who created the package c. One of the commands included in the package and what the command does.*

1.6 Help

Getting help with the various functions in R is a key part of learning. Overall, I reccommend getting most of your help online (i.e. on Stack Exchange, Google searches, etc.) rather than buying reference books since online sources tend to identify the most common problems you will experience. Moreover, packages are always changing, and new ones are being created. In Section 7 I list some of the helpful online references.

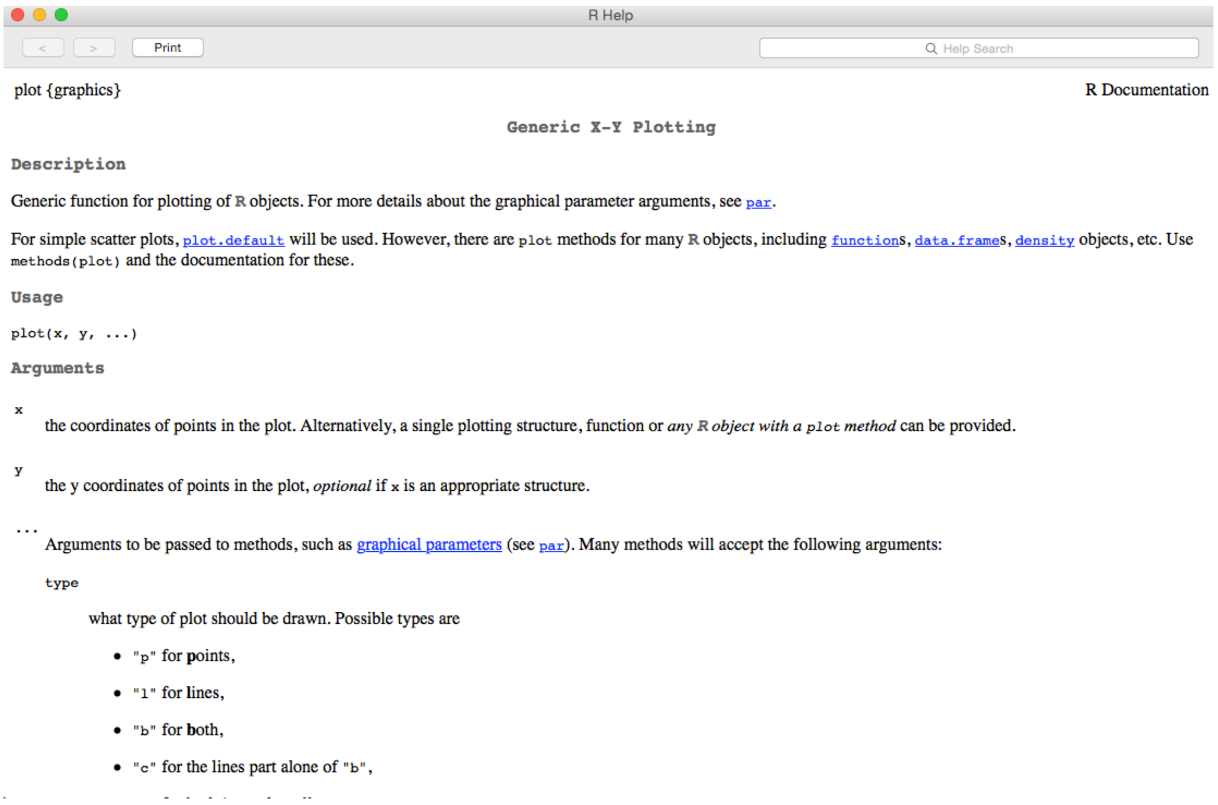
The help files created for each function and package can also be useful. We can access this by typing "?" and then the function. For instance, to learn about the plot function in R, we type

```
> ?plot
```

and a new window opens as shown in Figure 4. A help file has a number of parts. The upper left corner shows the package it comes from. The description gives a brief overview of what the command does. The usage and arguments sections detail what you would type into the script or console as well as what the various options do. Anything in blue you can click on and you will be directed to a new page. The "see also" section lists related commands. Last, at the bottom of every help file are examples of using the function. Some of these are very helpful, others are not.

RStudio makes this a bit easier by having a Help tab in one of the consoles. If you start typing it automatically searches for what you have typed so far.

Again, I have found the help files to be less helpful than the user community online. You may be better off finding online help first. Also, instead of using the ? for help you



plot {graphics} R Documentation

Generic X-Y Plotting

Description

Generic function for plotting of R objects. For more details about the graphical parameter arguments, see `par`.

For simple scatter plots, `plot.default` will be used. However, there are `plot` methods for many R objects, including `function`s, `data.frame`s, `density` objects, etc. Use `methods(plot)` and the documentation for these.

Usage

```
plot(x, y, ...)
```

Arguments

`x`
the coordinates of points in the plot. Alternatively, a single plotting structure, function or *any R object with a `plot` method* can be provided.

`y`
the y coordinates of points in the plot, *optional* if `x` is an appropriate structure.

`...`
Arguments to be passed to methods, such as graphical parameters (see `par`). Many methods will accept the following arguments:

`type`
what type of plot should be drawn. Possible types are

- `"p"` for points,
- `"l"` for lines,
- `"b"` for both,
- `"c"` for the lines part alone of `"b"`,

Figure 4: Using the Help Command

can just type:

```
> help(plot)
```

Assignment: *Look up a help file for a command.*

For some commands, an example is provided.[4] To see this for our plot command, type

```
> example(plot)
```

2 Basic R Structure

Every time you open R, a new workspace is created. This is where R stores *objects*. Objects are anything that we have/are/could work with—such as strings (statements

[4]A related example is called a vignette, which works through a step-by-step example of the command with a dataset.

enclosed within " "), numbers, variables, etc. Objects are case sensitive, so keep that in mind when creating and calling certain objects.

The easiest object to create in R is a vector. R runs mostly based off vectors stored in memory; unlike Stata it is very easy to "peel off" a vector from a dataset and perform an operation on it. Let's create a simple vector named x and see how it is stored in R:

```
> x <- c(1,3,5,6,8)
> x

[1] 1 3 5 6 8
```

What all did we do here? First, we gave a name, `x`, to our vector. Next we assigned the value `c(1,3,5,6,8)` by using the `<-` (the less than sign and the minus sign). The c stands for "concatenate", which means to link things together in a chain or series. When working in R, we use the `<-` to perform an operation (listed to the right of `<-`) which we store as an object (listed to the left of `<-`).[5] Remember to use commas to separate individual entries in the vector. As the output of `x` shows, we end up with a single row with five values.[6]

We can perform transformations on our newly-created vector. For instance, let's multiply everything by two, and assign it to a new object called `x2`:

```
> x2 <- x*2
> x2

[1]  2  6 10 12 16
```

Simple enough. Keep in mind that if we would have simply evaluated, not assigned the multiplication, we would not have created a new object. For instance,

[5]It is possible to flip the arrow signs the other way, so that the assigned object is to the right. Another assignment I have seen has been using the = sign. Overall, these are uncommon and the overwhelming convention seems to be to assign an object to the left of the arrow.

[6]Another (probably more correct) way of seeing what is in x is to type print(x)

```
> x*2
```

```
[1]  2  6 10 12 16
```

does not create anything new; we just get output in the console.

Sometimes, especially with long vectors, it can be helpful to learn the length of the vector. We can do this by using the following command:

```
> length(x2)
```

```
[1] 5
```

Just like in Stata, we can ask for summaries about the vector (another way of thinking about this is as a variable) we just created:

```
> summary(x2)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
    2.0     6.0    10.0     9.2    12.0    16.0 
```

Summary is actually a very utilitarian command—and always seems to provide useful information for almost every object.

> **Assignment:** *Create your own vector in R. Perform a mathematical transformation on it. What is the mean and standard deviation of your transformed vector?*

As mentioned above, instead of assigning a vector, we could have just evaluated it. If we do not assign a vector an object name, it is lost after the evaluation. So for instance:

```
> 1/x2
```

```
[1] 0.50000000 0.16666667 0.10000000 0.08333333 0.06250000
```

does not get saved anywhere.

RStudio makes it easy to see what objects we have created in our environment, as well as their class. These appear in the “Environment” tab (typically on the upper-right quadrant of RStudio). To see the current list of objects when using plain R, type:

```
> objects()
```

```
[1] "x"  "x2"
```

We can see that both x and x2 are currently in the environment. If we wanted to remove them we could type `rm(x x2)`; removing objects is relatively uncommon since you typically just write over them.

2.1 Special R Structures

Vectors are pretty basic, and form the basis of a lot of programming and computations in R. However, there are a number of other important data structures.

Matrices are simply two (or more) vectors combined together. For example, if we wanted to combine vectors `x` and `x2` from above to create a new vector, we could perform the following using the column-bind command:

```
> y <- cbind(x, x2)
> y
```

```
     x x2
[1,] 1  2
[2,] 3  6
[3,] 5 10
[4,] 6 12
[5,] 8 16
```

If we wanted to, we could declare `y` to be a matrix using `as.matrix(y)` and then we could perform matrix arithimetic on it. If instead we wanted to create two rows of five columns rather than two columns of five rows, we would have used the `rbind()` command to row-bind

```
> y <- rbind(x, x2)
> y
```

```
   [,1] [,2] [,3] [,4] [,5]
x     1    3    5    6    8
x2    2    6   10   12   16
```

Notice that we just overwrote `y`. R will overwrite any similarly-named object in the global environment, so take care to always use new names if you do not want this to happen.

Our previous operations were performed on vectors that were the same length. What happens when our vectors are different lengths?

```
> p1 <- c(1,2,3,4,5,6)
> p2 <- c(1,2,3)
> p.comb <- cbind(p1,p2)
> p.comb
```

```
     p1 p2
[1,]  1  1
[2,]  2  2
[3,]  3  3
[4,]  4  1
[5,]  5  2
[6,]  6  3
```

Note how p2 just got repeated again. Thankfully, R gives a warning message when you run this. Be careful when combining vectors of uneven lengths.

Assignment: *Create and combine at least two vectors.*

2.2 Special R Commands for Data Structures

We already saw how to use things like cbind to combine columns. But we can also create vectors using other useful commands. For instance, if we wanted to make a vector that was a sequence of numbers that increased by 0.5 and had a range from 2 and 20, we could type:

```
> ss <- seq(2,20, by=.5)
> ss

 [1]  2.0  2.5  3.0  3.5  4.0  4.5  5.0  5.5  6.0  6.5  7.0  7.5  8.0  8.5  9.0
[16]  9.5 10.0 10.5 11.0 11.5 12.0 12.5 13.0 13.5 14.0 14.5 15.0 15.5 16.0 16.5
[31] 17.0 17.5 18.0 18.5 19.0 19.5 20.0
```

Another command is `rep( )`, which repeats objects in a vector. For instance, to make an new vector that repeats the vector of x 3 times, type

```
> c1 <- rep(x, times = 3)
> x

[1] 1 3 5 6 8

> c1

 [1] 1 3 5 6 8 1 3 5 6 8 1 3 5 6 8
```

Instead, to repeat the first object in x three times, the second object in x three times, etc...we could specify `, each`:

```
> c1 <- rep(x, each = 3)
> c1

 [1] 1 1 1 3 3 3 5 5 5 6 6 6 8 8 8
```

2.3 Accessing Rows/Columns

R also makes it relatively easy to access particular parts of a vector. For instance, if we had a single vector (1:n or n:1), we could type:

```
> x

[1] 1 3 5 6 8

> x[2]

[1] 3
```

to get the second entry of the vector. If we have something more complex, the the matrix y, we use row-column—[row,column] (typically called "row *i*, column *j*") directions within the square brackets:

```
> y

   [,1] [,2] [,3] [,4] [,5]
x     1    3    5    6    8
x2    2    6   10   12   16

> y[2,]

[1]  2  6 10 12 16

> y[,2]

 x x2
 3  6

> y[2,1]
```

```
x2
 2
```

```
> y[1,2]
```

```
x
3
```

In `y[2,]`, we ask for all of the values in the second row. In `y[,2]` we ask for all of the values in the second column. In `y[2,1]`, we ask for the second row, first column. Last, in `y[1,2]`, we ask for the first row, second column. Note that R also tells us the name of the original vector (x or x2) that was combined to form `y`. Accessing individual components of an object can be very useful.

> **Assignment:** *With the combined vector you created: a. Access a entire row, b. Access an entire column, c. Access a specific* `ij` *row-column.*

2.4 Names and String Vectors

Adding strings (known as characters in R) to vectors is easy, and very similar to how we created numerical vectors. For instance, we could make a vector of countries. The only difference is we have to enclose each string in quotations:

```
> country <- c("Malta", "Turkey", "Vietnam", "Mexico")
> country
```

```
[1] "Malta"   "Turkey"  "Vietnam" "Mexico"
```

3 Programming in R

There are lots of useful programming tips when using R. We will focus on two somewhat related ones: functions and loops. Loops perform a repetitive task, such as transforming a list of variables. Functions have an input that we can define, a task that it carries out, and some end result (variable, statistic) that comes out of the function. Programs are difficult to learn, but they *will greatly increase your efficiency.*

3.1 Loops

Loops are carried out via the following, "for every [something] in [a list of something]: do [something]"

```
> q <- rep(NA, 5)
> for (i in 1:length(x))        {
+         q[i] <- x[i]/3
+ }
> q

[1] 0.3333333 1.0000000 1.6666667 2.0000000 2.6666667
```

What's going on here? First, we use `q <- rep(NA, 5)` to create an empty vector of length 5. It is filled with NAs (R's version of missing data). Next, `for (i in 1:length(x)) {` means that for every entry in x (from the first to the entire length), we are going to fill in the corresponding entry in q with the x entry divided by three (`q[i] <- x[i]/3`). Make sure to close the loop with the last bracket on its own line. Of course, this is a stylized example; we could have simply typed `q <- x/3`. Still this shows the powerful capabilities of loops to carry out a repetitive task.

3.2 Functions

The other advanced programming topic in R is creating functions, which define a task, define inputs to the task, and carry it out wherever we would like.[7] For example, let's say we wanted to find the t-statistic (beta over the standard error) for a made up variable.

[7]Functions are actually much more common than loops in R.

```
> # create data
> beta <- rnorm(10, mean = 1)
> se <- rnorm(10, sd=3)
> # define program
> t.stat <- function(b,s) {
+         tstat <- b/s
+ }
> # now run our program and print the results
> print(t.stat(beta,se))

 [1]  0.78146959 -0.04583574 -0.25682202 -2.58260277 -0.15442883  0.43088860
 [7]  0.80027807  0.11483445  1.79403220  0.11418403
```

Our first step was to create some data using the `rnorm()` command to generate 10 observations that are normally distributed (more on using distributions to create data later). We are able to set the mean and standard deviation of the distribution by adding options using a comma. The next step is to define the program, called "t.stat". We assign it a function where the user plugs in two inputs, `b` and `s` (beta and standard error, respectively). These are known as the arguments. Then the function calculates the t-statistic. This is called the expression section. The value we get is a vector we create called tstat. Finally we run the program on the beta and standard error we just created.

Functions can get *much* more complicated than this, but it does give you a basic idea. First, define a function name. Second, define your arguments in a list. Then define your expression (what calculation you are carrying out). Third, what value do you want? Then you close with a bracket. Functions are extremely useful for repetititve tasks and can greatly speed up the time it takes to accomplish tasks. The same goes for loops. We can even enclose loops within other loops.

4 Opening, Using and Saving Pre-Existing Datasets

So far we have generated and manipulated objects that we created ourselves. More commonly we will be working with datasets to conduct our research. R is good at handling a variety of data types. In this section, we will discuss how to open, view, and save datasets.

4.1 Working With Working Directories

The first step is to set the working directory, or the folder that we want to open, work with, and save data in. We do this through the following command:

```
> setwd("/Users/andyphilips/Documents/Texas A&M/Bootcamp Course")
```

Of course, you would put in your own file path. Right-clicking on an item in the folder you want and selecting "Get Info" on a Mac or "Properties" in Windows is the fastest way to get a file path. To see (in R) the files that are in your current working directory, type:

```
> list.files(path=".")
```

To see what the current working directory is:

```
> getwd()
```

```
[1] "/Users/andyphilips/Documents/Texas A&M/Bootcamp Course"
```

> **Assignment:** *Set your current working directory to the path where you placed your Bootcamp files.*

4.2 Opening Data

To open up a dataset in our current working directory, we may need the foreign package. With it, we can open up files that are saved under something else besides .RData, such as a .dta or .xlsx. Let's open the Bootcamp dataset that is saved as a Stata file:

```
> library(foreign)
> data <- read.dta("bootcampdata.dta", convert.underscore = FALSE,
+         convert.factors=TRUE)
```

We have now just created an object called data from the bootcampdata file. The options in the command are to convert any "_" in the dataset into a ".", which R is better at handling. In addition, we keep any names of factor variable labels instead of their underlying number (the blue variables in Stata).[8]

If we wanted to open an Excel spreadsheet file saved as a csv (comma-separated values), we could do so in a similar way.[9] Remember that we still need to have the foreign package loaded up. Instead of `read.dta( )`, for a .csv file we would type:

```
> data <- read.csv("bootcampdata.csv")
```

There are a few other ways to load in non-R data into R, but these are the most common. R is very flexible in the types of data you can import—in fact, one time I wanted to use SPSS data (.dat file) in Stata, so I loaded the .dat into R and saved it as a .dta!

> **Assignment:** *Load up the "bootcampdata". Use whichever data format you prefer.*

4.3 Examining and Attaching Data

Now that we have the data saved as an object in R, we can examine it more. If we want to view *all* of the data, we could type the following, which opens up a spreadsheet-type window

[8]Unfortunately, new versions of Stata (13 and 14) save the data in a format not readable by foreign (or Stata 12!). There appears to be a readstata13 library for R that works, but I do not think one has been created for 14 yet.

[9]It is almost always better to save a dataset as some multi-platform, readable format like csv, than something like xls or xlsx.

```
> View(data)
```

Slightly easier is just to get the header, which lists the variables names (in order) as well as the first couple observations:

```
> head(data)
```

```
  ccode              country          regimetype  gini   elf literacy
1     4          Afghanistan Direct Presidential    NA 0.668       29
2     8              Albania       Parliamentary    NA 0.064       85
3    12              Algeria Direct Presidential 38.73 0.299       57
4    20              Andorra                        NA    NA       NA
5    24               Angola Direct Presidential    NA 0.783       42
6    28  Antigua and Barbuda                        NA    NA       NA
  agriculture_pc       fdi     gdp_pc poverty_pct businessburden
1       45.15848  0.000000   628.4074          NA             90
2       26.31162  3.034135  4954.1982        8.76             41
3       10.00360  1.866684  6349.7207       23.61             24
4             NA        NA         NA          NA             NA
5        7.85317 14.626760  2856.7517       70.21            119
6        3.78930  9.230832 13981.9790          NA             31
```

It looks a little messy, so let's just list the names of the variables in the dataset

```
> names(data)
```

```
 [1] "ccode"          "country"        "regimetype"     "gini"
 [5] "elf"            "literacy"       "agriculture_pc" "fdi"
 [9] "gdp_pc"         "poverty_pct"    "businessburden"
```

We also probably want to attach the data. Attaching data means that when you give a command (such as telling R to run a regression), it assumes the data you are talking about is the one that is attached. We can do this by the following, which attaches data and calls it dat:

```
> attach(data)
```

Now that our data are attached, we can learn things about certain variables. This dataset has data on institutions and socio-economic conditions in a number of countries. Let's get simple summary statistics for the amount of foreign direct investment:

```
> summary(fdi)
```

```
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.     NA's
-10.1400   0.7167   2.5480   7.6820   5.0930 524.9000       19
```

Note that if we did not attach the data first, R would look through all objects and be unable to find an object called fdi. Instead, we told it to look within the 'data' environment. If we had multiple datasets we wanted to work with, we could specify the dataset and variable we wanted in order to let R know where to look. We could type the following:

```
> summary(data$fdi)
```

```
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.     NA's
-10.1400   0.7167   2.5480   7.6820   5.0930 524.9000       19
```

See how the $ is used to separate the dataset and variable. If we wanted a different variable in the environment called `data`, we would replace fdi with whatever we wanted. If we wanted to use a different envrironment, we would replace `data` with the environment we wanted to work with.

If we wanted to get a graphical depiction of the distribution, we could create a histogram:

```
> hist(fdi)
```

We could also create QQ plots. These let us compare the distribution of a variable to a normal distribution. Let's do this for the percentage of the population that is living in poverty:

```
> qqnorm(poverty_pct)
> qqline(poverty_pct)
```

A normal distribution would be perfectly lined up with the diagonal line. There appear to be very fat tails in the poverty distribution.

4.4 Saving Data

After working in R, we may want to save our data. There are a number of ways to do this. First, we want to save a single object. If we wanted to save data, for instance, we could type:

```
> save(data, file = "mysavespot.RData")
```

This saves it in the R format .RData. If we wanted to save it as a .csv file, so that it could be accessed by R and Stata, we could type:

```
> write.csv(data, file = "mysavespot.csv")
```

Last, we may want to save the current working environment. To do this, we type:

```
> save.image("mysavedimage.RData")
```

Assignment: *Save an object or your entire working environment.*

5 Statistics in R

5.1 Distributions

R has a lot of probability distributions built right in. You may need to use these if you are generating your own data—to see how the properties of a regression change, for instance. Or just to see what common distributions look like. As we saw above, we can use it to generate made-up data in order to see how a command works.

As an example, here is a normal distribution with 60 observations, a mean of 3, and standard deviation of 1:

```
> norm <- rnorm(60, mean = 3, sd = 1)
```

and we can plot this:

```
> plot(density(norm))
```

Here is the uniform distribution:

```
> uniform <- runif(100)

> plot(density(uniform))
```

> **Assignment:** *Create a distribution and plot it. The stats library contains all the distributions; to see them type library(help = "stats")*

5.2 Linear Regression

One of the most common commands to use is the linear regression, given as lm( ). Let's run an ordinary least squares (OLS) a.k.a. linear regression model that says that inequality (measured through the GINI coefficient) for country *i* is a function of the literacy rate, poverty rate, GDP per capita, and reliance on agriculture:

$$GINI_i = \beta_0 + \beta_1 Literacy_i + \beta_2 Poverty_i + \beta_3 GDP_i + \beta_4 Agriculture_i + \varepsilon_i \qquad (1)$$

To run this in R we would type

```
> res.1 <- lm(gini ~ literacy + poverty_pct + gdp_pc + agriculture_pc)
```

Here we used the `lm( )` command to run the linear regression. This is in the base library, so we don't have to load any library to run `lm( )`. Always specify the dependent variable (gini in this case) first. Then use the ~ symbol—the tilde to separate the dependent and independent variables, and plus signs to separate the independent variables. To see the results, which we assigned to object res.1, we type the following:

```
> summary(res.1)

Call:
lm(formula = gini ~ literacy + poverty_pct + gdp_pc + agriculture_pc)


Residuals:
    Min      1Q  Median      3Q     Max
-17.472  -7.454  -2.342   8.402  20.994


Coefficients:
                 Estimate Std. Error t value Pr(>|t|)
(Intercept)     4.529e+01  7.735e+00   5.856 1.35e-07 ***
literacy       -3.500e-02  7.073e-02  -0.495    0.622
poverty_pct     5.740e-02  8.112e-02   0.708    0.482
gdp_pc         -9.711e-05  4.439e-04  -0.219    0.827
agriculture_pc -1.381e-01  1.454e-01  -0.949    0.346
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1


Residual standard error: 10.41 on 71 degrees of freedom
  (118 observations deleted due to missingness)
```

```
Multiple R-squared:  0.0269,        Adjusted R-squared:  -0.02792
F-statistic: 0.4907 on 4 and 71 DF,  p-value: 0.7425
```

The result is a standard regression table. We get the "call" (the command we typed), summary statistics of the residuals, and coefficient estimates, standard errors, t-statistics, and p values. Below, we get some fit statistics (R^2) and overall model fit tests (F-statistic). Notice that by default, R estimates with a constant. If we did not want this for some reason, we could instead add a minus 1. This is just R's way of knowing we want the constant suppressed:

```
> res.1 <- lm(gini ~ literacy + poverty_pct + gdp_pc + agriculture_pc - 1)
> summary(res.1)

Call:
lm(formula = gini ~ literacy + poverty_pct + gdp_pc + agriculture_pc -
    1)


Residuals:
    Min      1Q  Median      3Q     Max
-23.461  -9.379   2.333  11.219  26.917


Coefficients:
                Estimate Std. Error t value Pr(>|t|)
literacy       0.2923150  0.0524196   5.576 4.04e-07 ***
poverty_pct    0.3437102  0.0782787   4.391 3.80e-05 ***
gdp_pc         0.0011167  0.0004747   2.352   0.0214 *
agriculture_pc 0.0895250  0.1694695   0.528   0.5989
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1


Residual standard error: 12.58 on 72 degrees of freedom
```

```
  (118 observations deleted due to missingness)
Multiple R-squared:  0.9204,       Adjusted R-squared:  0.9159
F-statistic:   208 on 4 and 72 DF,  p-value: < 2.2e-16
```

Note too that if we wanted to change the data that are used for the regression we could specify the `data` as an option. Since we defined our dataset as an object named "data", we type the following:

```
> res.1 <- lm(gini ~ literacy + poverty_pct + gdp_pc + agriculture_pc - 1,
+                          data = data)
```

In practice, it is *never* a bad idea to specify your dataset using the data option, even if you have already used `attach( )`.

While the summary of the results are helpful, other times we may want to access certain stored characteristics. These are called values, and are stored within res.1. Note that *almost all* estimation-type commands store at least some values after they are run; so you can use this for commands other than OLS. To see them, type:

```
> names(res.1)
```

```
 [1] "coefficients"  "residuals"     "effects"       "rank"
 [5] "fitted.values" "assign"        "qr"            "df.residual"
 [9] "na.action"     "xlevels"       "call"          "terms"
[13] "model"
```

So if we wanted to access the coefficients we could type (remember that $ is used to access things within an object)

```
> res.1$coefficients
```

```
      literacy    poverty_pct         gdp_pc agriculture_pc
   0.292314984    0.343710159    0.001116736    0.089524996
```

How is this helpful? We might want to create a plot of our y variable (GINI) against the residuals in order to see how well our model fits, for instance. To do this, first we need to make a subset that contains the observations that were run in res.1 (thus, we need to remove missing observations)

```
> subset <- cbind(gini, literacy, poverty_pct,gdp_pc,agriculture_pc)
> subset <- as.data.frame(subset) # tell R this is a dataset
> subset <- na.omit(subset) # listwise delete missings
```

Note that we created an object called subset, which we combined some of the variables from `data` in. Then we used `as.data.frame( )` to let R know that this is a dataset. If we didn't do this, we would not be able to use certain commands. Next, we used `na.omit( )` to omit the missing observations. It does this via listwise deletion (i.e. delete the entire row is just one observation-variable is missing)...just like a regression would do. Now we can plot GINI from the subset object against the residuals from the model:

```
> plot(subset$gini,res.1$residuals)
```

It looks like our model is systematically underfitting for low values of GINI, and systematically overfitting at high values.

5.3 Factor Variables

So far, it looks like we are doing a pretty bad job explaining differences in the GINI coefficient across countries. Maybe the regime type of a country explains GINI? Remember though that the regime type is a factor variable:

```
> head(data$regimetype)

[1] Direct Presidential Parliamentary       Direct Presidential
[4]                           Direct Presidential
Levels:  Assembly President Direct Presidential Parliamentary
```

That is, we really care about the distinct names of the variables, not the underlying value that they are given in a single factor variable. In this case, it looks like 1 is a strong president, 2 is a parliamentary, and 0 is a direct president. Let's add them to our model. This is done by enclosing the factor variable—regimetype in this example, in `factor( )` within `lm()`:

```
> res.2 <- lm(gini ~ literacy + poverty_pct + gdp_pc + agriculture_pc +
+                          factor(regimetype), data = data)
> summary(res.2)

Call:
lm(formula = gini ~ literacy + poverty_pct + gdp_pc + agriculture_pc +
    factor(regimetype), data = data)

Residuals:
    Min      1Q  Median      3Q     Max
-17.138  -6.644  -2.378   7.263  21.616

Coefficients:
                                        Estimate Std. Error t value Pr(>|t|)
(Intercept)                           35.6152346  8.7904470   4.052 0.000131
literacy                               0.0039665  0.0700051   0.057 0.954980
poverty_pct                            0.0872339  0.0780596   1.118 0.267644
gdp_pc                                 0.0001950  0.0004425   0.441 0.660787
agriculture_pc                        -0.1216004  0.1391175  -0.874 0.385104
factor(regimetype)Direct Presidential  6.5197208  3.7194116   1.753 0.084063
factor(regimetype)Parliamentary       -1.9251040  4.3715239  -0.440 0.661043

(Intercept)                           ***
literacy
```

```
poverty_pct
gdp_pc
agriculture_pc
factor(regimetype)Direct Presidential .
factor(regimetype)Parliamentary
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 9.929 on 69 degrees of freedom
  (118 observations deleted due to missingness)
Multiple R-squared:  0.1389,       Adjusted R-squared:  0.06404
F-statistic: 1.855 on 6 and 69 DF,  p-value: 0.1011
```

Assignment: *Run an OLS regression; you can choose which variables to include. Make sure to include regimetype as a factor variable.*

5.4 Exporting to LaTeX

After we run a regression, we may want to add it to one of our LaTeX articles we have been working on. Exporting a table to LaTeX is a good idea, since there is a high chance of making an error if we were to type in all the data by hand. In addition, it saves us a lot of time. I could find five different packages that export various types of regression tables in R: xtable, memisc, texreg, stargazer, and apsrtable. No one package is compatible with every type of regression model.[10] But they all can export simple linear models like the one we ran above. Let's load (and install, if we need to, by uncommenting) the texreg package, which is probably the most common.

```
> #install.packages("texreg", dependencies= TRUE)
> library(texreg)
```

[10]An overview on which packages can run which models can be found here.

So far, we have run two regressions and saved them as res.1 and res.2. Let's put them both in a regression table that we can export:

```
> texreg(list(res.1,res.2), caption = "Our First Exported Table",
+                                    model.names = c("Reg 1", "Reg 2"))

\begin{table}
\begin{center}
\begin{tabular}{l c c }
\hline
 & Model 1 & Model 2 \\
\hline
literacy                                & $0.29^{***}$ & $0.00$        \\
                                        & $(0.05)$     & $(0.07)$      \\
poverty\_pct                            & $0.34^{***}$ & $0.09$        \\
                                        & $(0.08)$     & $(0.08)$      \\
gdp\_pc                                 & $0.00^{*}$   & $0.00$        \\
                                        & $(0.00)$     & $(0.00)$      \\
agriculture\_pc                         & $0.09$       & $-0.12$       \\
                                        & $(0.17)$     & $(0.14)$      \\
(Intercept)                             &              & $35.62^{***}$ \\
                                        &              & $(8.79)$      \\
factor(regimetype)Direct Presidential &              & $6.52$        \\
                                        &              & $(3.72)$      \\
factor(regimetype)Parliamentary         &              & $-1.93$       \\
                                        &              & $(4.37)$      \\
\hline
R$^2$                                   & 0.92         & 0.14          \\
Adj. R$^2$                              & 0.92         & 0.06          \\
Num. obs.                               & 76           & 76            \\
RMSE                                    & 12.58         & 9.93           \\
\hline
\multicolumn{3}{l}{\scriptsize{$^{***}p<0.001$, $^{**}p<0.01$, $^*p<0.05$}}
\end{tabular}
\caption{Our First Exported Table}
\label{table:coefficients}
\end{center}
\end{table}
```

It's a little hard to see the output here but basically it takes all the coefficient estimates and associated standard errors and p-values and turns it into LaTeX code. From there you can just copy and paste it into your document. The end result looks like Table 1. Notice how much of the work is done for us. We still have to change the variable names to something that makes sense to readers (although some articles I have read literally include Stata/R variable names as is...don't do this!). It also looks like the intercept should be renamed "Constant" and placed below the factor variables.

In fact, if we wanted to make variable names, we could do it within R:

```
> texreg(list(res.1,res.2), caption = "Our Second Exported Table",
+   model.names = c("Reg 1", "Reg 2"), custom.coef.names =
+   c('Literacy', 'pct in Poverty', 'GDP pc',
+          'pct Income from Agriculture', 'Constant', 'Direct President',
+          'Parliamentary'))
```

And the output would look like Table 2.

The texreg package is very useful. If we wanted to visually compare two or more models side-by-side, but didn't want to export to LaTeXor elsewhere, we could use the following to generate a nice-looking table that appears in the R console:

```
> screenreg(list(res.1, res.2))
```

Table 1: Our First Exported Table

	Model 1	Model 2
literacy	0.29^{***}	0.00
	(0.05)	(0.07)
poverty_pct	0.34^{***}	0.09
	(0.08)	(0.08)
gdp_pc	0.00^{*}	0.00
	(0.00)	(0.00)
agriculture_pc	0.09	-0.12
	(0.17)	(0.14)
(Intercept)		35.62^{***}
		(8.79)
factor(regimetype)Direct Presidential		6.52
		(3.72)
factor(regimetype)Parliamentary		-1.93
		(4.37)
R^2	0.92	0.14
Adj. R^2	0.92	0.06
Num. obs.	76	76
RMSE	12.58	9.93

$^{***}p < 0.001$, $^{**}p < 0.01$, $^{*}p < 0.05$

Table 2: Our Second Exported Table

	Model 1	Model 2
Literacy	0.29^{***}	0.00
	(0.05)	(0.07)
pct in Poverty	0.34^{***}	0.09
	(0.08)	(0.08)
GDP pc	0.00^{*}	0.00
	(0.00)	(0.00)
pct Income from Agriculture	0.09	-0.12
	(0.17)	(0.14)
Constant		35.62^{***}
		(8.79)
Direct President		6.52
		(3.72)
Parliamentary		-1.93
		(4.37)
R^2	0.92	0.14
Adj. R^2	0.92	0.06
Num. obs.	76	76
RMSE	12.58	9.93

$^{***}p < 0.001$, $^{**}p < 0.01$, $^{*}p < 0.05$

```
===========================================================
                                       Model 1    Model 2
-----------------------------------------------------------
literacy                                0.29 ***   0.00
                                       (0.05)     (0.07)
poverty_pct                             0.34 ***   0.09
                                       (0.08)     (0.08)
gdp_pc                                  0.00 *     0.00
                                       (0.00)     (0.00)
agriculture_pc                          0.09      -0.12
                                       (0.17)     (0.14)
(Intercept)                                       35.62 ***
                                                  (8.79)
factor(regimetype)Direct Presidential              6.52
                                                  (3.72)
factor(regimetype)Parliamentary                   -1.93
                                                  (4.37)
-----------------------------------------------------------
R^2                                     0.92       0.14
Adj. R^2                                0.92       0.06
Num. obs.                              76         76
RMSE                                   12.58       9.93
===========================================================
*** p < 0.001, ** p < 0.01, * p < 0.05
```

6 Graphics

R has a very intuitive base graphics package. We have already used it a bit above with the `plot()` command, but it is helpful to go into greater detail. Plots are generally divided into high-level and low-level commands. High-level commands define the plot type; for instance when we typed `plot()`. Low-level commands add specific characteristics to an existing plot. We already saw above how to make a scatterplot between two variables:

```
> plot(subset$gini,res.1$residuals)
```

This is a quick plot, and rather ugly. It is certainly not publication quality. To fix this, let's give it a title:

```
> plot(subset$gini,res.1$residuals, main = 'GINI vs. Residuals')
```

and we can add X and Y axis labels:

```
> plot(subset$gini,res.1$residuals, main = 'GINI vs. Residuals'
+                   , xlab='GINI', ylab='Residuals')
```

We can also add a subtitle:

```
> plot(subset$gini,res.1$residuals, main = 'GINI vs. Residuals'
+                   , xlab='GINI', ylab='Residuals',
+                   sub='Our Model Does Not Fit Well')
```

Since we really want white noise residuals centered around 0, it may be helpful to add a horizontal lines. This is done using the low-level command `abline( )` after we create the plot:

```
> plot(subset$gini,res.1$residuals, main = 'GINI vs. Residuals'
+                   , xlab='GINI', ylab='Residuals',
+                   sub='Our Model Does Not Fit Well')
> abline(h=0)
```

To make it pop we can add color to our points and our line:

```
> plot(subset$gini,res.1$residuals, main = 'GINI vs. Residuals'
+                    , xlab='GINI', ylab='Residuals',
+                    sub='Our Model Does Not Fit Well', col='plum3')
> abline(h=0, col='tomato1')
```

Last, since it is hard to see the points, we can use the `pch` option to change the syle of the points to a triangle (and the `bg` option to fill in the triangles):

```
> plot(subset$gini,res.1$residuals, main = 'GINI vs. Residuals'
+                    , xlab='GINI', ylab='Residuals',
+                    sub='Our Model Does Not Fit Well',
+                    col='plum3', pch = 24, bg='plum3')
> abline(h=0, col='tomato1')
```

Note that we used the default plot style, which is a scatterplot. We could specify this by the option `type='p'` in the `plot( )` command. If we had a time series, we may want to draw a line by using the option `type='l'` instead.

We could also draw a regression line through two variables that are plotted:

```
> res.3 <- lm(gini~businessburden,data=data)
> plot(businessburden,gini)
> abline(res.3)
```

First we run the regression between GINI and the burden of businesses to get a license. Then we plot the two. Finally we add in the regression line using `abline( )`. As we can see, the business burden does not do a very good job of predicting the GINI coefficient.

> **Assignment:** *Make a plot. It can be a scatterplot, line plot, or something else. Define the axes, and add color and other "fancy" components as desired. Use* `colors()` *to see all the available default colors in R.*

6.1 Combining Plots

Sometimes we may want to show two or more plots side by side. The way to do this is to use the `par( )` command. For instance, if we wanted to make a figure that had two plots (1 row, 2 columns), we would type the following:

```
> par(mfrow=c(1,2)) # tell R we want 1 row, 2 cols
> plot(subset$gini,res.1$residuals, main = 'GINI vs. Residuals'
+                  , xlab='GINI', ylab='Residuals')
> plot(subset$poverty_pc, subset$gini, main = 'GINI vs. Poverty',
+                  xlab='Poverty', ylab='GINI')
```

The command `mfrow=c(1,2)` is where we specify the number of rows and columns of plots: `mfrow=c(rows,columns)`. We can do any number of combined plots; for instance here is a 2 by 2 plot of the kernel densities of a number of important variables:

```
> par(mfrow=c(2,2))
> plot(density(data$gini, na.rm=TRUE), col= 'steelblue')
> plot(density(data$poverty_pc, na.rm=TRUE), col = 'salmon')
> plot(density(data$gdp_pc, na.rm=TRUE), col='palegreen')
> plot(density(data$agriculture_pc, na.rm=TRUE), col='grey17')
```

Note that to plot densities you have to enclose the density command within plot. In addition, since there were missing values for some observations, I had to use the `na.rm=TRUE` command to get rid of them.

Assignment: *Combine at least two plots.*

6.2 Exporting Graphics

Like tables, we want to save our plots to use in papers. This is easily done in R. R has the following formats:

- PDF's (.pdf)
- Windows Metafile (.wmf)
- Portable Network Graphics (.png)
- JPEG (.jpg)
- BMP (.bmp)
- Postscript file (.ps)

One way to save graphs is just to make the graph, let it pop up in the graphics device (i.e. X11 or Quartz) and then "File/Save As". This is not ideal since the size of the window is not displayed, and so it can look strange if you blow it up or down in size. Instead, we probably want to turn on the graphics device with a pre-specified size and file type, run the graph, and save it before turning off the device. Saving as a pdf is almost always the best file type, though if your picture is appearing on the web you may want to save as a .png. This link provides some good advice about exporting graphs in R.

To save a graph as a .pdf, we can type the following:

```
> pdf(file="graph1.pdf",width=400,height=350)
> plot(x=rnorm(100), y=rnorm(100), main="My First Saved Plot")
> dev.off()
```

```
pdf
  2
```

Assignment: *Create and save a plot. It can be one from the regression analysis section, or one using simulated data.*

6.3 Advanced Graphics

Last, there are two really nice graphics packages you can add: lattice and ggplot2. Here is a little taste of what it can do using the ggplot package. First we simulate some data:

```
> #install.packages("ggplot2", dependencies=TRUE)
> library(ggplot2)
> set.seed(20343)
> # Make up some cross-sectional data about corruption's effect
> # on growth:
> corruption <- rnorm(150,6,2)
> democracy <- as.numeric(rnorm(150)>0)
> growth <- 20 - 5*log(corruption) + 5*democracy  + rnorm(150)
> dataset <- data.frame(growth,corruption,democracy)
```

And we can create a scatterplot with relative ease, as shown in below:

```
> ggplot(data=dataset, aes(x=corruption,y=growth)) +
+         geom_point(color="blue") +
+         labs(x="Level of Corruption", y = "Growth Rate",
+         title= "Corruption's Effect on Growth") +
+         theme_minimal()
```

If you are interested in this, feel free to check out the downloads section on my website, where I have a .R example of creating various graphs using ggplot. It is really a nice package since it lets you "add on" additional options; this makes moving from a simple graph to a complicated one quite easy. Although the base-level R graphs look pretty good, ggplot takes full advantage of everything R has to offer, and in my opinion produces superior graphs.

7 Resources and Other Help

This document was designed to help you get familiar for R for the first time if you have not seen it before. In addition, it shows the capabilities that R has in data management, graphics, and statistics. However, there is much more detailed help available online. In addition to any links above, below are a list of websites or .pdf documents that I have found particularly useful.

- RStudio has some nice "cheatsheets" for using ggplot to make graphs, as well as some more advanced topics: https://www.rstudio.com/resources/cheatsheets/.
- Quick-R has many useful guides to various tasks in R: http://www.statmethods.net/.
- R-bloggers is a user-contributed blog that is very active. It is worth signing up for their daily emails: http://www.r-bloggers.com/.
- Rseek is supposed to work like Google's search function, but only direct you to relevant R topics: http://rseek.org/.
- If your're feeling really adventurous, you can check out Shiny. It uses R code to create (and embed) interactive graphs, tables, and charts: http://shiny.rstudio.com/.
- Here is an introduction to R available on CRAN: https://cran.r-project.org/doc/manuals/R-intro.pdf
- There is a WikiBooks for R. I have not looked at this thouroughly, but I have had good experience with other WikiBooks: https://en.wikibooks.org/wiki/R_Programming/Introduction