LEGISLATIVE POLICY AND RESEARCH OFFICE
Technical Brief

Creating Interactive Webmaps in R (Choropleths)

The use of interactive Webmaps provides LPRO with powerful visual representations that can easily be distributed to members. Webmapping provides an interactive means of representing data overlaid with spatial dimensions that allows exploration of data in ways often not immediately accessible. Examples can be seen here. This piece will provide a template for the creation of interactive maps for the exploration of quantitative data with regions (polygons).

WORKING WITH R

Using R can be intimidating at first. R is used by typing precise commands into the IDE, which will respond to exactly what is typed. R also can install new libraries (packages of commands) on the fly, allowing maximum flexibility once comfort with the system has been achieved. Here we can see what R looks like when we first open it:
There are four boxes available to us here. The top left box, known as the Integrated Development

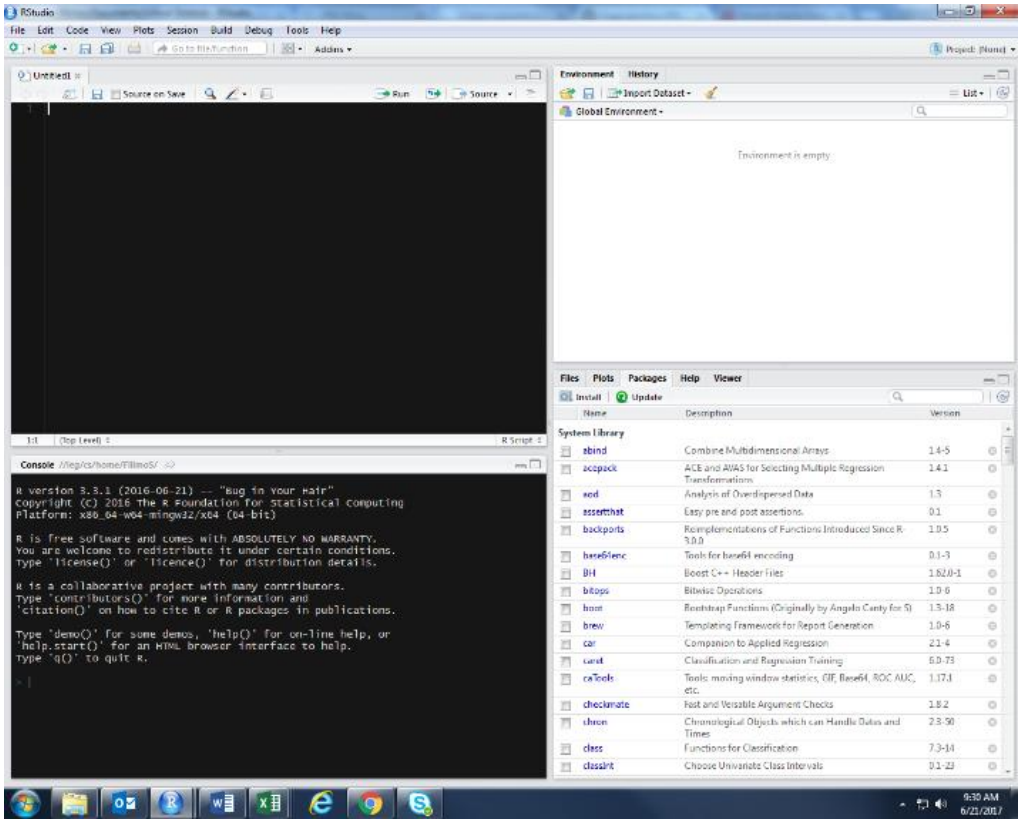

Figure 1: Opening R

Environment (IDE), is the area we will be inserting commands. The top right box, known as the Global Environment, shows us everything that has been loaded into R. Right now, we see it is empty. The bottom left box, the console, gives us information about what R is currently working on or error messages. Finally, the bottom right box is the help/viewer box which provides technical support or displays some types of output. This can be displayed here:

Sione Filimoehala

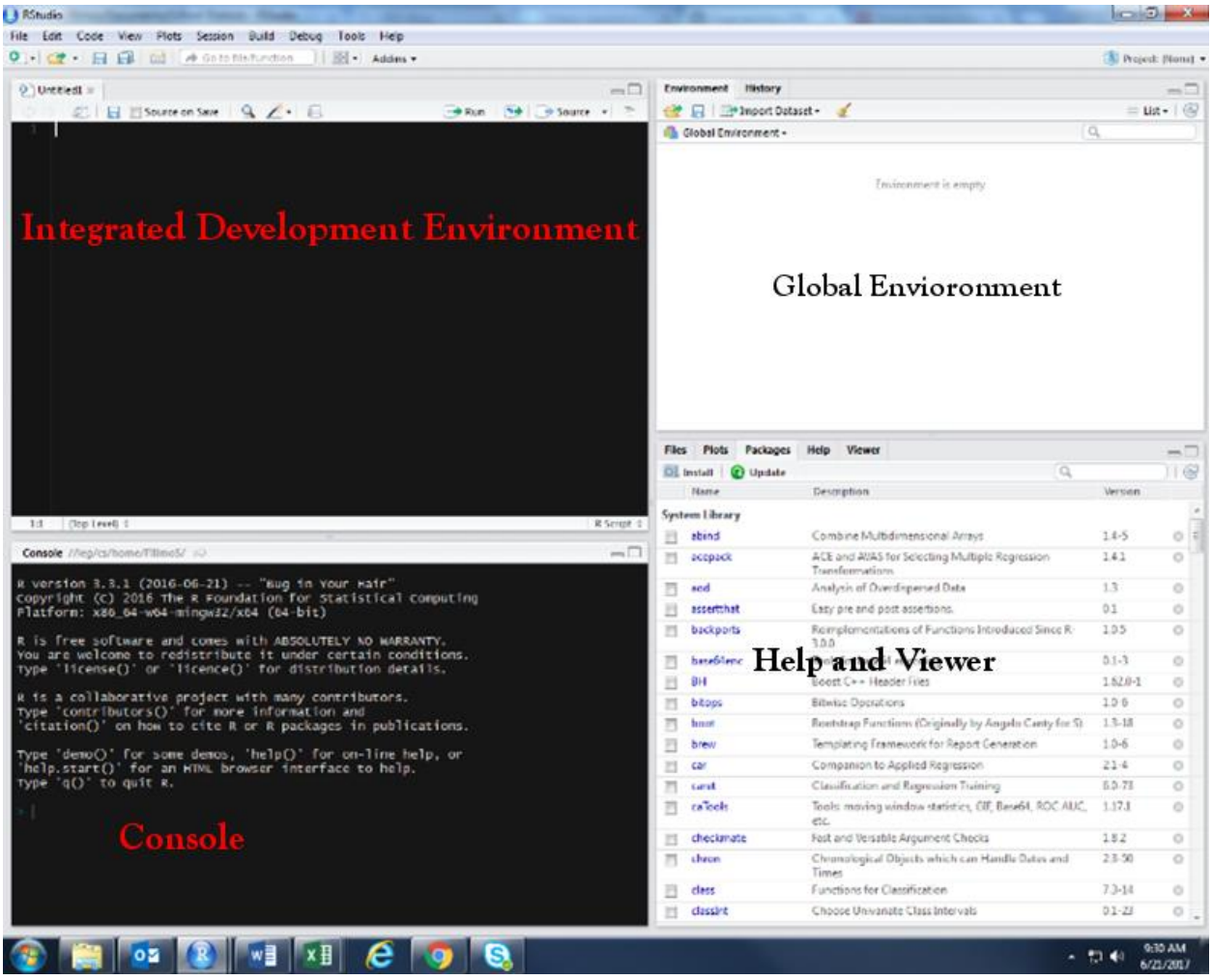

Figure 2: R Components

From this basic understanding, we can start to create some maps.

STARTING TO MAP

The first step to creating Webmaps with R is to load the required libraries. These libraries provide R with the necessary commands to produce Webmaps. Three libraries are important for this task: Rgdal, leaflet, and magrittr. Rgdal is an extension for R that allows it to read GIS data, leaflet provides the Webmaps, and magrittr gives some shortcuts for the code we will write.

The first time we use these libraries, we will have to install them into R. This can be done in the Help and Viewer box. Select the "Packages" and click "Install" which can be seen here[1]:

[1] Shorthand: Packages>Install

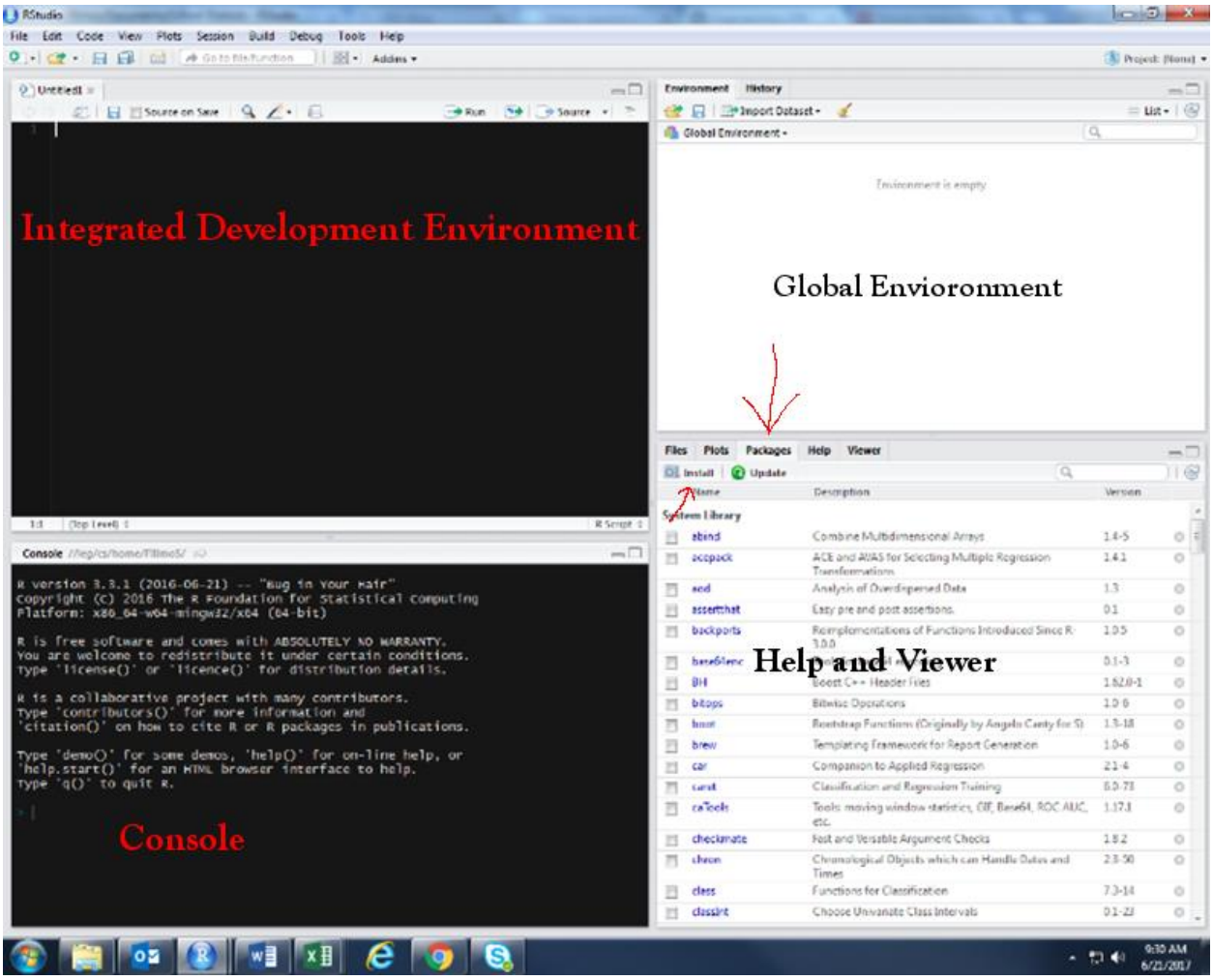

Figure 3: Installing Packages 1

After clicking "Install," a dialogue box will appear. Begin typing the name of the required libraries in the text box (rgdal, leaflet, and magrittr) and the box will provide a dropdown list of the libraries. Select the libraries and click "Install" in the dialogue box.

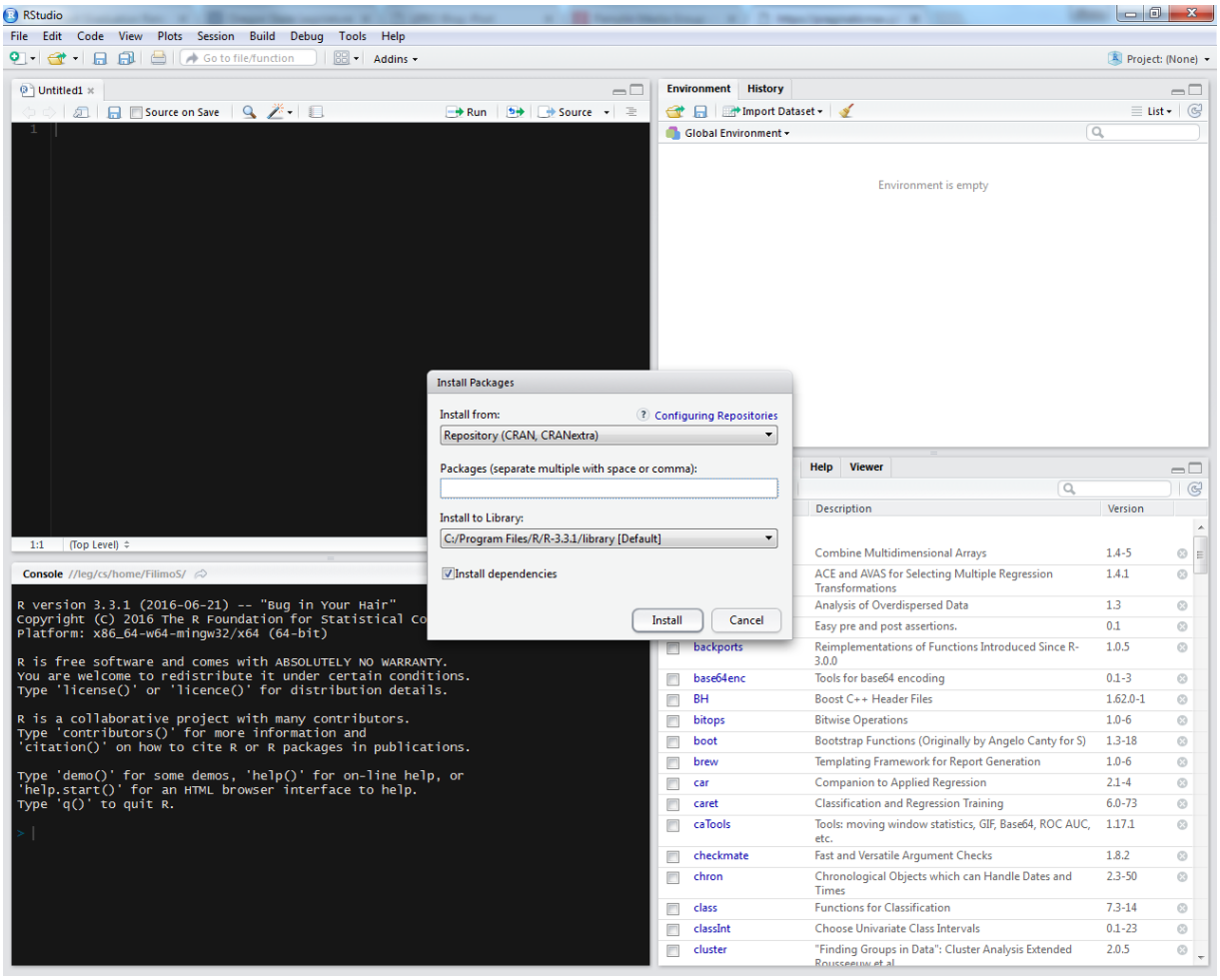

Figure 4: Installing Packages 2

SETTING DIRECTORIES AND LOADING LIBRARIES AND DATA

Now we are ready to start. First, we have to tell R where it can save all generated files and where to look for files. This is known as setting the directory. To set the directory, in the top ribbon click "Session," highlight "Set Working Directory," and select "Choose Directory."

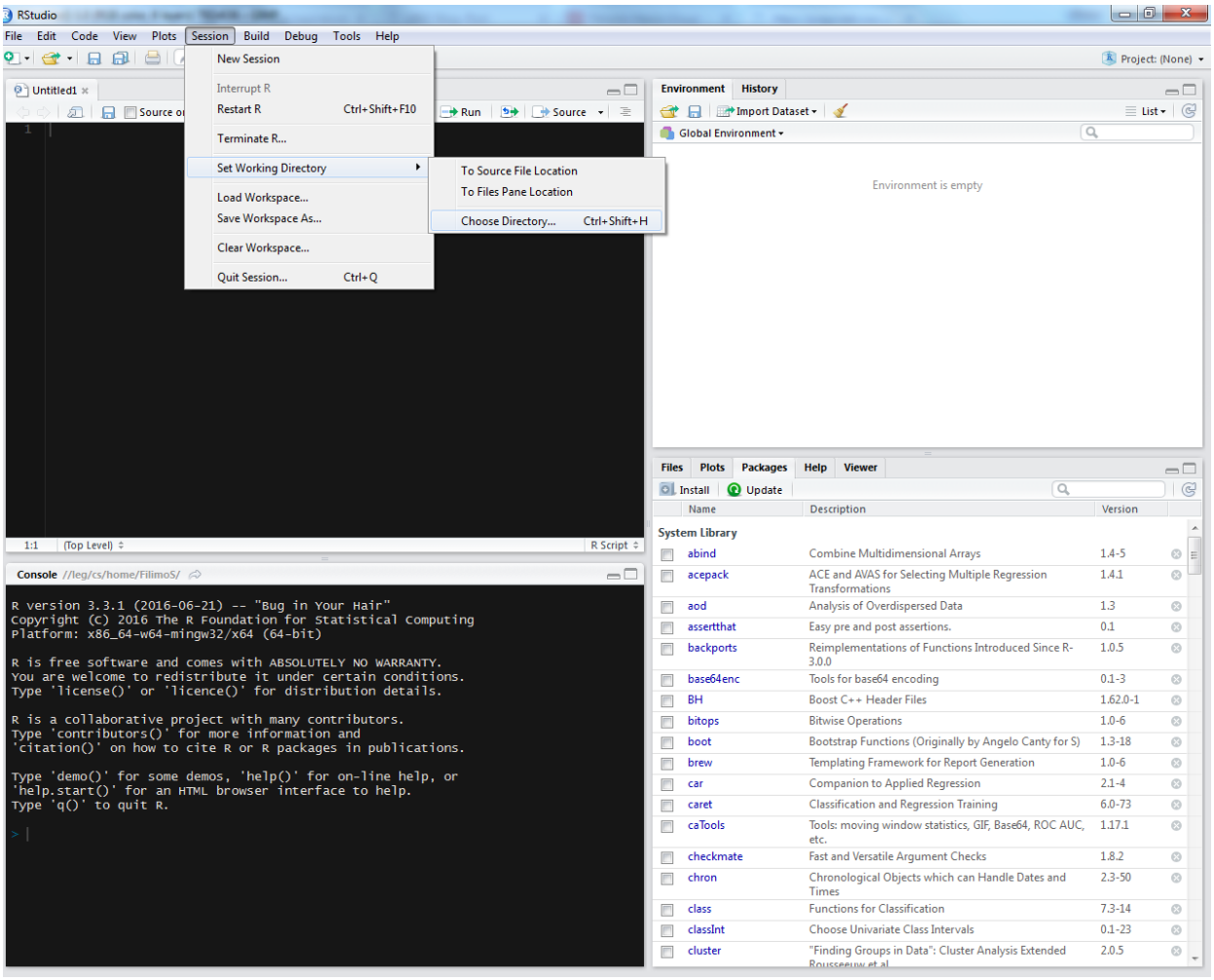

Figure 5: Setting the Directory

A dialogue box will then ask you where you want your directory. This will be a folder that you want to save everything in, and may also contain the data you want to use[2].

Loading Libraries

Now that the directory is set, we load the libraries with the command: library( ). In the parentheses, insert the names of the libraries you would like to load. In this case rgdal, magrittr, and leaflet. That would look like this:

[2] If this folder does not contain all the data, you will have to select it through its long address.

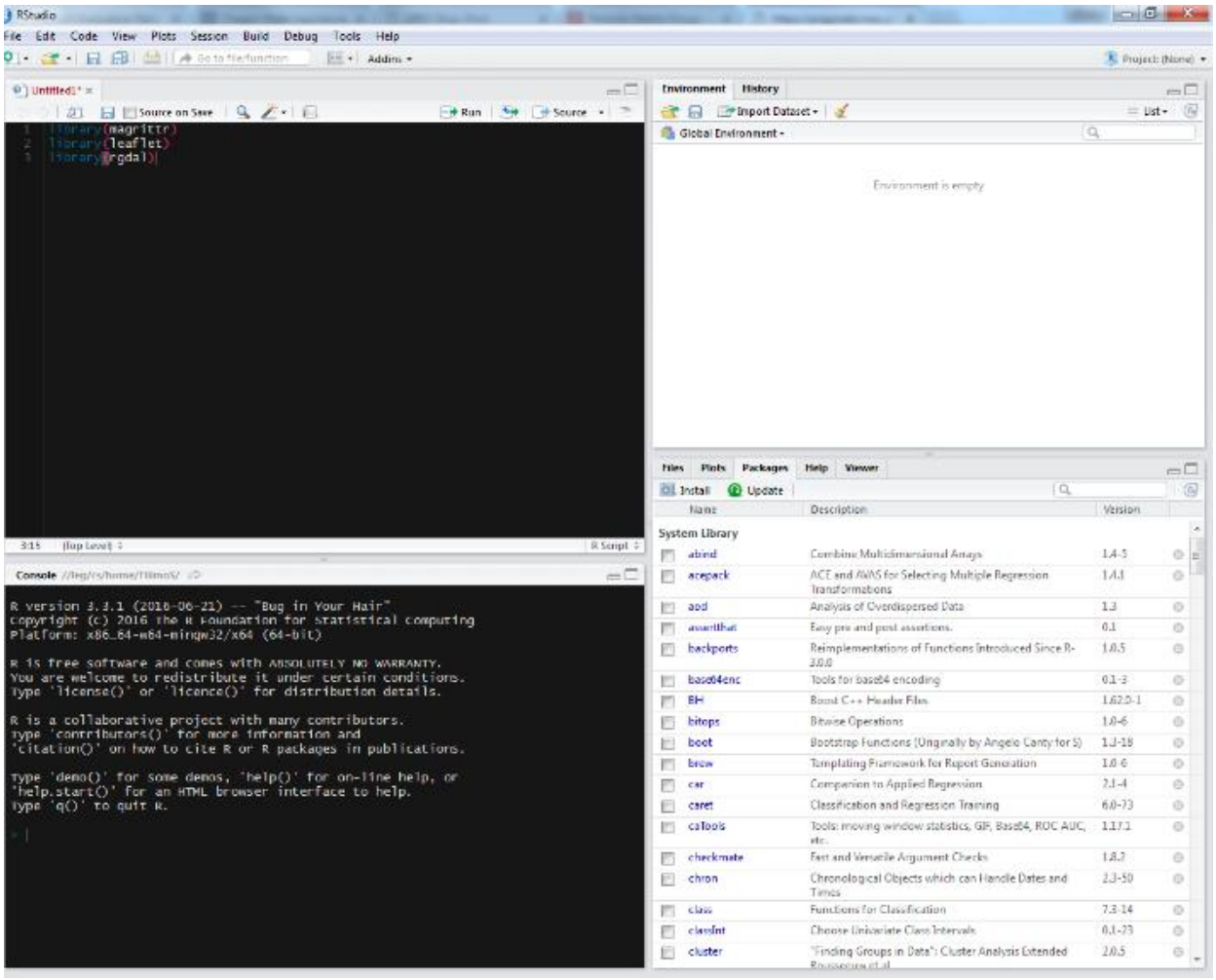

Figure 6: library( ) command

Highlight those lines, then click "Run:"

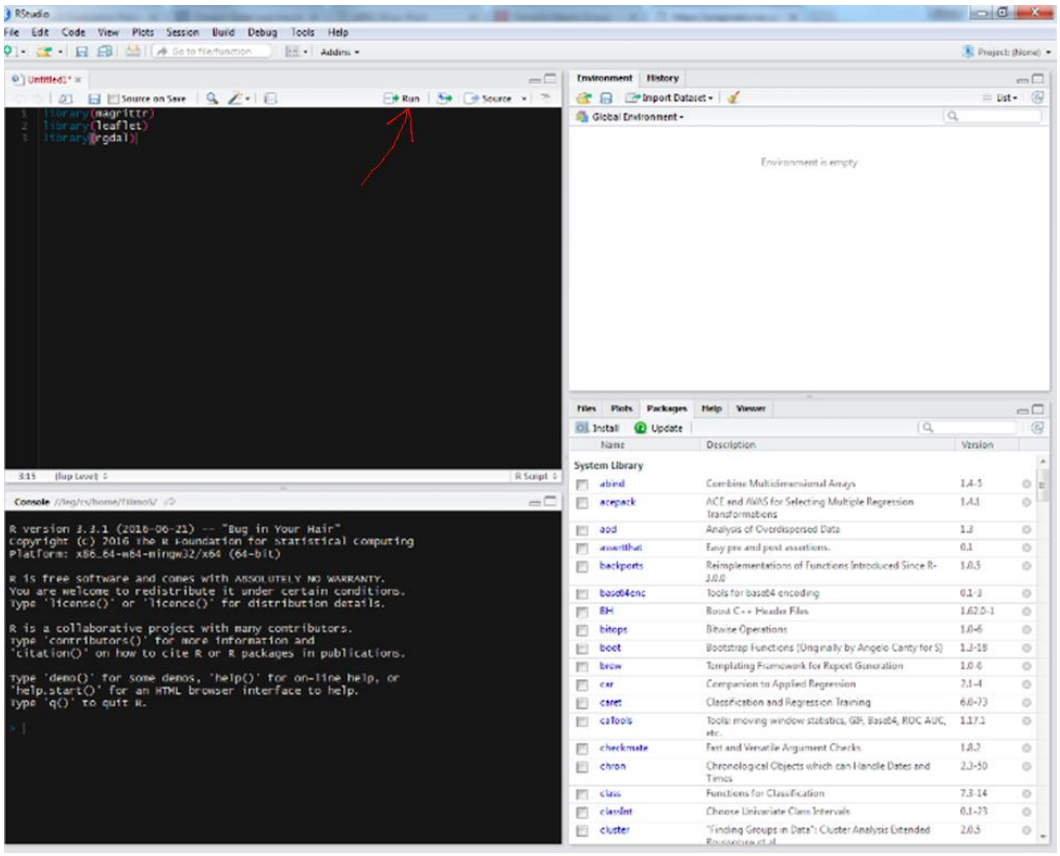

Figure 7: Run Button

From here on, I will use the term "run those lines" or "run that code," which refers to highlighting the indicated text, and clicking the "Run" button. Once the libraries are loaded, we can load our data.

Loading Data

To load data from excel, I suggest formatting the excel data down to the simplest form you need for what you are creating. For example, I will be working with school graduation data. I have a very large data set that includes multiple categories. For this piece I only want my school names, cohort sizes, and graduation rates. As such, my excel file should only have those three variables. Additionally, when making the data simple, save it in the simplest format. For spreadsheets, this would be CSV files. Be sure to save your data as a CSV in excel before proceeding.

To load my CSV, I have inserted a short command that is actually full of simplifications. That command looks like this:

GR.df <-read.csv("Grad Rates.csv")

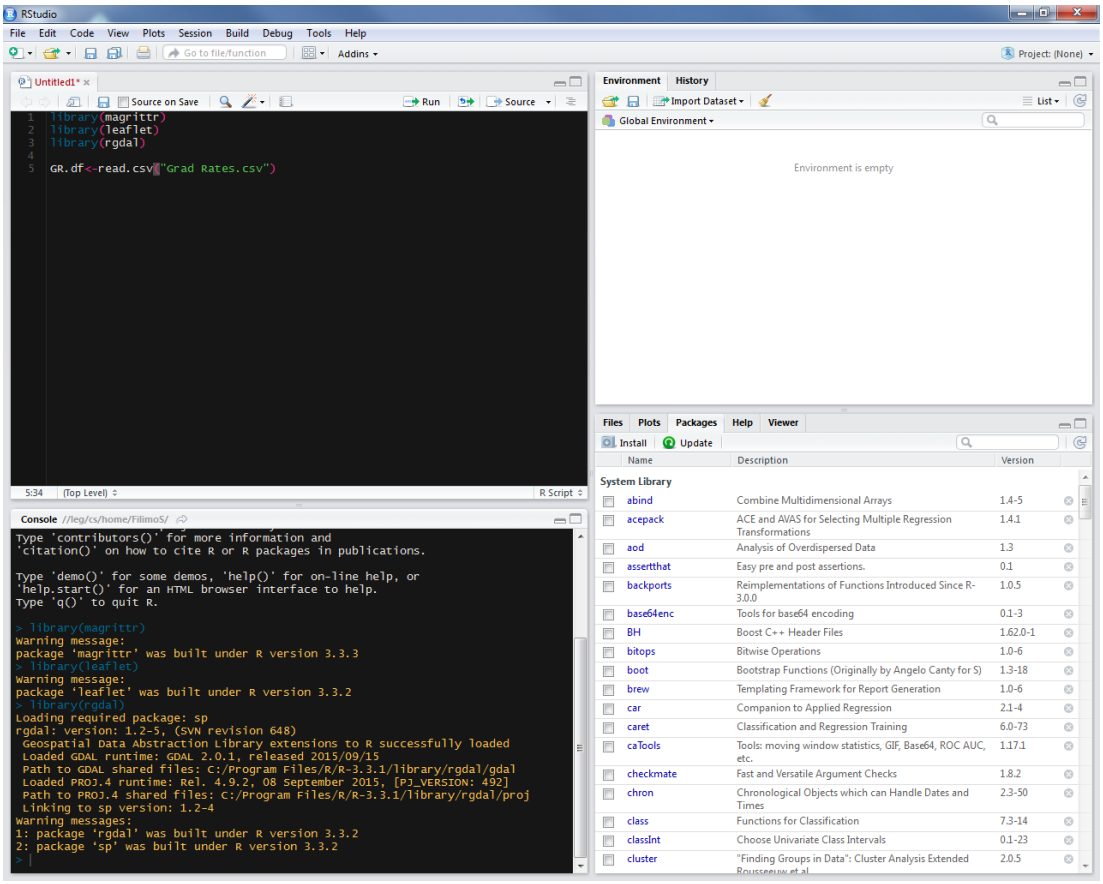

Figure 8: Loading Data- Simple Command

Here I have some built in shortcuts. First GR.df is what I am naming the data while it is in R. GR indicates Graduation Rates, while .df indicates that this is a dataframe[3]. The <- works as a command to shove everything into the GR.df. The command "read.csv( )" tells R to look for csv data, while "Grad Rates.csv" is the data file. Here I can use just "Grad Rates.csv" to grab my data file because it is in the folder we set as the directory earlier. Now that we know what this line is doing, run that line. If the data you want is not in the directory, you must tell R exactly where it is.

[3] Special terminology for spreadsheets in certain contexts

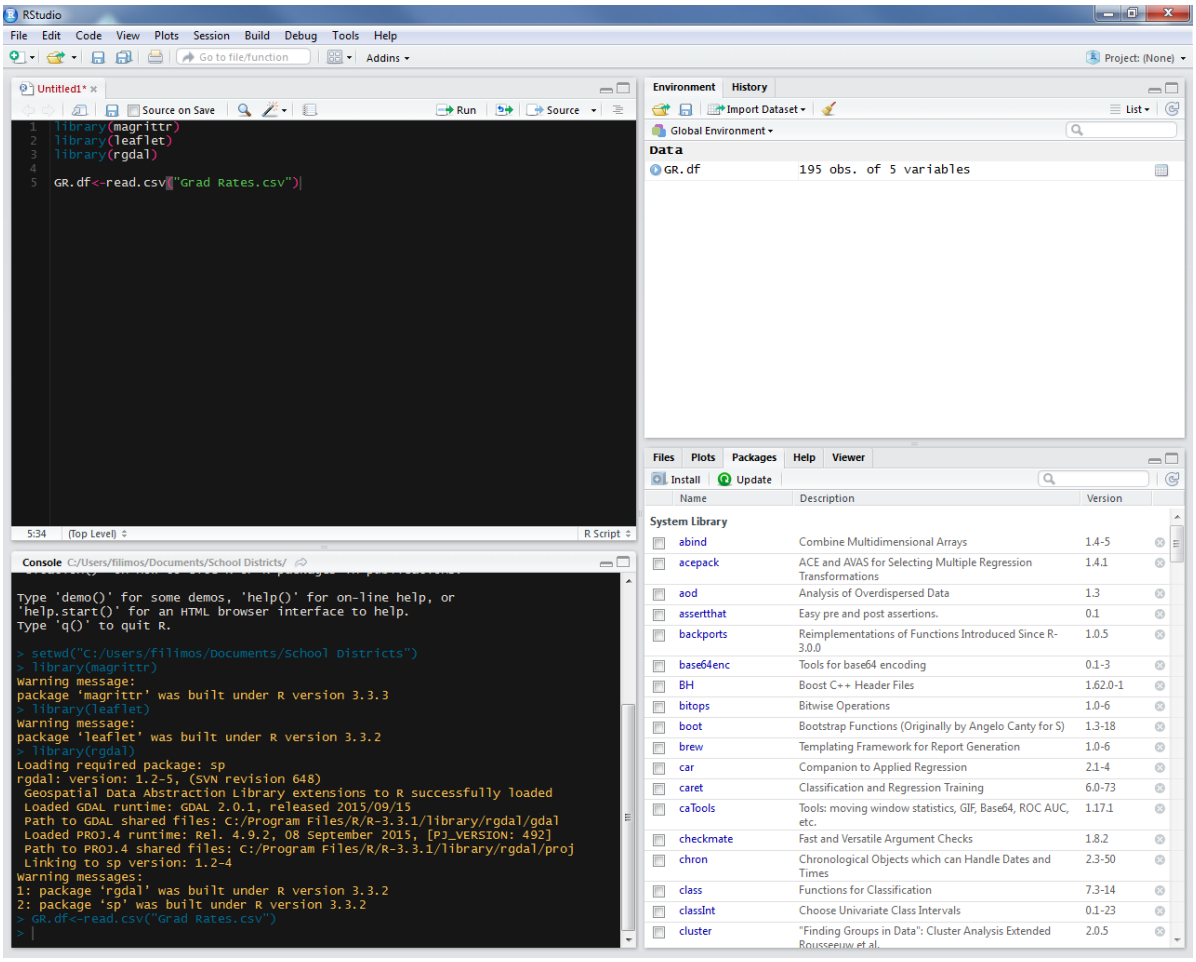

Figure 9: Items in the Global Environment

Now we have something in our global environment. It is the dataframe we created from the CSV file. If you click on it, you will see the data loaded in a spreadsheet:

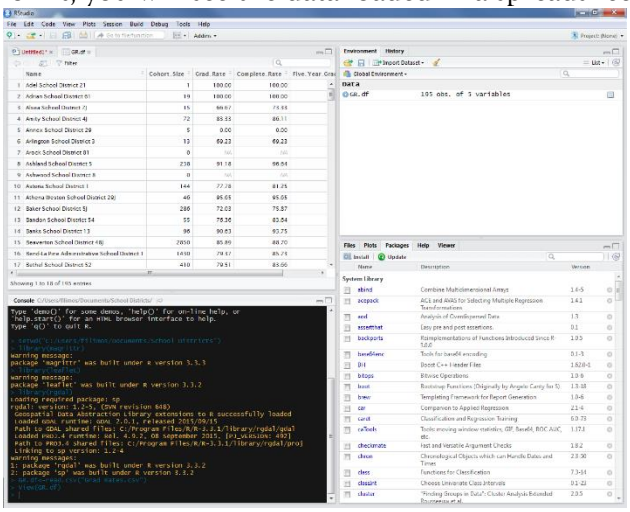

Figure 10: Loaded data as a Spreadsheet

Sione Filimoehala

Loading GIS data

To load the GIS data, we will use the readOGR command provided by Rgdal. In my project, I used this command:

SD.mp <-readOGR(dsn = "C:/Users/filimos/Documents/Maps/School Districts/OR_SD_Com", layer = "OR_SD_Com")

Let's break this down. Here, "**SD.mp**" indicates School District map. I use the dots to indicate what type of data each object is. If you recall, the "**<-**" shoves everything after into what precedes. The "**readOGR( )**" command lets us read GIS data. Inside of the parentheses, I indicate the data source (**dsn = "C:/Users/filimos/Documents/Maps/School Districts/OR_SD_Com"**) then the name of the GIS file (**layer = "OR_SD_Com"**). Be sure to put the comma between the data source and the name of the GIS file. Now let's run this line of code:

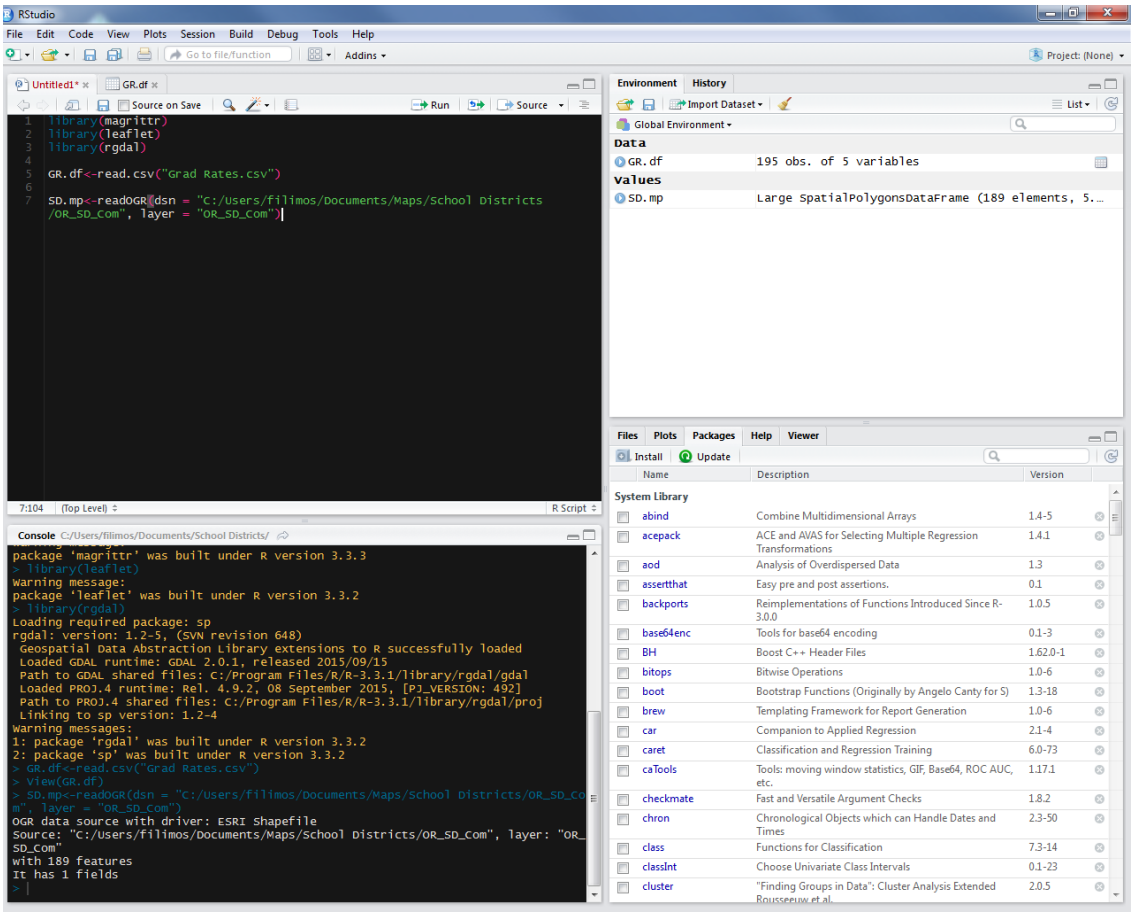

Figure 11: Loading GIS data

Now we have data and a map loaded. Next we have to join the data and map together.

COMBINING DATA

To combined the data, we need something that identifies the data in both the dataframe and the map. Luckily, we have unique names on each of these. On the dataframe, this is the "Name" variable, while in the GIS data this is the "**NAME**" variable. R sees capital letters and lower case letters as different, so these are not the same. If I try to merge the data as it is, I will get

an error message. So I will rename the "**Name**" variable in the dataframe to "**NAME**". This is done with this line of code:

colnames(GR.df)[1]<- "NAME"

This line of code tells R to find the column names of my **GR.df** column 1, and renames it to "NAME." Now I can combine the data.

To combine GIS data, I must make sure that the map data is first when I insert my **merge** command. This can be achieved with this command:

SD.dm<- merge(SD.mp, GR.df, by = "NAME")

Here, I have named this SD.dm to indicate it is the School District data map. Once SD.dm is loaded into my global environment, I can start putting together my final piece.

LABELS AND COLORING

Now that the data is merged, we can start making the Webmap. The Webmap will be a choropleth, which is the technical term for a map that shows different colored regions based on some value. To create the choropleth, we will need to define colors and labels for the regions first. Then we will generate the choropleth. Finally, we will make stylistic changes to the overall look and feel.

Defining Colors

To define colors, first I must define the breakpoints in the data between different values. To do this, I will create some bins with the following command:

cbin <- c(0, 10, 20, 30, 40, 50, 60, 70, 75, 80, 85, 90, 95, 100)

Let's break this down. I have named this object "cbin" to remind me that this is the color bins. On the left side, the **c(0, 10, 20, 30, 40, 50, 60, 70, 75, 80, 85, 90, 95, 100)** combines all of the values to determine the cutoffs between each bin. If there are values outside of the range defined here, they will be treated as missing data.

With these defined bins, now I must assign colors to the values. To do this, I use this command:

pal <- colorBin("RdYlGn", domain = SD.dm$Grad_Rate, bins = cbin)

Here I named the object "pal" to remind me that this is the palette I will be working with. I then use the **colorBin( )** command to define the colors for the bins. '**"RdYlGn"**'indicates a pre-generated color palette that is based on a graduated scale of the colors Red, Yellow, and Green. Please note the quotation marks indicate that we are using something defined somewhere else. The "**domain = SD.dm$Grad_Rate**" indicates that we will be creating the domain of our values based on the domain of **Grad_Rate** variable within our datamap **SD.dm**. the "**$**" indicates to R that we are looking inside of the **SD.dm**. Finally, the **bins = cbin** tells R that the bins we are using is the **cbin** variable we defined. Let's run the code so far:

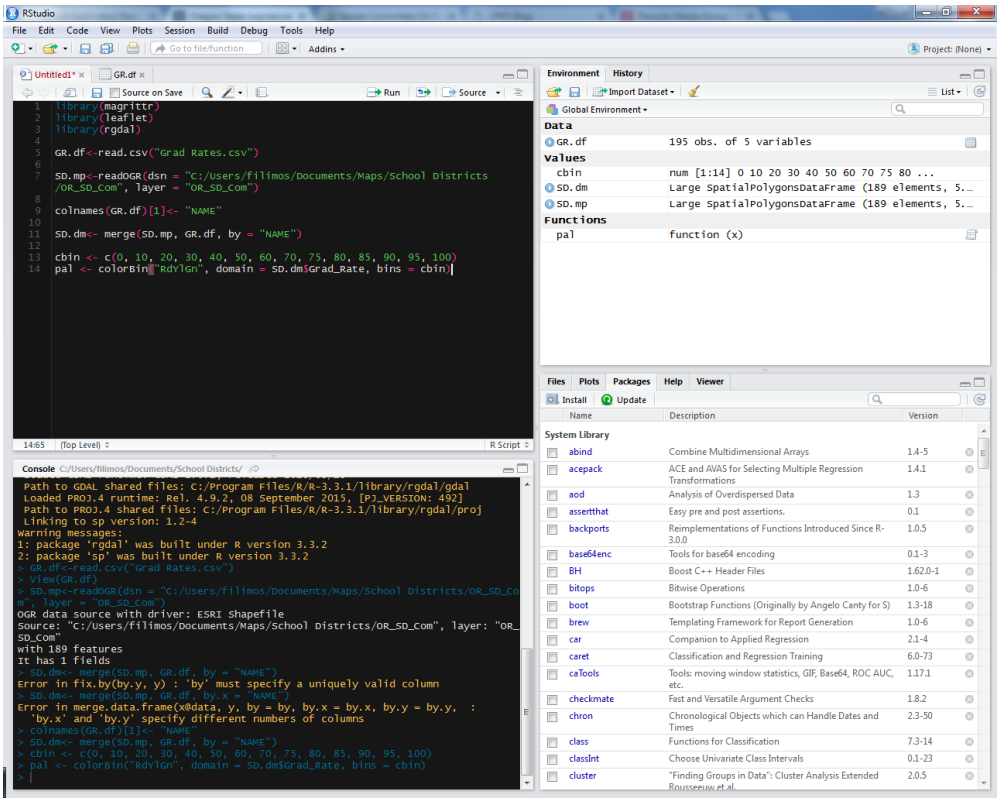

Figure 12: Now with bins and a color palette

Notice now in the Global Environment we have **cbin** and **pal** defined. Next we will create the labels.

Creating the Labels

To create the labels we will define **labels**. To do this, we will use the command:

```
labels <- sprintf(
        "<strong>%s</strong><br/>%g percent Graduated in 2015 -2016",
        SD.dm$NAME, SD.dm$Grad_Rate) %>%
        lapply(htmltools::HTML)
```

This command looks a bit more intimidating than the others, but understanding the parts will show you what is going on. First, we are using the **sprintf** command to create an object that is a combination of numbers and text. Inside the **"%s
%g percent Graduated in 2015 -2016", SD.dm$NAME, SD.dm$Grad_Rate)** we have some html language that tells R how to format the labels. The bit of text "**percent Graduated in 2015 -2016**" can be any text that does not include special character, and will be inserted into our labels. Feel free to customize this section. The second line "**SD.dm$NAME, SD.dm$Grad_Rate**" tells R to create the labels based on the **NAME** and **Grad_Rate** variables. The "**%>%**" is a special command loaded from magrittr that simplifies holding pieces together. The last line "**lapply(htmltools::HTML)**" allows us to use the html formatting mentioned in the second line of this code. After running this bit of code, our R environment should look like this:

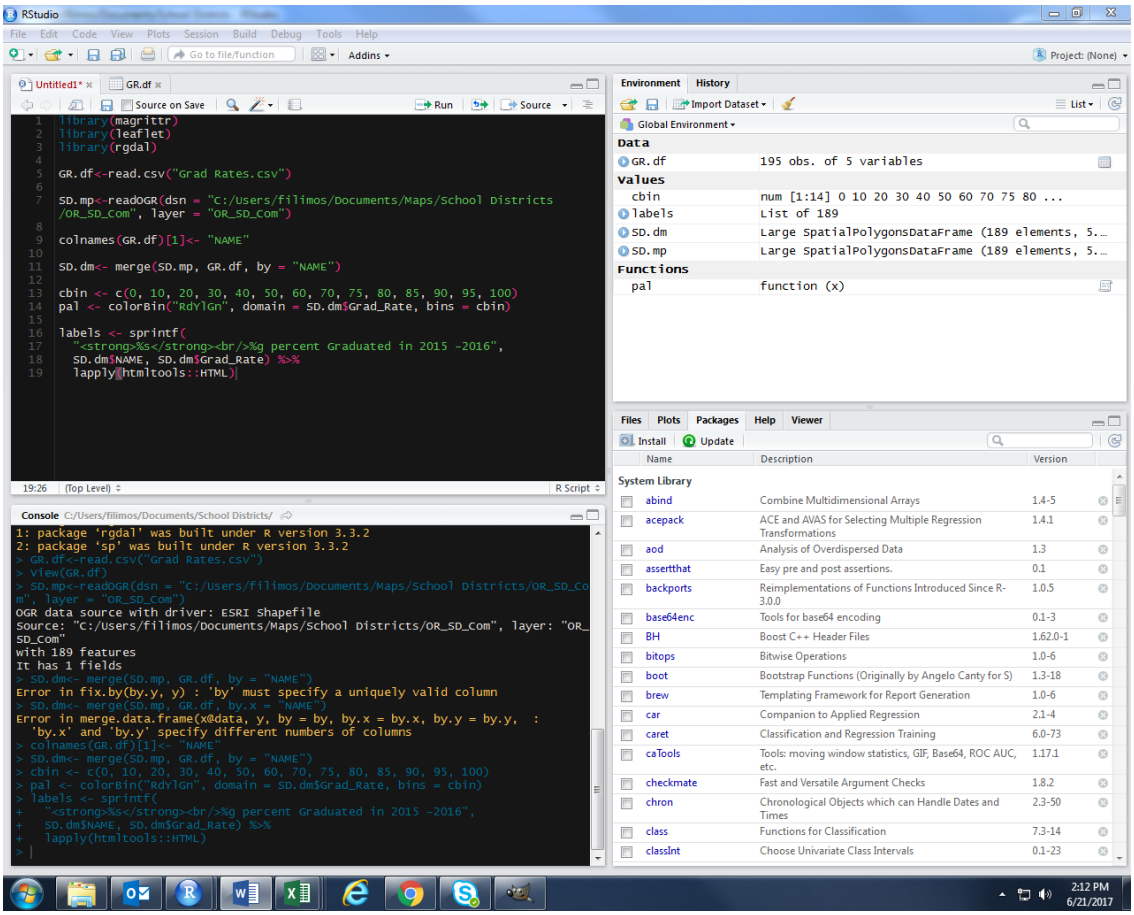

Figure 13: Finished labels

MAP MAKING

This last section of code is the most intensive, but by looking at each individual line we can understand what is happening. Moreover, once we understand what is happening, we can start changing everything to work for how we want everything to look in the final product. First, I will scare you with the full code. Then we will break it down. The full code looks like this:

```
SR.lf<- leaflet(SD.dm, options = leafletOptions(minZoom = 5)) %>%
            setView(lng = -123.0317114, lat = 44.9384532, zoom = 7) %>%
            addProviderTiles(providers$Esri.NatGeoWorldMap) %>%
            addPolygons(
                  fillColor= ~pal(Grad_Rate),
                  weight = 1,
                  opacity = 1,
                  color = 'white',
                  dashArray = "3",
                  fillOpacity = 0.7,
                  highlight = highlightOptions(
                        weight = 5,
                        color = "#666",
```

```
                dashArray = "",
                fillOpacity = 0.7,
                bringToFront = FALSE),
    label = labels,
    labelOptions = labelOptions(
        style = list("font-weight" = "normal", padding = "3px 8px"),
        textsize = "15px",
        direction = "auto")) %>%
    addLegend(pal = pal, values =~ Grad_Rate, opacity = 0.7,
        title = "Graduation Rates in the 2015-16 School Year",
        na.label = "No Data",
        position = "bottomright")
```

SR.lf

A bit scary, but there are piece in here you might recognize already. Let's contrast this with the final product:

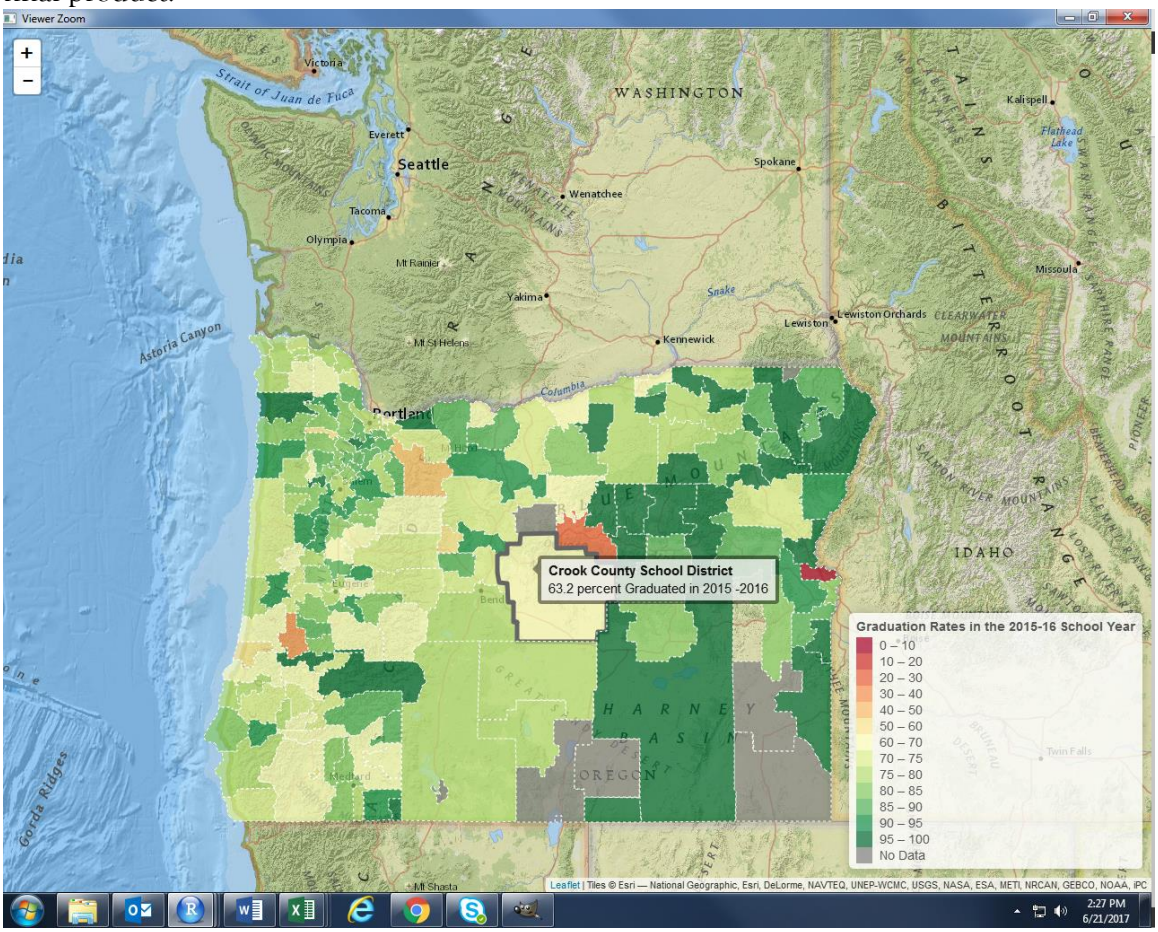

Figure 14: Final product. We are almost there!

To begin, we will break down the first line of the map making code:

SR.lf<- leaflet(SD.dm, options = leafletOptions(minZoom = 5)) %>%

Here we have the command "**leaflet**" that tells R to create a leaflet object. Leaflet is the piece that is directly responsible for the Webmaps. We are creating the leaflet out of our School Districts data map defined earlier and setting some options. Here, I have set a minimum zoom to keep the end user from being distracted by parts of the globe that don't have any data. Next I set where the map opens to and what the basemap is with these two lines of code:

```
setView(lng = -123.0317114, lat = 44.9384532, zoom = 7) %>%
addProviderTiles(providers$Esri.NatGeoWorldMap) %>%
```

Now I add the shapes of the school districts with the command "**addPolygons**." Within this command, we have definitions of how the polygons must behave. Here we have:

```
addPolygons(
        fillColor= ~pal(Grad_Rate),
        weight = 1,
        opacity = 1,
        color = 'white',
        dashArray = "3",
        fillOpacity = 0.7,
        highlight = highlightOptions(
                weight = 5,
                color = "#666",
                dashArray = "",
                fillOpacity = 0.7,
                bringToFront = FALSE),
```

First we fill the polygons based on the color palette we defined earlier based on the "**Grad_Rate**" variable. Next we set the weight, opacity, and color of the lines separating the polygons. I elected to create dashed lines, evident in the "**dashArray**" command. The opacity of the polygons is set in the "**fillOpacity**" command. Finally, I set the highlight and highlight options for when the mouse is over a particular area[4].

Finally, we can apply the labels and the legend. The labels command simply equates labels with those that we created earlier:

```
label = labels,
labelOptions = labelOptions(
        style = list("font-weight" = "normal", padding = "3px 8px"),
        textsize = "15px",
        direction = "auto")) %>%
```

The legend is defined by our palette, variable values, and its opacity is set. We add a title, define how to handle missing values, and set where the legend will be on the final piece. That code looks like this:

```
addLegend(pal = pal, values =~ Grad_Rate, opacity = 0.7,
        title = "Graduation Rates in the 2015-16 School Year",
        na.label = "No Data",
```

[4] Important to not here, on the highlight options, the "**bringToFront**" must be set to "**FALSE**." If it is not, the webmap will interact unfavorably if it is opened in Internet Explorer.

position = "bottomright")

Finally, we look at the leaflet created simply by calling its name:

SR.lf

Here is the complete piece in R: