1. Introduction and Objectives

After this section of the course, you should be able to understand how to utilize version control in your work, how to employ R and RStuido to create replicable research and to use various commands within R to perform data collection, cleaning, analysis, and presentations. All the software used in this section is open source and free to download; however, in some cases you may need to ensure the software is compatible with your operating system (OS).

Research that can be easily replicated, as you can image, is research that is sufficiently documented so that it can easily be replicated by anyone else including yourself several days, months, or years later. Included in this is the availability of the data, any code used to compile the data, and presentation materials. This is becoming more standard with many journals requiring access to at least some aspects of the research (data, code, ect.) especially considering the social science replication crisis recently brought to light. In this project, researchers found that many of the results that have been published in peer reviewed journals were not replicable given the information and data provided by the original authors. Given that one of the hallmarks of "good science" is the ability to replicate findings, this can be rather problematic.

As part of this idea of replication, version control is a means to keeping track of what "version" of a project you are working on. Inevitably you will be forced at some time to step away from a project for a time and will find that upon your return, you do not recall where you were at when you left. Additionally, you will often be "struck" with an idea either lying awake at night or in the morning shower that you want to try. Version control allows you to keep a record of what changes you make in your work and why allowing you to jump right back into a project and allows you to branch off with new ideas that can either be folded back into your original flow of research or discarded completely without risk to your previous work.

The software we will be using in this first section of the course are R for data and statistical analysis and Git for version control. The software R-Studio is going to act as our interface that brings each of these elements together. Another nice feature of R-Studio is that is also allows incorporation of other software that you will be learning in this course including Python and LaTeX (or *knitr* to create R-Markdown file). With that said, let's get our laptops ready.

* 1. Computer and Software Needs

First and foremost, make certain that you have a laptop that is fully up-to-date and able to access the internet either directly or wirelessly. In terms of operating systems, any should do, and the preference is going to depend on the instructor and their own history. Myself, I prefer Windows mostly because that is what I have always used and understand. That does not mean you cannot use Lynx or iOS, just know that I have little to no knowledge and so you may need to rely on fellow students or Google for troubleshooting. Most of the software has versions for any major OS, but there will come times when you run into problems with programs producing unexpected errors.

In these cases, your most valuable research tool is going to be Google. I have found that simply doing a Google search of the error message many times will produce a blog post or discussion forum post with the solutions. This also holds true for problem solving any research question. The one source that is constantly popping up in my own troubleshooting searches is Stock Overflow (https://stackoverflow.com/). This forum covers pretty much ANYTHING that involves coding and I have found that many of my questions have already been answered and, in several cases, those answers have been peer reviewed and either verified or improved.

* 1. GIT and GITHUB

The first specialized software is our version control software and a place to put our research so that we and others can access it easily from wherever whenever we need. While there are several different options for each of these needs, we will use Git and GitHub.

GIT is a version control software that can be downloaded for any operating system via their web page (https://git-scm.com/downloads). Because we are going to mainly be using Git through R-Studio, we are only going to concern ourselves with the "command line" version. What this means is that we will be using the command line interface (rather a GUI or Graphic User Interface such as Windows) via the command prompt on your computer. We will discuss accessing your command prompt more when we set up and use Git for the first time. Just know that for now, you will NOT need to install a user-interface other than what comes with the install package.

So that we have a place to store our information, we will be using GitHub so go to their website (https://github.com/) and set up an account. Since we will be using this account at NIU, please set this up using your NIU email address. This is also important because you will be sharing data with me via this site as part of your assignments. Once you have an account, you may download and install the GitHub desktop interface (it is another GUI for Git), although as I discussed above, we will likely NOT be using it.

* 1. Notepad++

While word processors are nice, there are times when we do not want to have all the extra baggage that comes with the word processor. For example, good raw data is typically in a *.txt* or *.csv* file because of their flexibility and universality. To that end, it is good to have a good text editor on board and the Notepad++ is a very flexible and powerful text editor that is also open source. You can download it via their website (<https://notepad-plus-plus.org/downloads/>) and you can also find several plugins that you can install to enhance the core product. This is also a good source that will create a universal text file that you can use to log any steps you make to create and/or clean your data.

* 1. R and RStudio

You can think of R-Studio as being your way of interacting with R and several other software packages to accomplish each of the objects outlined above. Later in this course you will learn LaTeX and Python and you can use R-Studio to interact with those as well.

First, we need to install the core of what we will be using; R. Use this link if you have Windows (https://cran.r-project.org/bin/windows/base/) and use this link if you have iOS (https://cran.r-project.org/bin/macosx/). Once you have installed R, we will move on to install R-Studio which we will use to interact with R. This link (<https://rstudio.com/products/rstudio/download/>) will take you to the download page where you will download and install R-Studio Desktop for the appropriate OS.

* + 1. Integrating Git with RStudio and Git Setup

*[Packages: usethis]*

The first step once you have installed Git it to tell it who you are. To make this easiest, we will do this through RStudio but first we need to integrate Git into RStudio. To do this, open RStudio and then direct your mouse to the menu bar at the top and choose TOOLS à Global Options. This will open a new popup window and you look down the left-side choice and choose Git/SVN. Clicking this will open on the right-hand side a set of options where you will need to put in the location of the Git .exe file. On a Windows machine this is likely located at ‘C:/Program Files/Git/bin/git.exe’. If you need to, you can choose the “BROWSE” option to locate the file. Before you can see the GIT tab, you have to start a version controlled project.

Once you have completed this, select OKAY and then shut down and reopen RStudio. Now in the upper-righthand side window you should see a series of tabs with names like “Environment”, “History”, “Connection” and so forth. Choose the “GIT” tab and in the menu in the window that opens for that tab, choose the MORE option (with a down arrow). In this drop-down menu, choose “SHELL” which will open a command window on your screen. We are going to use the command-line feature of GIT to tell is who we are by inputting the following two commands:

git config --global user.name "Jeremy R. Groves"

git config --global user.email "jgroves@niu.edu"

Right now, make sure you are using your NIU student email ([z########@students.niu.edu](mailto:z###)) for this email address, but you can always change it when you want.

The last thing we want to do is to set up a safe and secure communication between RStuido and Git using a Personal Access Token. To do this, install the package *usethis* in RStuido and then run the command usethis::create\_github\_token() [Note that this is a means of using a command in RStudio and also noting the package it is from, you can skip the usethis:: if you like]. On the screen that pops up you should be logged into Git and if not, log in and then scroll to the bottom and click **CREATE**.

On the new screen will be a code that was generated with a clipboard next to it. Click on the clipboard and then open Notepad++ and paste the code in a new sheet. Save this text file as R\_PAT anywhere on your drive. You are doing this because once you leave the screen on GitHub, you will NEVER see this code again. If you lose it, you will have to generate another. Once you have a copy saved somewhere, return to RStudio and run the command gitcreds::gitcreds\_set() and choose option 2 and paste the code you saved into the console in response to the question. Once you enter, you are all set. You will need to replicate this process on any other computer you wish to use RStudio and communicate with Git.

* + 1. Setting up Library Location

The software R is an open-source software and really works best as a framework within which you can install and use different packages or libraries. These are little software plugins that other users write and maintain to do specific tasks such as specialized regression analysis, web scraping, or data cleanup and management. Unfortunately, anytime that R is updated, it is designed to install in a new subdirectory based on the version number so all the previous packages that you used must be reinstalled. One way to avoid this is to create a user location for the package libraries and then when you update R, you need only refer here and “remind” R to look in the user library for the packages.

You can create this user library wherever you choose. I put mine in the My Documents folder under the subdirectory R. Go here (or wherever) and create a new directory called `userLibrary`.

#My Stuff

myPaths <- c("C:/Users/HP/Documents/R/userLibrary", .libPaths())

.libPaths(myPaths)

To tell R where to install and look for packages you need to change the `Rprofile` file that is typically located at `C:/Program Files/R/\*current install\*/library/base/R`. Once you find the `RProfile` file, right-click it and choose "Edit in Notepad++" (assuming you installed that). You will then scroll to the very end of the file and enter the following commands.

You then save (note, you may be asked to switch to administrator mode, but just agree) the file and then when you open RS, you can type the command `.libPaths()` and you should see your directory and two others. All your packages will be installed here and you only need to do this again when you update the base R software.

* 1. Setting up a PROJECT

RStudio has the added feature of trying to help you keep everything involved in a project in one place by using the PROJECT concept. A project is just that, it is a project that you are working on and if you use this right, everything you need is always going to found right there. We are going to combine this idea of a project with our version control so that we will have a backup of everything we do, and we will be able to access this project anywhere and from any computer connected to the internet with RStudio installed which also will allow collaborator access.

**STEP 1: Set up the GitHub placeholder**

Go to Github.com and log into your account. At this point you should be in your Dashboard and notice in the upper, left corner there is a heading called Repositories with a green button that has a PC image on it and the text “NEW”. A repository on GitHub is just that, it is like a safe in which it stores everything that you “commit” to the repository to keep track of your work and updates. You will create a new repository called ECON691XXX with the X denoting your initials. You should be able to choose the defaults in the dialog box but do make sure to choose the option that creates a ReadMe.txt file.

Once done, you will be on the web page for this repository, and it is from here that you will do things such as share. What you need to do is to copy the web address in the browser as we will need that in the next step.

**STEP 2: Create RStudio Project**

I like to have a PROJECTS directory on my personal computer under the My Documents folder where I keep all the R projects I am working on. You can create a similar folder there or somewhere else on your system; whatever fits your organization style. Next, open RStudio and in the upper-righthand of the main window you will see a dropdown button that has the blue RStudio icon and the text Projects (none). Click on this and choose New Project.

This will open a popup window with three options: New Directory, Existing Directory, Version Control. Choose the last of the three and then choose Git in the next screen. In the text box under Repository URL, paste the URL that you just copied from your web browser. Then, in the bottom box, choose the BROWSE option and navigate to your PROJECTS directory and choose this folder and click OKAY or OPEN. The middle box should autofill with the name of the repository ECON691XXX. Click Create Project and RStudio will “open” a new session and in the upper right corner you will see where it said Project (None) it will now say ECON691XXX.

When or if you move to a different computer, you will need to copy the URL from GitHub again and set up a new project in RStuido and then PULL the repository from GitHub onto the new computer to start work.

**STEP 3: Set up main directory structure.**

Again, remember that we are trying to ensure that we can maintain an organized workspace in which we know where all the files we need for a given project are located. As such, we want to have a standard and easily understood file structure within our project directory, so it is easy to find everything that we need. This is also helpful because we do not have to worry about inputting the right directory strings when we load data, but more on that in a bit.

In RStudio you should see the main window divided into two halves with the left half being the “console” which is your communication screen with the core R package. The right side is again divided in half with the top being where we found our GIT tab and the bottom also containing a window with several tabs. The tab that should be open is called FILES and it is showing what is in the project directory we just created (it should just show text file called ReadMe.txt). We are going to create four subdirectories inside of this folder to help organize our work and we do this by clicking on the New Folder button in the menu just under the tab name in the lower-right window. We will name this first folder “Build”. Repeat these steps three more times with the folder names “Analysis”, “Data”, and “Presentation”. This should be self-explanatory as to what we are going to put in each of these.

Here are some best practices when creating file names.

* First, try to avoid using "special characters" as these will cause problems when coding in many languages.
* Secondly, avoid spaces! It was a big deal when computers switch from a 32-bit to a 64-bit system because it allowed for use of spaces in file and folder names. Unfortunately, when coding, spaces can cause serious problems, so it is best to just avoid them. Rather, many coders suggest using CamelBacking (or CamelCase) when naming things which is the processes of simply using a capital letter for each new word.

If you double click on the “Build” folder, it will take you inside that folder where we want to create the directories “Code”, “Input”, “Output” and then click on the three dots next to the green arrow at the top of your directory list and it should take you back to the name directory. Once there, click on “Analysis” and create the same three subfolders. Again, this should be self-explanatory as to what we are going to put in each of these folders.

**STEP 4: Start the ReadMe.txt log**

If you navigate back to the core directory of the project, you should see a file called ReadMe.txt. If you double-click this, you should see the left side of the RStudio window split in two with the console being pushed down to the lower-left and the upper window being a text editor. Note that you can also edit the text in the ReadMe.txt file in Notepad ++.

This file is going to serve as a log for everything that you do. Pretend that you are writing your future self (or someone else) a log of the important steps that you are taking. For now, let’s just simply type in something like “Project for Dr. Groves’ part of Economics 691, Fall 2022”. You should see the name ReadMe.txt in the tab over the window where you type this. Under this tab you will see a row of icons. Find and click on the disk to save the file.

The reason we are keeping this log in a .txt file rather than a Word Doc or TeX file, is because we want to ensure as much universality as possible. A .txt. file is about as basic as one can get and it can be read on any computer today or any computer from my high school days (ah, the 386 systems...) making it essentially universal. This is important because if you used something like Microsoft Word, it can ONLY be opened by a using Word or something that can translate Word and there is no assurance Word will exist in twenty years. For example, if I ever found a disk with a paper I wrote in high school, I would need a Word-Perfect converter (of course I would also need a 1.5 or 5.25 floppy disk reader as well).

**STEP 5: Initial Commit**

You have now made a change in your project, and we want to make sure that we track all our changes with our version control. In RStuido, go to the upper-right window and click on the GIT tab and you will see a list of the directories you created and the ReadMe.txt file. To the extreme left you will see a check box and between that and the name of the file will be two yellow boxes. This is GIT’s way of telling you that these files exist but are not currently part of your repository. To accomplish this, we need either click the file name or the checkbox to the extreme left of the file name. this action is called *stagging* the file and it is telling Git what files we are going to commit.

After you have staged the ReadMe.txt file, we need to commit the file to the repository which we do by clicking on the “COMMIT” button. This will open a new window which will show us what we are committing (in the upper-left), what is in the file we have currently selected in the upper-left (in the lower part of the screen), and an empty box to the upper-right. There is nothing remarkable at this point in the lower screen; however, if we had previously committed the ReadMe.txt file and we updated the repository with a change, the changes (including deletions) would be noted in the window at the bottom.

The upper-left screen is for our notes regarding this commitment which may include what our goals where or a description of what we have changed so far. You do not need to put a date here or your name as this will be tracked by Git internally. Since this is the first one, I always put the note “Initial Commit” and then select the Commit button. A new popup will appear on the screen and once it is done, you can click CLOSE.

This step has committed our updated ReadMe.txt file to the ECON691XXX repository ON OUR PERSONAL COMPUTER. We want to make sure we can access this anywhere, so we need to PUSH this new repository to the GitHub repository, and we do this by clicking the PUSH button. This will compare what is on our laptop/desktop to what is on the cloud and update the cloud with any changes that have been committed. Note that the PULL command will compare the cloud to what is on your laptop/desktop and update that image so that it matches what is on the cloud.

**STEP 6: Exit RStudio**

Congratulations! You have completed your first task in RStudio and ensured that the changes are tracked via your version control and updated the cloud version of your repository. Now you or a collaborator can go anywhere with a computer that has RStudio and Git installed and PULL down this repository (the same way we created our project in RStudio) and work on the project. Now just close RStudio and when you do, you will be asked if you wish to save the environment. Generally, the answer is NO because you should have saved all your files already or at least have the code to replicate them. The dialog will also remind you if you have not saved any of your scripts or text file as well.

1. **Welcome to R and R-Studio**
   1. Basic Syntax of R

Like most programming languages, R has its own syntax and some of the common commands and ideas are covered here. R is an Object Orientated Language meaning that items within R are considered objects that have various values assigned to them. What is assigned to a given object depends on the type of object that it is and the most basic object in R is the vector.

Vectors can have two dimensions (length and width) and can hold various types of data ranging from numeric to character to dates. The only "catch" with vectors is that a given vector must contain the same TYPE of data so a numeric vector can only have numbers and a character vector can only have characters. A vector is referenced by its name or id and individual items within a given vector can be accessed using the location of that specific item within the given vector. To assign a value to a vector, R uses the `<-` which can be read as “is assigned the value of/object”. For example, if I want *x* to be assigned the value of 2, I type `x<-2`. You need to note that the dash is part of the syntax so if you are assigning a negative value you use `x<- -3`. If we want the object being assigned to a vector be to a character string (i.e., letters), we use the assign syntax as before and enclose the character string in quotation marks.

x<-"Hello, my name is Jeremy"

x

This command assigns the value of “Hello, my name is Jeremy” to the object ‘*x*’ and then places that object within the global environment. You can see what is in the global environment by going to the upper righthand window and choose the Environment tab. This will list all the objects and functions in the current global environment. Objects remain in the environment until they are changed or removed.

The workhorse in R is the dataframe. A dataframe is a collection of vectors and those vectors can be of any type. An additional feature is that row matters meaning that each element in location *i* in a given vector is related to the item in every other vector found at location *i*. As a result, all vectors within a dataframe must have the same length. Dataframes are essentially what you would expect to work with in Excel or STATA.

Each of the vectors within a dataframe do not necessarily also exist in the global environment as its own object unless you build the dataframe from a set of existing objects or create new objects using vectors from a dataframe. You can, however, call on any part of a dataframe using the “coordinates” of the item. As such, if you wish to refer to a specific element you can use the command *df*[x,y] where *df* stands for the name of the dataframe object, *x* is the column number and *y* is the row number within *x* that holds the element you are looking for. You can reference an entire row using the command *df*[,y] or an entire column using *df*[x,]. The vectors within a dataframe, because they are essentially objects, also have names and you can refer to a specific column via its name using the syntax *df*$VNAME. This method is preferred to using the column number because if the order is changed, then you may end up calling the wrong column of data. One challenge with this method, however, is if your variable name includes a space in the name. While I like toa void this whenever possible, it is sometimes forced on you. In these cases, you will use single quotation marks to denote that the space is NOT a space in the code. The syntax would then be df$’V NAME’.

Another data type you will utilize in R is the List. You can think of lists as containers holding a set of different dataframes/vectors that can be of different types and different lengths. Regression results are reported as a list containing separate “elements” for the vector of coefficient names, the dataframe of coefficients, the vector of standard errors, the vector with the F-Stat, the vector of residuals, among others. You can access elements of a list the same way you access elements of a dataframe. You can use the name of the item within the List using the syntax is `x$u` where `x` is the name of the list and `u` is the specific element. You can also use `x[[n]]` to pull the nth element within the list x. Lists can introduce problems in coding if you are not careful so make sure you fully understand what you are doing with lists before you utilize them. I, personally, avoid them at all costs but there are times when output in placed into lists so you at least need to know how to get data out.

* 1. Making R Work

Commands can be input into R in several ways with the two most common being directly through the console (command line) or via a script. Because we want to focus on reproducible research, we are going to use scripts as a means of keeping track of what we do to our data.

We create a script by clicking on the green plus sign in the upper-left window and choose R Script from the list. This will open our editor window on the left side, and we can start typing our script. Scripts are then saved with the *.R* extension. Good scripts are generally broken down into four key parts: intro and loading packages, defining functions, loading data, and commands. Once we have a script, we can run all or part of it using either the “run” or the “source” command.

The “run” (upper-right in editor window) the program will start at the position of the cursor in the script and run everything from that point forward. The “source” (located to the right of “run”) will run the entire script from the top to the end.

* + 1. Parts of the Script: Introduction

[Packages: tidyverse]

We want to start each script with a brief description and something about who created it and when it was created and updated. This is, again, a means of keeping track of our work and is just as much a reminder for us as it is for others that might utilize our programs. Since we do not want R to “run” our information, we need to comment it out of the script. We do this using the ‘hash’ character or ‘#’. Anything following this will be ignored by R until it encounters a carriage-return (ENTER KEY) so you can add comments “in line” with a command, but it must be located at the end of the command land or else nothing on that line will be run.

The top of my scripts looks something like this:

#This is a script to introduce the idea of scripts

#created by J. R. Groves on June 1, 2022

rm(list=ls())

library(tidyverse) #tells R to load the package tidyverse

The first command line ‘rm(list=ls())’ is a means of clearing the global environment to ensure that no objects remaining that might confuse the software. The text ls() lists everything in the environment and the rm() is the remove command. You can list objects individually or using a list as is done here.

Below that is the command that tells R to load our first package using the library() command. Notice that I have a comment at the end of the line telling the reader what this command does. It is not necessary with self-explanatory commands such as this, but in-line comments are a good way to explain things that may not seem self-evident.

If you run this script, you will get an error because we have never loaded the package *tidyverse* into our library. We can do that in one of two ways. The first is to go to the main menu in RStudio and choose the path ToolsàInstall Packages… which will open a popup window in which we can type ‘*tidyverse*’. This will reach out to one of the many CRAN sites and download the most recent *tidyverse* package and install it in your library.

A QUICK NOTE ON TIDYVERSE: The *tidyverse* package is a package of packages that make basic data manipulation, cleaning, organizing, and visualizing easier. In most cases, simply installing *tidyverse* is much easier than installing all the individual packages and takes up less room. Also, the authors have combined all the information in one web location: <https://www.tidyverse.org/>

* + 1. Parts of the Script: Load Data

The third part of the script (yeah, I know we skipped second, but hang tight) loads any data that you need for the script. Again, you can load the data at any point, but having it near the top and all together helps with reading the script and ensuring everything is where it is supposed to be. Data can be loaded in several ways ranging from creating the data, recalling data saved by a previous script, or importing data created outside of R.

Creating the data, as we will do here, is simply the act of assigning some elements or vectors to an object.

xd<-runif(10,0,1) #Creates a set of 10 random numbers using the uniform distribution

In this case we are assigning to the object ‘xd’ a set of 10 random numbers drawn from a uniform distribution across the values 0 and 1.

If we have an existing dataset created and saved in R, it will be saved as an *.RData* file and we can simply use the load() command. If, on the other hand, the data is created elsewhere and needs to be imported, we will need to use the read\_ command in *tidyverse* or find a package that has an import command for the data you need to import (such as SAS or STATA or .dbf). We will do examples of each throughout the class.

* + 1. Parts of the Script: Functions

Sometimes you may find yourself repeating the same set of commands repeatedly. Rather than retype the commands each time, you can write a function and then simply call that function. Think of a function as a script within a script as the function is only valid within that script because the script must load the function into R.

While it is not necessary to install functions at the front end of a script, it is good practice because, you cannot call a function unless it has been loaded into the environment. It also is helpful because if you need to make a change in the function, you know where to find it.

The syntax for a function is as follows:

sum.stat<-function(x){

v<-x / 10

a<-summary(x)

return(a)

}

Where the ‘sum.stat’ is the object to which the function is going to be assigned and the function(x) tells R that we are creating a function that will use the inputs ‘*x*’. This line is then ended with an open-curly bracket to indicate we are starting a series of commands rather than one-off commands. In other words, R will always run EVERYTHING within the curly brackets no matter where the cursor is.

When we are inside a function, any object created is consider “local” meaning that it only exists within the function. Once the function has completed and the output, if any, is reported, all objects created are deleted or forgotten by R. Notice how that we create an object ‘*v*’, but when we run our function, ‘*v*’ is nowhere in our environment. This is because it is a local object only. To ensure the output or result of the function is passed from the local environment to the global, we use the command return(). So, in this function we will return the summary statistics of our object *x* which we assigned to the object ‘*a*’ by using the command return(a). Finally, it is important to remember that the function is closed with ‘}’ which must be on its own line.

It is important to realize that returning the output of the function still does not necessarily create an object, it merely reports the output to the console. If we want to save the output as an object, we must assign the output to an object using syntax such as:

OB<-sum.stat(xd)

This will assign the output of the function sum.stat() to the object OB.

It is important to remember that if we change any element of the function, it must be “run” before those changes are recognized by R. If we wanted to pull the object ‘v’ instead of ‘a’, we would replace ‘a’ in our return(). If we then run the function again, notice we still return the value of a and not v. This is because we have updated the function in the script, but not in the memory. To do this, we must re-run the function to make sure the script matches what is in the memory.

* + 1. Parts of the Script: The Work

Finally, we want to do the actual work. When writing scripts, we want to keep them as clean and orderly as possible and it is a good idea to have a script do one thing such as clean and prepare a specific set of data for a project, and then a second script for running the regressions. While this may seem wasteful, remember that the marginal cost of a byte of memory is essentially zero and it will make the process easier to build and fix. You can get track of the order in which scripts should be run and what they do in your ReadMe.txt log file.

sum.stat(xd)

Here we are telling R to pass our set of random variables through our function sum.stat(). We can save this script, or we can see if it works by using the source command in the editor window. Using the source will produce the following in our console.

> sum.stat(xd)

Min. 1st Qu. Median Mean 3rd Qu. Max.

0.05728 0.49457 0.71262 0.62925 0.83977 0.91246

1. The First R Script

We next want to create our first script in R so you can navigate to `File --> New File --> R Script` and a new tab will open in your editor in RS. At the top we want to add a comment to explaining what we are doing with this file so we will add the following.

#First script to manipulate data

#Economics 691

#Your Name

#Today's Date

rm(list=ls())

library(tidyverse)

Located on Blackboard you will find a file called `ILCovidIL.csv` which is a data file that contains the daily cumulative counts of cases, deaths, and tests performed staring in March of 2020 through the most recent update. This file should be downloaded into your `./Project/ECON691JRG/Data` directory.

To get the file into R we use the read\_csv() command and assign the data to an objected named covidIL.

covid<- read.csv("https://raw.githubusercontent.com/nytimes/covid-19-data/master/us-counties.csv")

Next, we want to make sure we got what we think we are getting so we are going to use the `head()` command which instructs R to display the top five rows of the dataframe or vector. This is the one example of when we might enter a command directly in the console. If we put in it the script, it will run the command every time we run the script. Since we only want to do a one-time check, we can just run it in the console.

We have many observations from this raw data, but we only want observations from the State of Illinois, so we notice in the console that we have a variable ‘state’ and also a variable ‘fips’. The first is, obviously the state and is the most direct means to pull out the Illinois data, but the latter is a geographic code constant across all U.S. government data (at least in the last few decades). We can filter the data both ways using the tools in our *tidyverse* package.

covid.IL <- filter(covid, state==”Illinois”)

First notice that we use double equal signs to denote a logical argument of is equal to. Secondly, since we are using a character string, we put it in quotation marks.

We want to find the percentage change in cases for Cook County over the entire dataset and there are a variety of ways we can do that. We can do this is by creating a function and then using the piping function in *tidyverse* which allows us to string a series of commands together and keeps us from having to repeatedly type the dataframe in our script.

To create the percentage change we could find a time-series package if we wanted to create the percentage change from day-to-day or we could use something like the command lapply(). For practice, however, we are going to write a function called delta that will find the percentage change for a given vector. Further, we will use the standard new-old divided by old percentage change formula. Since we have installed *tidyverse* we can find the lagged value of a vector in one of two ways. One is to use the command lag() or we can use the syntax v[-1] which takes the lagged value of vector v.

delta<-function(x){

temp<-((x-lag(x))/lag(x))

return(round(temp,4))

}

So, in this command we create the local object “temp” that will calculate the percentage change in values of x and then return that vector with all observations rounded to four decimal places.

Another feature of the *tidyverse* package is that we can manipulate and clean data without having to repeat a bunch of commands. We want to find the percentage change in tests, cases, and deaths and add those vectors to our data frame. To do this we will use the ‘piped’ [%>%] syntax which is a means of nesting commands that provides for simple reading and does not require constantly inputting the same dataframe name repeatedly.

Finally, because we want our work to be as transparent as possible, we are going to work with our original raw data. Thus, our commands are going to include the filter commands and the mutate() command, which is how *tidyverse* creates new variables. Since we are focusing on a single jurisdiction in Illinois, we can use the fips code to filter our data

covid.IL<-covid %>%

filter(fips == 17031) %>%

mutate(pc\_cases = delta(cases),

pc\_deaths = delta(deaths))

We are assigning to the object covid.IL the dataframe covid with the following changes:

* filtering the data for only Cook County, Illionis,
* create a new vector named *pc\_cases* which does the same for the cases vector,
* create a new vector called *pc\_deaths* using the vector deaths.

Notice how within the mutate() command we use the equals sign rather than the assign parameter. This is simply an artifact of the syntax of the mutate()command which we can learn about by going to our lower-right window and click on the tab “HELP”. On the right side we see a textbox with a magnifying glass (search box) and if we type in the command mutate and hit enter, we will get the documentation for that command explaining all the options and, at the bottom, examples of its use.

Notice that if we were to use the mutate() command outside of a pipped series of commands, we would need to input the source of the data (.data in the help file) each time. By using the pipped method, we are essentially pipping the dataframe covid through all the following commands.

As with any data, however, there may be evil lurking deep in the data so another check is to use summary statistics (therefore every research paper should include a table of summary statistics). To check the summary statistics in R, we use the command summary() to produces the summary statistics for any integer vectors within our dataframe. Again, if we are doing this as a check, then we can just enter it in the console.

We see a possible error in the data for the percentage change in the number of deaths because we see 7 NA terms (missing data in R) and the mean is Inf which stands for infinite. The Inf term is occurring because, when we visually inspect the data, we see several cases where the denominator of our percentage change is zero and if we try to divide by zero, the university explodes, thus the error. To fix this, and to get R to ignore those cases, we want to switch them from Inf to NA. We can fix this in the code a few ways, but we will use the following which utilizes the is.infinite() command in the base R.

Additionally, we want to make sure that the date variable is being read as a date rather than just a character vector and so we can use the command as.Date() to tell R this is a date and so the sequence means something. In the interest of transparency, we will add these changes to our pipped commands used above.

covid.IL2<-covid.IL %>%

mutate(pc\_deaths = ifelse(is.finite(pc\_deaths), pc\_deaths, NA),

Dates = as.Date(date, "%Y-%m-%d"))

Notice that to change the infinites we use an ifelse() command. This allows us to look at each element of the vector in question one at a time. Also notice that we gave this object a new name. We could have simply changed the covid.IL object; however, I like to use new object names until I know I am doing what I think I am doing with code. Now that I know it works (I can run summary() again), then I can either add the two new elements to my original mutate() command or replace *covid.IL2* with *covid.IL*.

Changing the inf to an NA we now can see that the average percentage change in total deaths is about 1.4% over the span of the data.

Another item to notice is how we used the as.Date(). We first told it what vector to recognize as the date and then we told it the format that the vector is written in. The list of options are as follows:

* %Y indicates four digit year
* %y indicates two-digit year
* %m indidates two-difit month
* %b indicates abbreviated character month (Dec)
* %B indicates full name month (December)
* %d indicates two-digit day.

It will appear that the only thing that change (when you use the head()) command is that the date is reorganized; however, if you look at the type of column where *tidyverse* puts the type of vector, we see it is Date rather than char. This means that R knows there is a sequence here that has meaning.

Finally we want visualize the percentage change in deaths by creating a scatterplot using the plot() command. When we run this part of the script, we get a scatterplot in the Plots tab of our lower-right window (where Help and File tabs are) that looks something like this.

A picture containing shape

Description automatically generated

plot(covid.IL2$Dates,covid.IL2$pc\_deaths)

Of course, the problem with this is it is very bland and hard to read. Part of this is due to what we are looking at. Specifically, we are looking at the percentage change from day-to-day in the TOTAL DEATHS, not the new deaths reported. As a result, we expect to see very small numbers as the cumulative total rises. The same is true for the total cases and total tests.

We can also dress up the graph a bit by using the options built into the plot() command to define a total, use a line instead of points, change the axis titles and the color.

plot(covid.IL2$Dates,covid.IL2$pc\_deaths, main="Percent Change in Deaths", ylab="", xlab="Date", type="l", col="blue")

This; however, is still a bit underwhelming so next week, among other things, we will introduce the *ggplot* package, which is part of *tidyverse*, to produce some nice-looking graphs.

Graphical user interface, text, application

Description automatically generated

**Homework Assignment 1:**

* Share your GitHub Repository with me.

Log into your GitHub account and go to the Repository’s main page. Click on the settings tab and then on the Manage Access. Here, add a collaborator and search by username. My username is ‘jrgroves’ and my image the NIU logo on a white background.

* Create a function called DIF that that calculates the number of daily new cases and new deaths and then use this function and the mutate command to add these as new columns in the original dataframe.
* Also using the mutate command, find the percentage change in these values and then use the filter command to limit the data to everything after January 2021.
* Create the following plots [Daily Percentage Change in New Cases, Daily Percentage Change in Deaths] and then using the Save as Image command in the Export dropdown menu to save them and then insert them into a Word document (using insert picture). If you all three pictures should fit on one page in Word. Print this file as PDF and then upload it to your GitHub Repository (drag and drop should work, or you can google how to do this).
* Commit and push the final script you used (named HW1.R) and the PDF for this assignment to your GitHub Repository. I will grade the script and the posted image.