# Advanced Data Collection

Sometimes it is just not the case that the data you need is in a nice, computer friendly form for you to either download or purchase. Additionally, you may need to have constant “live” access to data, or the data may live online, and you would like to avoid the steps of downloading, cleaning, and processing each time you use it. In these cases, there are methods you can utilize to obtain this data using various computer languages. We are going to focus on how we can use R to scrape data from the Web and utilize APIs. I should also note that R is also capable of “reading” PDF files using optical character recognition, a tool I used to process about 10 years’ worth of tax books for Saint Louis County with each book consisting of about 100 pages. Rather than typing each page in individually, I was able to scan the pages as PDF and then have packages in R process them and turn the PDF tables into actual tables in R. Since time is an issue, we are going to focus on using the two more common techniques, web scraping and APIs.

## Website Scraping

Websites are nothing more than GUIs for HTML code and the information shown on a website MUST be in computer form somewhere. Website scraping takes advantage of this and instructs the software (here R) to read through the HTML code, find the desired data, and extract that. What is nice about this method is it is very helpful if you need data that may be located on different pages. Hopefully this will make more since as we move through the example.

To get started, we need to install a couple of elements that will help us as we go. First, install the Java Development Kit (JDK). This will ensure that Java is available on your machine for what we need to do. Secondly, install the firefox web browser. It is free and works better with the library we will be using called RSelenium

What this package allows you to do is actually open up the website you are going to be scraping and

#### Setting up R Script

#### *[library rvest]*

The first thing we need to do it obtain the correct package for reading and translating HTML into something we can use for statistics. The best way to find the state-of-the-art is to do a simple Google search and find the method that will work for you. Most of this lecture is based on work I have done using the guide found here.[[1]](#footnote-1) As you can see by reading the tutorial, we will need to install the R package *rvest* to set up our scraping script.

Before we can learn to scrape a webpage, we must understand the webpage. Webpages are just another form of programing code, and a browser is simply a compiler. The language of the internet is HTML and just like any other language, it has elements we can exploit to get what we want from a website. Specifically, everything in HTML is tagged. Common tags are <body>…</body> indicating that everything between the two tags is the “body” of the website. Common tags that we will typically utilize are <table>…</table> which are the tags used to indicate that a table of something is present.

The largest challenge to web scraping is to find the right tag that you need and then direct your “crawler” to the right spot. Sometimes you can also exploit the attributes within tags. These includes things like the font size, language, or other attributes used to determine the appearance of the information within the tag on the web browser.

To see how this all works, let us set up a script, saved in the Build -> Code directory called **691ClassTwo.R** and set it up as we have before. We should be working in our ECON691 project in R so that we can commit the initial script set up to Git before we go too much further. We need install *rvest* and load in the libraries for *rvest* and *tidyverse*.

#### Finding the Data

#This is a script to download the election results by county for the

#2016 US Presidential Election, download Census data via API, and use

#ggplot to create state maps

#Created by J. R. Groves on October 2, 2022

rm(list=ls())

library(rvest) #rvest is used to scrape the New York Times website for the needed data.

library(tidyverse)

We want to get the county level results from the 2020 U.S. Presidential Election and we will do so from Politico.com at the URL <https://www.politico.com/2020-election/results/president/>

To simplify our search, we will first focus on the State of Illinois, so we will scroll down to the bottom of the page and click on the Illinois link.

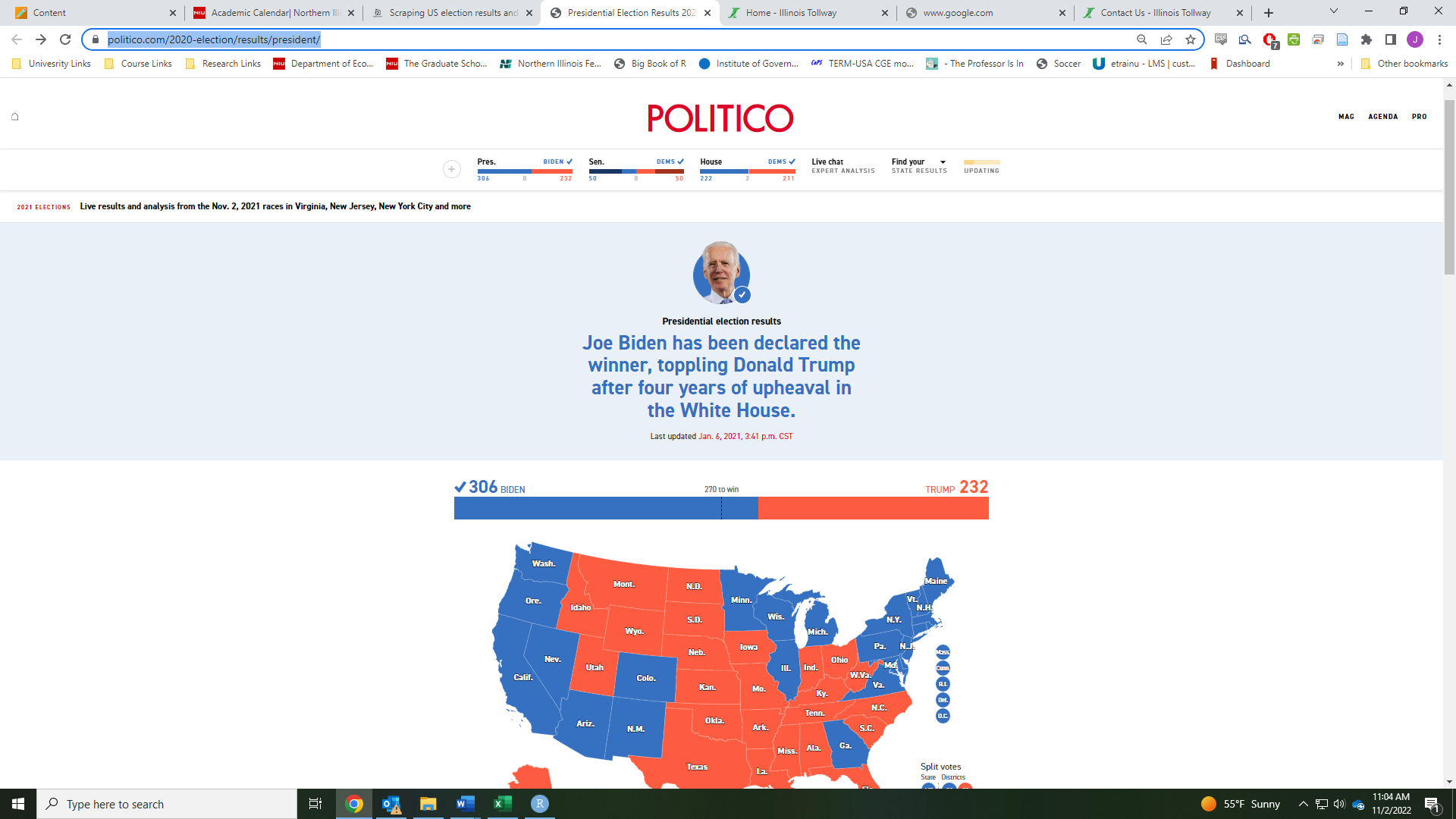


Figure 1: Image of the IL State Results Page on politico.com

On this page, if we move our mouse around the state, we get a popup of the results, but this does not help us get the results quickly from the webpage to a form we can use in R. What we need to do is find where in the web page this information is found so we know where to tell R to look. To do this, right click (in Chrome) and choose INSPECT and a small window will appear to the right with HTML code for this website. Notice that as you move your curser up and down this code, different parts of the website are highlighted. This tells us where in the HTML code that part of the site is located.

The problem with the map is that it is being driven by an additional script behind the scenes and is more complicated to pull data from. If we scroll down the page; however, we see there is a table of all the data we need. Once you have the table on the screen, move your mouse over the HTML code until the table is highlighted. At this point in the code, you will notice a gray triangle pointing to the right. Click this and the code will expand with additional lines of code below this. This triangle, when facing right, indicated intended or sub-code under that current heading and is used as a means of limiting the output you see on the screen. Keep clicking these triangles and moving your mouse of the revealed code until the table is the only thing highlighted.

At this point you should see the tag <table> which is exactly what we are looking for. So we want to tell R to go to this URL and find this table and then read the data from it.

#Specifying the url for desired website to be scraped

url <- "https://www.politico.com/2020-election/results/illinois/"

#Reading the HTML code from the website

webpage <- read\_html(url)

tables<-webpage %>%

html\_nodes("table") #This pulls out all the "table" nodes in the HTML code

By running this code, we get in our environment, three new objects. The first is the object **URL**, which is just the web page we are going to. The second is the object **webpage** which is a list of two items and is the entire HMTL code that produces the web page we have been looking at. Finally, we see the object **tables** which is a list of four items. By typing tables into the console, we see that it is a list object containing the HTML code for ALL the tables on the page. To help identify the table we want, we look into the attributes of the table, and we see something like <table class="jsx-3713440361 table"> so we can use that class ID to determine the table we want, which is table four on our list.

To have R read the HTML table, we use the command HTML\_table() and since we want table four in the list tables, our object will be tables[[4]].

#read html table into data frame. Note the need for double brackets

data <- html\_table(tables[[4]]

By using the double brackets, we are telling R to use the information within the fourth list inside of the main list. If we used single brackets, the resulting object would have been a list giving us the fourth list of the main list. We would have to translate this into a dataframe, but by using the double brackets, it automatically puts it into a data frame.

Looking out the output we notice that there is a bit of an error in how the data is coming out and translated so we need to do a bit of cleaning.

> head(data)

# A tibble: 6 × 5

County `Biden votes` `Biden pct` `Trump votes` `Trump pct`

<chr> <chr> <chr> <lgl> <lgl>

1 Adams County 25.7%8,633 72.2%24,220 NA NA

2 Alexander County 42.6%1,114 56.8%1,486 NA NA

3 Bond County 28.1%2,288 69.1%5,625 NA NA

4 Boone County 42.2%10,542 55.6%13,883 NA NA

5 Brown County 19.3%486 76.6%1,931 NA NA

6 Bureau County 38.3%6,669 59.7%10,411 NA NA

Before we move on, notice in the last bit of the code we see a new operator **%>%** being used. This is called a pipe and is a way to simplify code a bit. In this case it is telling R to start with the object web page and then apply to it the function HTML\_nodes(). We could then keep adding commands below that where the results from the previous line are “pipped” into the function on the next line. In short, this is a different way to write nested commands in a form that, especially for commands with LOTS of parentheses, might get confusing. We could accomplish the same thing by using the nested command tables2<-HTML\_nodes(web page,"table"). In the code chunk below, we do both the nested and piped version of the steps necessary to pull out the second table and turn it into a dataframe.

Notice the fill=TRUE option in the command. This is used to ensure that if the vectors do not have the same number of observations, NAs are inserted to avoid error messages. Now we just need to do a bit of cleanup and we have a nice, workable dataframe with the county level election results. We want to remove the row names, change the name of the first vector in the dataframe and, because we are going to nest this process into a loop, we want to change the name of the object with our data in it. Each of those tasks are completed in the code below.

> head(results)

Vote.by.county Clinton Trump

1 Cook 1,611,946 453,287

2 DuPage 228,622 166,415

3 Will 151,927 132,720

4 Lake 171,095 109,767

5 Kane 103,665 82,734

6 McHenry 60,803 71,612

We want to clean up the data so we will use some of our tools from the *tidyverse* package. Specifically, we are going to change the name of the first column and remove the commas from the vote tallies and turn them into numeric characters, and then calculate the percentage of the vote carried by either candidate.

Illinois<-results2 %>%

rename("County" = "Vote.by.county") %>%

mutate("Clinton" = as.numeric(gsub(",","",Clinton)),

"Trump" = as.numeric(gsub(",","",Trump)),

"pctClinton" = (Clinton)/(Clinton+Trump),

"pctTrump" = Trump/(Clinton+Trump))

We want to do this same process for the states that boarder the State of Illinois. To accomplish this, we are going to use a loop. One needs to be careful using loops because, in some cases there are more efficient means to accomplish the same tasks and loops can sometimes get stuck or produce unexpected errors. Since our job is straight forward, we will use a simple loop.

The loop command for R is for(i in x){} where *i* is the current element of the loop within the set or sequence *x*. The curly brackets act as a container for the code that is to be carried out within the current loop. We can nest loops within loops, if need be, simply by adding another for() command within our existing command. The one key element to remember, however, is that just like for functions, the closing curly bracket **must be on a line by itself**.

For our case we want our elements to be the different state names, so we create a character object with the names we need. These names must be in the correct format as our URL which means we want to use lower case. Additionally, if we had a state with a space in the name, we would want to see how the web page references this in the URL and in the case of our example, the space is replaced with a dash. We will use this “variable” part of the URL with the “static” part by defining the static part to the object “**url1**” and then utilize the paste0() command.

This command is like the excel command concatenate as it allows us to paste parts of strings together to make a larger string. This paste0() is actually a sub-command of the more general paste() command where the former assumes there is nothing that is being used to separate the strings (such as a comma or a dash) while the latter requires that you define the delineator for the new string (we could use the latter and define the delineator as “” or nothing). Since we want R to paste in each of the state names, we paste the static URL with *i*.

#List of the states the data will be pulled for states

states<-c("kentucky","indiana","illinois","missouri","wisconsin","iowa")

for(i in states){

#Specifying the URL for desired website to be scraped

url.1 <- "https://www.nytimes.com/elections/2016/results/"

url<-paste0(url.1,i)

webpage <- read\_html(url)

tables<-webpage %>%

html\_nodes("table") #This pulls out all the "table" nodes in the HTML code

results2<-tables[2] %>%

html\_table(fill=TRUE,header=TRUE) %>%

as.data.frame() %>%

rename("County" = "Vote.by.county") %>%

mutate("Clinton" = as.numeric(gsub(",","",Clinton)),

"Trump" = as.numeric(gsub(",","",Trump)),

"pctClinton" = (Clinton)/(Clinton+Trump),

"pctTrump" = Trump/(Clinton+Trump))

assign(i,results2)

}

The next part of the code cleans up the data just as we did before and then we want to save the data as the name of the state. This can be tricking in a loop because if we assigned the output of our loop to *i*, then R would think we have an object named *i* we want to assign something to. To resolve this problem we use the assign() command, which is what the <- is really a shortcut for, to assign the value of our cleaned up data to the object *i*.

NOTE: If we had to run the loop of a series of numbers, rather than defining our loop over the text string as we do here, we would assign it over the sequence of numbers using the command seq(x,y,z) where the x is the start value, y is the end value, and z is the size of the step between values.

## APIs

Another way to get data from the internet is to hope that the data provider has created an API. This acronym stands for Application Programming Interface (API) which are user interfaces for other computer programs rather than for a human. They allow other computer programs to call up a server and ask for and receive specific data provided by the owner of the API. The benefit of this is that when writing code, a data analyst or researcher does not need to download the code and put it somewhere, rather, they can build into the code the act of downloading the data. With regards to the idea of replicable research, this is ideal because the researcher does not need to keep the raw data (which could be quite large) somewhere accessible to others. The downside to this, however, is if the API is down or no longer serviced, the raw data could be lost for good. Therefore, it is a good practice to have a backup available.

Other places we might see APIs used is in the form of data Dashboards we might find on the internet or interactive data analysis websites. I suppose, as a colleague of mine did, you could create an R program that utilizes the FRED API to do cool party tricks (and they say economists don’t know fun!). Although since shaving his head and he looks more like Lex Luthor, there may be other nefarious reasons for this.

Different APIs provide different data and may have different means of accessing. Typically, the owner will provide documentation on how to interface with their API in various programs and if your program is not listed, I am sure you can find an answer on the internet. Most APIs use the JSON data format which stands for JavaScript Object Notation. This is an alternative to a simple .csv data file because it allows for more detailed notation and naming and makes, as we saw with the scraping exercise, finding specific data a bit easier because everything as a “tag” or “node” attached to it. Either data type can be handled by R quit easily.

#### U.S. Census Bureau API

One of the most frequently used data sources in economics is the U.S. Census and the man y other surveys and datasets they maintain. As a result, the U.S. Census Bureau has their own API and there are a couple of R packages that are specifically designed to make navigating this API simple.

Before you can access the census AIP, you must first have a key. A key allows the host of an API to track its use and authentic everyone who accesses the data. In the case of the U.S. Census, users must have a key, but a key is free to obtain by filling out the form when you click the button you can see on this web page (<https://www.census.gov/data/developers/data-sets.html>).

The key is a string of numbers and characters that uniquely identifies you and you will need it anytime you want to utilize the API. To do this, you have two options. You can save it somewhere (I have the emailed saved) and then put the key in the script where you are accessing the API or, in the case of R, you can save the API as part of the initial boot-up information for the program but to do this we first need to load the package we are going use: *tidycensus*.

After you have installed the package and loaded it into the environment via the library() command, we will use the census\_api\_key() function. In the first element of this command, we paste the key as a character string (so inside of quotation marks). The second element is a TRUE/FALSE element asking if we want to overwrite any existing keys and the default is TRUE so we can skip this. The last element asks if we want to install this key into the *.Renviron* so we no longer need to and we want to so we will use the install=TRUE option.

The Census has a vast repository of data, and you want to be very careful when choosing the variables to use as some may not be available for all years or may be inconsistent from year to year. The Census website does a good job of making documentation obvious to find so you will want to make sure you are familiar with the dataset you are using. We will use the American Community Survey, 5-year files which, according to the *tidycensus* documentation, this is the acs5 data.

To obtain a list of what we can pick within the *acs5* data, use the following command v<-load\_variables(2016,"acs5"). Upon viewing the object **v** we see a list of ALL the data we can get from the 2016 *acs5* API along with the variable names (which we will need). The unfortunate use of this is that there are more than 20,000 variables and so it can be quite time consuming to comb through them all. We can use the tools in R to parse the list using keywords or, alternatively, go to the ACS web page and scroll down to the variable list for the year you need and scroll through this list.

library(tidycensus)

vars<-c("B01001\_001","B01001\_002","B02001\_001","B02001\_002",

"B02001\_003","B05001\_001","B05001\_006","B07001\_001",

"B07001\_017","B07001\_033","B07001\_049","B07001\_065","B07001\_081”)

#Command to pull data from ACS

acs <- get\_acs(geography = "county", #defines geography level of data

variables = vars, #specifics the data we want

state = 17, #denotes the specific state

year = 2016, #denotes the year

geometry = TRUE) #downloads the TIGER shapefile data

An alternative is to go to the Census.gov page (<https://www.census.gov/data/developers/data-sets/acs-5year.html>) and find the ACS information pages where you can find various ways to find the data you need. Scroll down this page and we see a heading for Detailed tables and we can click on the html link for the Detailed Tables Variables.

The first option is defining the geography of the data we want, in this case we want county level data, so we use “county”. We could obtain state level, block group level, census tract level, and several others. For the list of the available geographies, just check the ACS website. Also notice that I stack my code here by hitting return after each comma in the options part of the command. This helps to see what I am looking for and makes this command easier to read. I highly recommend it for code with lots of option settings.

Next, we list our variables and in this case I have defined an object *vars* to contain the list of the variable IDs I am wanting to download. You can see that just above the get\_acs() code. Now I must tell the API which state or states I want the data from so I use the FIPS code for Illinois, which is ‘17’. Each geography level in the census data has a unique FIPS code and I would need that code if I wanted specific counties or census tracts within a given county. The next option is for the year or years of data I want and finally, I have an option called geometry which is a TRUE/FALSE option that tells the API whether you would like the necessary data to draw a map (or output a shapefile) with the data. Once this runs, we can use the head() command to see what the data looks like.

> head(acs)

Simple feature collection with 6 features and 5 fields

Geometry type: MULTIPOLYGON

Dimension: XY

Bounding box: xmin: -89.51839 ymin: 36.9703 xmax: -89.13268 ymax: 37.33525

Geodetic CRS: NAD83

GEOID NAME variable estimate moe geometry

1 17003 Alexander County, Illinois B01001\_001 7051 NA MULTIPOLYGON (((-89.51839 3...

2 17003 Alexander County, Illinois B01001\_002 3462 60 MULTIPOLYGON (((-89.51839 3...

3 17003 Alexander County, Illinois B02001\_001 7051 NA MULTIPOLYGON (((-89.51839 3...

4 17003 Alexander County, Illinois B02001\_002 4365 39 MULTIPOLYGON (((-89.51839 3...

5 17003 Alexander County, Illinois B02001\_003 2388 90 MULTIPOLYGON (((-89.51839 3...

6 17003 Alexander County, Illinois B05001\_001 7051 NA MULTIPOLYGON (((-89.51839 3...

Notice the very last column of this dataframe, the geometry; this is what we will use when we plot the data and having this in the dataframe makes this a different type of dataframe called a Simple Feature Collection and we can do things with this using the sf package but we will not need to do that much with this so we will not worry about installing this package.

We see the vector **GEIOD** which is where the FIPS code is located and we see the 17 for the state and, in the case of Adams County, the 001. We see the county name, the variable name, the estimate, and margin of error (**moe**) as well. This data is current in what is called “Long Form” because there is a line for each county-data pair. I would like to have this in “Wide Form” where there is a line for each county and a column for each data.

There are several ways to reshape the data but for us we will use is via the *tidyverse* package which is a set of tools for cleaning data. Within this package we want to spread() the data after we also drop the margin of error data. We also want to change the variable names and there are a few ways we can do this using the *tidyverse* package. The safest means is to use the mutate() command with the case\_when(). The other option is to spread the data and then use the names() command to just rename the column names of the dataframe. The risk with this latter method, while easier, is dependent on the columns staying in the same order. Using the case\_when() command avoid this.

il.acs<-acs %>%

mutate(variable2 = case\_when(variable=="B01001\_001" ~ "TotPop",

variable=="B01001\_002" ~ "Male",

variable=="B02001\_001" ~ "TotRace",

variable=="B02001\_002" ~ "White",

variable=="B02001\_003" ~ "Black",

variable=="B05001\_001" ~ "TotCit",

variable=="B05001\_006" ~ "NonCit",

variable=="B07001\_001" ~ "TotMob",

variable=="B07001\_017" ~ "Stay",

variable=="B07001\_033" ~ "SameCounty",

variable=="B07001\_049" ~ "SameSt",

variable=="B07001\_065" ~ "OthState",

variable=="B07001\_081" ~ "Abroad",

TRUE ~ "other")) %>%

select(!c(moe,variable)) %>%

spread(key=variable2, value=estimate)

The case\_when() command allows us to create a variable based on the logical result of another variable. In the code above we are looking to the “variable” and seeing when it is exactly equal (==) to various values and telling R that whenever this condition is true, then to assign our new variable (variable2) the value to the right of the squiggle line. The very last line of this case\_when() command covers all of the cases where none of the above conditions are met and is a good way to make sure you do not have any coding errors in your data.

The select() command is part of the *tidyverse* package that allows me to pick which columns I want to be carried over into my new dataframe. Using the ‘!’ as a command switch that tells R ‘not’ we see that we are telling R to choose the columns that are NOT moe and variable which essentially deletes these columns. The spread command is used to turn long data into wide data and the first option shown here tells R where to find the different columns names of the wide data and the value option tells R where to find the element to put in each column.

Raw numbers do little for us so we want to create a set of percentages and we can do this by simply adding a mutate() command with a series of variable definitions to the above set of commands. The code we add is shown below.

mutate(perMale = Male/TotPop,

perWhite = White/TotPop,

perBlack = Black/TotPop,

perCit = 1-(NonCit/TotCit),

perStay = Stay/TotMob,

perSameCounty = SameCounty/TotMob,

perSameSt = SameSt/TotMob,

perOthState = OthState/TotMob,

perAbroad = Abroad/TotMob) %>%

select("GEOID",starts\_with("per"),"geometry")

An advantage of doing this type of work within the mutate() command as part of a pipe is that we do not have to keep identifying the dataframe we are talking about. Without doing such modifications within this framework, we would have to add ‘acs$’ before each variable to which we are referring. Also notice at the end of the mutate command we want to keep only a couple of columns and all our percentage columns. Since we named (purposefully) all of our percentage columns with the prefix “per”, we can use the select command with the subcommand starts\_with() to include all of the variables that start with the same prefix. Other versions of this command can be found by searching the help for starts\_with.

Also notice that we are keeping the geometry column. This is so we can draw nice graphs to display our data and, more specifically, it is keeping our dataframe as an sf object. We can change that by simply removing the geometry variables with the command df$geometry=NULL.

#### Other APIs

If we want to access other APIs that do not have built in packages in R, we can follow along with the tutorial here (h[ttps://www.dataquest.io/blog/r-api-tutorial/).](https://www.dataquest.io/blog/r-api-tutorial/) In short, we will use the packages *httr* and *jsonlite* and the key commands are going to be GET() and fromJSON() which will get the data and convert it from the JSON format we spoke on earlier into a dataframe.

As an example, we are going to access the historical Covid data kept by the CDC at their API. (<https://data.cdc.gov/Case-Surveillance/COVID-19-Case-Surveillance-Public-Use-Data-with-Ge/n8mc-b4w4>). In the upper-right corner we see a button API and if we click this, we get a popup window telling us a little about the API and, most importantly, giving us the end point. We copy that end point and paste that inside the GET() command and save the data to an object called covid.

library(httr)

library(jsonlite)

covid<-GET("https://data.cdc.gov/resource/n8mc-b4w4.json")

head(covid$content)

If we look at what our API query just produced, we see something that is rather confusing because it is in raw UNICODE. We need to convert this to JSON and then into a dataframe using the nested command fromJSON(rawToChar()) which will convert UNICODE to JSON and then JSON to Dataframe.

And now we have a full data of all Covid cases in the United States (all 27.5 million-ish) with data on their location, age group, and some other information about them. This data would have been very difficult to get any other way and we can consistently update the data so long as the CDC maintains the API.

covid.2<-fromJSON(rawToChar(covid$content))

head(covid.2)

## 

## Displaying Data and Results - ggplot2

So far, we have created two dataframes that show us the results of the 2016 Presidential Elections in several states by county and we have some county-level demographics. The challenge we have now is displaying this data in a form that is understandable and accessible. We also would like to “kick it up a notch” from the simple plot() command form last week so we are going to use the go-to graphics package in R, *ggplot2* which is included in the *tidyverse* package.

library(ggplot2)

ggplot(il.acs) +

geom\_sf(aes(fill = perMale))

Chart

Description automatically generated

Here we have a nice simple map of the how the percentage of the population that is male is distributed across the State of Illinois by county. What makes this work is that the *ggplot2* package is based on a set of geomsdepending on what you want to graph. For example, a line graph would have the geom of ‘line’ and, in our case, since we have an sf-object (because we downloaded the geometry data from the Census), *ggplot2* knows that this is typically geographic data if we use the geom\_sf command.

The *ggplot2* package is so massive, there is no way to cover everything you can do it with here, so I am going to focus on just a few key elements. More information can be found by simple Google searching *ggplot2* or via the cheat sheet on Blackboard.

The syntax of *ggplot* is based on “aesthetics” and “layers”. The first part of the command is the ggplot() command which you can think of as setting the ‘global’ parameters of this graphic we are creating. Typically, we would express the source of the data in this step if it is from a single dataframe. If the graphic is a simple one, such as a line graph or even the one element graphic we just created, we could put all the relevant information in this command as such ggplot(acs3,aes(fill=perMale)) which tells the package that the data is found in dataframe **acs3**and that we want to fill the polygons based on the values found in the ***perMale***column. This by itself, however, is still not enough because we have to tell *ggplot2* what kind of graph to make and so we add a plus sign at the end of the line + (this is similar to the pipe we used earlier) and then define the geom as geom\_sf(). You can see in the original code chunk that we saved our fill command for this line because it was specific to this particular “layer” of the graphic.

Let’s do something slightly different and plot a graph with the percentage of males and the percentage of whites living within each county. I can do this on one graph by adding layers and then changing the aesthetics within each of the specific layers. NOTICE: I add a new variable called num so that I can simulate an observation number and provide *ggplot2* with a sense of order for the values.

il.acs$num<-seq(1:102)

ggplot(il.acs, aes(x=num))+

geom\_line(aes(y=perMale), color="red")+

geom\_point(aes(y=perWhite), color="blue")

Chart, scatter chart

Description automatically generated

I can now clean up the various elements of the graph by using the layers that correspond those elements such as the x-label, y-label, title, and legend.

ggplot(il.acs, aes(x=num))+

geom\_line(aes(y=perMale, color="red"))+

geom\_point(aes(y=perWhite, color="blue"))+

xlab("County")+

ylab("Percentage")+

ggtitle("Percent of Population by County in Illinois")+

scale\_color\_manual(name="Percent of",

breaks = c("red","blue"),

values = c("red","blue"),

labels = c("Male","White"))

Chart, scatter chart

Description automatically generated

Unfortunately, our voter data is not in an **sf** object we need to link it to some type of sf object such as our ACS data. We can link these data with the County Before we merge the data, however, we must make sure that our common element is, in fact, common and identical. A quick look at the list of county names in our voter data and the list of county names in our ACS data quickly shows at least one problem and that is the inclusion of the text “County, Illinois” in the ACS data. We can quickly get rid of this by using the gsub() command we used to remove commas. The other issue we see is that the counties are not sorted the same way, so we need to fix that as well.

We use the gsub() to substitute the pattern " County, Illinois" with nothing in the acs3$NAME object and we nest that command inside the trimws() command with trims any leading or trailing spaces. Next we want to sort or order the dataframes and so we use the order() command. Notice that we use the dataframe we are going to order, and then open bracket followed by command and inside the command we defined the vectors we base our sorting on. The comma at the end of the order() command tells R that we are ordering the rows of data rather than columns.

#Remove the added text in ACS data

il.acs$County<-trimws(gsub(" County, Illinois","",il.acs$Name))

#Sort both data so they have the same sorting process

il.acs<-il.acs[order(acs3$County),]

illinois<-illinois[order(illinois$County),]

#Logic test to make sure the names match

il.acs$County==illinois$County

> il.acs$County==illinois$County

[1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE

[17] TRUE TRUE FALSE FALSE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE

[33] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE FALSE TRUE TRUE TRUE TRUE TRUE

[49] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE

[65] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE

[81] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE

[97] TRUE TRUE TRUE TRUE TRUE TRUE NA

Now we want to see how close we are to matching up using a simple logical request. We do see that most are TRUE meaning they do match, one-to-one, but that a couple are FALSE. To see why, let’s pull up the first FALSE which occurs at points nineteen and twenty.

il.acs$County[19:20]

illinois$County[19:20]

il.acs$County[19]<-illinois$County[20]

il.acs$County[43]<-illinois$County[43]

Ah, one data set put a space between the e and W while another did not. We must be careful in fixing this, however, because it did impact the overall sort, but we just need to replace the “De Witt” with “DeWitt” as we do below. We will see that a similar occurrence happened at 43 with Jo Davis County.

Now that the data is fixed, we reorder and check again, and there is a match so now we know we can merge. We could have also found these by doing the merge and then see what did not match up. Either method works fine. To carry out the merge we use the merge() command which instructs R to merge an **x** dataframe with a **y** dataframe by some common element. NOTE: You can only merge two dataframes within the merge command; however, if you have multiple dataframes with a common element you wish to merge you can use the command

Reduce(function(x, y) merge(x, y, all=TRUE), list(df1, df2, df3))

where the list contains all the dataframes you wish to merge.

il.acs<-merge(il.acs,illinois,by="County",all=TRUE)

b<-il.acs[is.na(il.acs$perMale),]

il.acs<-il.acs[!is.na(il.acs$perMale),]

In our merge command we merge the *il.acs* and *illinois* dataframes by the common element “County” and we say we want to keep all the element whether they match or not. This helps us found why we have one extra observations in one of the data and come to find out it was an **NA** that was in the raw data. We could have merge with a common element that had different names by replace the by= with by.x=, by.y= and we can use the c() to merge by more than one common element. The all= is defaulted as FALSE meaning that any non-matched items are dropped. We could also set this to keep all the unmatched observations from x or y by using all.x or all.y. The next two lines of code in the above chunk is us seeing what did not merge because the missing fields will have been filled with NAs and so the isna() tells R to find those NAs. We see that the reason that there was one more observation in the ACS data was that there was a field, likely a blank row, in the original data that caused a row of NAs. By using the ‘not’ operator we are able to change finding NAs to finding non-NAs and use the isna() command again to keep only the observations with full records in the Total Population vector.

Now we can plot our map of Illinois to show how well, or poorly, Candidate Clinton did in the 2016 Presidential Election.

ggplot(il.acs)+

geom\_sf(aes(fill = pctClinton))+

scale\_fill\_gradient(low="white",high="blue",limits=c(0,1),aes(name="Percent Clinton"))

Chart

Description automatically generated

We can use the options within *ggplot2* to change the color scale if we so desire and to drop the latitude and longitude markings along the axis and remove the entire plot area background since we are really doing a map here rather than a graph.

Since race appears to have had some impact on the election in 2016 let’s see if there is any visual correlation between the percentage of whites in a county and the percentage of votes candidate Clinton received. To do this we will download a package called *cowplot* to utilize its ability to put multiple graphs on a single page. Additionally, we are going to assign our graphs to an object so that we can recall them at any time. The script to produce the two graphs with the plot area cleaned up is below.

library(cowplot)

p1<-ggplot(il.acs)+

geom\_sf(aes(fill = pctClinton))+

scale\_fill\_gradient(low="white",high="blue",limits=c(0,1),aes(name="Percent Clinton"))+

theme(panel.grid.major=element\_blank(),

panel.grid.minor=element\_blank(),

panel.background=element\_blank(),

axis.text.x=element\_blank(),

axis.text.y=element\_blank(),

axis.line = element\_blank(),

axis.ticks = element\_blank())

p2<-ggplot(il.acs)+

geom\_sf(aes(fill = perWhite))+

scale\_fill\_gradient(low="black",high="white",limits=c(0,1),aes(name="Percent White"))+

theme(panel.grid.major = element\_blank(),

panel.grid.minor = element\_blank(),

panel.background = element\_blank(),

axis.text.x=element\_blank(),

axis.text.y = element\_blank(),

axis.line = element\_blank(),

axis.ticks = element\_blank())

plot\_grid(p1,p2)

Map

Description automatically generated

Notice that RS does not actually show us the graphs when we assign them to object until we call for that object. Sure enough, we can see a visual correlation between race and votes for Clinton with areas that have larger percentages of non-Whites (blackish) are the same counties with more votes for Clinton (darker blue in the left map). To show this is a factor, however, we would want to regress one on the other and control for other factors other than race which will be the topic of next week’s class.

# List of Packages

Below is a list of the packages used and/or referenced in this lecture:

rvest

cowplot

ggplot2

tidycensus

tidyverse

stringr *referenced but not used*

sf *referenced but not used*

1. https://www.analyticsvidhya.com/blog/2017/03/beginners-guide-on-web-scraping-in-r-using-rvest-with-hands-on-

   knowledge/ [↑](#footnote-ref-1)