Introduction to Data Science in R

Joseph M. Westenberg

2021-07-15

Contents

Chapter 1

Preface

These notes are a first draft and quite prelimenary. If you find errors, please let me know by emailing jwesten@iu.edu

An example of my code for this course is avaialabe on my github repository at https://github.com/r-introtodatascience/sample_repo.

The goal of these short courses are to give graduate students a very prelimenary introduction into coding in R. While having an 'instructor' teach you these things may be helpful, in my belief, coding, like math, you have to learn by doing. So I highly encourage following these notes, but then play around with the code. Change things and see how it changes the output. I think this is the best way to learn.

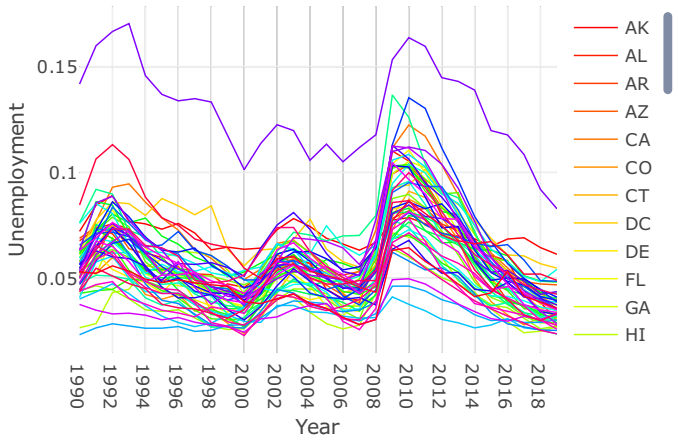
Over the next 5 'sessions' we will be looking into county level unemployment data. The goal will be to provide examples and descriptions of the commands and techniques I have discovered to be most useful while working with data. Specifically the goal is by the end of the sessions it is my hope that you will know how to:

- download data straight into R
- read in data (Rda, csv, xlsx, dta)
- merge datasets
- subset datasets
- create summary statistics and plots
- run basic regressions

all while paying particular attention to file pathing and to teach you how to export graphics and tables from R into a form that can be directly read into LaTex (so that it can be easily read into paper/presentation documents).

Below is an example of an interactive plot you can make with a package called plotly. Try interacting with it! In the plot you can select/deselect states you

want to view. There is also a link to a short post I made about using GIS with R. I show these to illustrate a bit of what is possible to do in R after we familiarize ourselves with the basic functions.



Chapter 2

Introduction

The goal of this section is to provide a very quick introduction or refresher to R. If this is your first time coding, this brief introduction will most likely be quite insufficient. I encourage you to seek other resources to learn the basics before proceeding.

In particular we will cover:

- Writing Loops
- String Manipulation
- Project Organization
- Downloading/Reading Data
- Downloading Loop

In this lesson we will be downloading and cleaning data from the BLS on county level employment statistics from 1990-2019.

2.1 Very Basics

I assume you have R installed and running. There are plenty of guides online on how to do this.

Let's first define some arrays within R.

These can be numeric based, in this case integer.

```
numvec1<-c(5,6,7)  # 1
numvec2<-c(7,8,9)  # 2
```

Let's dig into what is going on a bit more. We are telling R to define a vector, this is the c( ) part, with elements 1,2,3 and give that vector a name A. The backwards arrow tells R what is the name and what is the element we are defining.

We can make character based vectors as well.

```
charvec1<-c("a", "b", "c")  # 1
charvec2<-c("d", "e", "f")  # 2
```

We can then combine these vectors into a dataframe (this is relevant for when we start thinking about reading in/manipulating actual data). Since all of our vectors are length three, we can easily create a dataframe (think a matrix) where our column names will be the name of the vectors, and the rows will be the elements of the vectors.

```
first_dataframe<-data.frame(numvec1,numvec2,charvec1,charvec2)  # 1
print(first_dataframe)                                          # 2
```

```
##   numvec1 numvec2 charvec1 charvec2
## 1       5       7        a        d
## 2       6       8        b        e
## 3       7       9        c        f
```

Now first_dataframe is going to be of similar format as we will typically have when we read in data from excel files into R. We can access certain rows and columns within the dataframe by putting square brackets after the name of the dataframe. For example if we wanted to print the element in the first row and first column, we could define the variable x as this and then print x. (Keep in mind the ordering is rows, columns)

```
x<-first_dataframe[1,1]  # 1
print(x)                 # 2
```

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

What if we wanted to print **all** elements in the first row, we just leave the column (after the comma) blank:

```
x<-first_dataframe[1,]  # 1
print(x)                # 2
```

```
##   numvec1 numvec2 charvec1 charvec2
## 1       5       7        a        d
```

How about 1st & 3rd row?

```
x<-first_dataframe[c(1,3),]  # 1
print(x)                     # 2
```

```
##   numvec1 numvec2 charvec1 charvec2
## 1       5       7        a        d
## 3       7       9        c        f
```

Another way to do this is to define another variable, say y as a vector with elements 1 and 3. Notice how the below output is the same as the above.

```
y<-c(1,3)               # 1
x<-first_dataframe[y,]  # 2
print(x)                # 3
```

```
##   numvec1 numvec2 charvec1 charvec2
## 1       5       7        a        d
## 3       7       9        c        f
```

We can do the same thing for the columns (we need to remember the order for the square brackets are rows, columns). Note: if we put a negative sign in front of these commands in the brackets, instead "keeping" certain rows or columns, it means remove! That is if in the below command we have -3, it would be saying REMOVE column 3!

```
x<-first_dataframe[,3]   # 1
y<-first_dataframe[,-3]  # 2
print(x)                 # 3
```

```
## [1] "a" "b" "c"
```

```
print(y)  # 1
```

```
##   numvec1 numvec2 charvec2
## 1       5       7        d
## 2       6       8        e
## 3       7       9        f
```

Our third column is named C, we can also pull this column by referencing it's name after a dollar sign, such as:

```
x<-first_dataframe$charvec1  # 1
print(x)                     # 2
```

```
## [1] "a" "b" "c"
```

This may not sound useful now, but think if we have many columns of variables, say wage, hoursworked, fulltime, and hundreds of more. We don't want to have to find what column number hoursworked is, we can just reference this column name.

2.1.0.1 "And" and "Or" Operators

A couple common operators that we may want to use are "and" and "or" statements. Within R:

- "&" : Our "and" operator
- "|" : Our "or" operator

For example:

```
first_dataframe[first_dataframe$numvec1>=6 & first_dataframe$numvec2<9,]  # 1
```

```
##   numvec1 numvec2 charvec1 charvec2
## 2       6       8        b        e
```

```
first_dataframe[first_dataframe$numvec1>=6 | first_dataframe$numvec2<9,]  # 1
```

```
##   numvec1 numvec2 charvec1 charvec2
## 1       5       7        a        d
## 2       6       8        b        e
## 3       7       9        c        f
```

2.2 Writing Loops

```
1 letters<-c("a", "b", "c", "d", "e")  # 1
2 letters_l<-length(letters)           # 2
3 for (i in 1:letters_l){              # 3
4   print(letters[i])                  # 4
5 }                                    # 5
```

```
## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
## [1] "e"
```

Going line by line:

- Line 1: Define a vector of letters.
- Line 2: Report the number of elements in our letter vector and save as letters_l
- Line 3: Defining for loop. Our index will be i, and it will run from 0 to however long letters vector is (try adding some more letters!) Note our for loop action is defined within the curly braces.
- Line 4: For every i defined in Line 3 we want to print the corresponding element in the vector letters.

Okay, great. How does this help us with reading in the data? We'll get to that in the next section.

2.3 String Manipulation

2.3.1 String Concatenation

```
a<-"This is a the start"  # 1
b<-"of a sentence"        # 2
```

```
print(paste0(a,b))        # 3
```

```
## [1] "This is a the startof a sentence"
```

```
print(paste(a,b))  # 1
```

```
## [1] "This is a the start of a sentence"
```

Notice we define a and b as strings. What paste and paste0 do are combine these strings into one string. We can see that paste places a space between the two strings while paste0 does not. paste0 comes in quite handy for working with file pathing as we will see. Yes it's that easy!

We will not be using this for this project but it may be useful to know we can concatenate two vectors of strings as well.

```
a<-c("This is a the start", "Now we have", "This really is", "Economics is")  # 1
b<-c("of a sentence.", "another sentence.", "quite handy.", "awesome!")       # 2
print(paste(a,b))                                                             # 3
```

```
## [1] "This is a the start of a sentence." "Now we have another sentence."
## [3] "This really is quite handy."        "Economics is awesome!"
```

2.3.2 String Padding

Consider we have a vector of numbers which currently runs 1-19. Now what if we need all the 'single character' digits to have a leading zero. That is instead of "1" we need "01".

We could use paste0 as above and combine a 0 with our vector.

```
numbvec<-as.character(1:19)  # 1
print(paste0("0", numbvec))  # 2
```

```
##  [1] "01"  "02"  "03"  "04"  "05"  "06"  "07"  "08"  "09"  "010" "011" "012"
## [13] "013" "014" "015" "016" "017" "018" "019"
```

But we don't want a leading 0 in front of the double 'character' digits (ie We DON'T want "090"). We could break our vector into single character digits and two character digits, manipulate the single character digits, then combine it back in with the double character digits. But there is an easier way: str_pad !

```
require(stringr)                          # 1
numbvec<-as.character(1:19)               # 2
print(str_pad(numbvec, 2, "left", "0")  )  # 3
```

```
##  [1] "01" "02" "03" "04" "05" "06" "07" "08" "09" "10" "11" "12" "13" "14" "15"
## [16] "16" "17" "18" "19"
```

Now this looks like what we want! But what is str_pad doing? With str_pad we are telling stringr we want all elements of numbvec to be of length 2. So stringr checks to see if the elements are less than 2 characters, if an element is it adds "0"'s to the left side until it reaches length 2. If it is already length 2, it will leave it alone.

There is many other handy commands to deal with strings in R. These are just a couple of commands we will be using. I will be writing a post with some other handy functions in the coming weeks and will link it here.

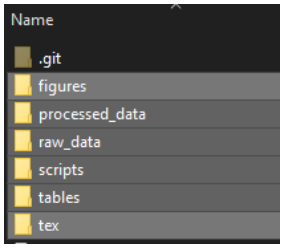
2.4 Project Organization

Before we get started, let's set up a folder for our project and create subfolders to keep things organized. For this project I recommend the following subfolders, which are a good minimum for organizing any project:

- raw_data: This is where we will put our 'preprocessed' data we will be getting from MIT Election Lab.
- scripts: Where we will save all of our R-scripts in this folder
- processed_data: If we want to save some intermediate data steps between raw data and our output.
- tables: Any tex tables we generate we will save to this folder
- figures: Any figures we generate we will save to this folder
- tex: Where we can have our paper and/or presentations

You can make more folders if you feel it keeps you organized. The main point I want to make here is it is well worth your time to think about how you want to organize your project. Oftentimes if we jump right in without a plan, things become a jumbled mess. (If you want to go further with your organization strategies, I recommend looking into waf, specifically check out Templates for Reproducible Research Projects in Economics.)

My project folder now looks like this:



2.4.1 First R script!

Now that we have our folders created, let's now create paths to those folders with R. Let's start by creating a variable workingdir which we can define as the path to our main folder.

```
workingdir<-"PATH_TO_YOUR_WORKING_DIR"  # 1
```

Now we are going to create a script, let's save it as workingdir.R and place it in our main parent folder. We are then going to be using the path we defined above in this script to create paths to our other folders.

```
folder_figures<-paste0(workingdir, "figures")                # 1
folder_processed_data<-paste0(workingdir, "processed_data")  # 2
folder_raw_data<-paste0(workingdir, "raw_data")              # 3
folder_scripts<-paste0(workingdir, "scripts")                # 4
folder_tables<-paste0(workingdir, "tables")                  # 5
folder_tex<-paste0(workingdir, "tex")                        # 6
```

Now at the top of all other scripts we can have:

```
workingdir<-"PATH_TO_YOUR_WORKING_DIR"    # 1
source(paste0(workingdir, workingdir.r))  # 2
```

Now all this says is we have a variable named workingdir that points to our main parent folder. We then have a script in which we define paths to all other folders. With this at the front of all of our scripts, we can easily reference them to create easier pathing for ourselves.

2.5 Downloading within R

We can use R to with a direct link to download. The first argument the download.file() command takes that we will use is the url of the xslx document and the second argument is the destination it will be saved. The last argument is basically telling R that the excel docs are not plain text. (Don't forget to have workingdir defined as we did in Project Organization

```
download.file("https://www.bls.gov/lau/laucnty90.xlsx",      # 1
  paste0(folder_raw_data, "bls_unemp_90.xlsx"), mode="wb")   # 2
```

2.6 Reading in Data

Before we read in this data to R, let's see what we are dealing with. Opening up the file in excel we can see there will be issues reading the file in.

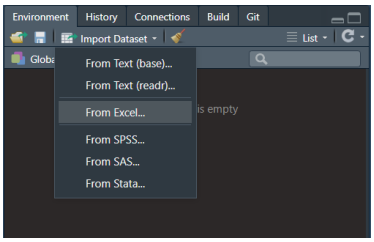
	A	B	C	D	E	F	G	H	I	J
1	Labor Force Data by County, 1990 Annual Averages									
2										
3		State	County							Unemploy-
4	LAUS	FIPS	FIPS				Labor			ment Rate
5	Code	Code	Code	County Name/State Abbreviation	Year		Force	Employed	Unemployed	(%)
6										
7	CN0100100000000	01	001	Autauga County, AL	1990		16,908	15,813	1,095	6.5
8	CN0100300000000	01	003	Baldwin County, AL	1990		46,824	44,379	2,445	5.2
9	CN0100500000000	01	005	Barbour County, AL	1990		11,490	10,592	898	7.8

We can notice that the first row has the file title spread across columns A:J. Variable names are spread across anywhere 1-3 rows. And lastly we have an

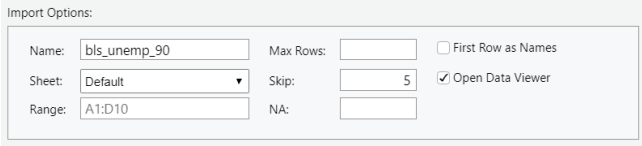
empty column in F. The bright side is if we observe the other year's files, they all have this exact same structure. Hence we will able to use a loop eventually to clean them all instead of one at a time.

The main package we will be using is readxl, which is quite self explanatory. It is a package meant to help to read in excel docs. Let's try to open the file for 1990 we downloaded in R. We can do this through R Studio's functionality.

Within the "Environment" area of R Studio, click Import Dataset, then From Excel...



There is definitely multiple ways to do this, as we can see from the options available. I first deselect "First Row as Names" (This option is very nice if your data is already in a precleaned form and your first row simply has your variable names.) I then begin to skip rows, 5 rows of skipping leads to the first row being the first row of data.



Next we can handle column F that we noticed was blank. This is column 6 and stays consistent across all years (you can check this). Remembering our lessons from the Very Basics section we can subset this dataframe by removing column 6.

Hence we have something that will look like the following for our command for our script.

```
require(readxl)
workingdir<-"C:/Users/weste/Documents/GitHub/r-introtodatascience/sample_repo/"
bls_unemp_90 <- read_excel(paste0(folder_raw_data, "/bls_unemp_90.xlsx"),
                           col_names=FALSE, skip=5)
bls_unemp_90<-bls_unemp_90[,-6]
colnames(bls_unemp_90)<-c("LAUS_code", "State_fips", "County_fips", "County_name",
                          "Year", "Labor_force", "Employed", "Unemployed", "Unemp_rate
```

Where the last line above we are giving our columns names based on the names we saw in the excel document.

We can then observe our data frame to ssee if it's fully cleaned. At first it seems so (I actually initially thought so). However when reading in the data, we grabbed 2 extra rows at the end of the file. Hence we have 2 rows at the end of our data frame that are NA's. Let's drop these two rows, to do this we can use a command is.na and prior to it include an "!", saying 'not is.na'.

```
bls_unemp_90<-bls_unemp_90[!is.na(bls_unemp_90$State_fips),]  # 1
```

Here is a good place to pause if you want a challenge. You should have all the tools needed to write a loop to download all files from 1990-2019.

2.7 Download Loop

In the next sections we will be using county level employment/labor force data from BLS to learn more about working with actual data. We will be using the **Labor force data by county, yearly annual averages**. There is data from 1990-2019 (as of writing these notes). To start we are going to download this data and then read it into R.

We can use a loop to download/clean our data such as this:

```
require(stringr)                                                          # 1
require(readxl)                                                           # 2
unemp_data<-data.frame()                                                  # 3
years<-c(90:99, 0:19)                                                     # 4
years<-str_pad(as.character(years), 2, "left", "0")                       # 5
years_l<-length(years)                                                    # 6
for (i in 1:years_l){                                                     # 7
url<-paste0("https://www.bls.gov/lau/laucnty", years[i], ".xlsx")         # 8
destination<-paste0(folder_raw_data, "/bls_unemp_", years[i], ".xlsx")    # 9
download.file(url, destination, mode="wb")                               # 10
temp_df <- read_excel(paste0(folder_raw_data, "/bls_unemp_", years[i],    # 11
  ".xlsx"),col_names=FALSE, skip=5)                                       # 12
temp_df<-temp_df[,-6]                                                     # 13
colnames(temp_df)<-c("LAUS_code", "State_fips", "County_fips", "County_name",  # 14
  "Year", "Labor_force", "Employed", "Unemployed", "Unemp_rate")          # 15
temp_df<-temp_df[!is.na(temp_df$State_fips),]                             # 16
unemp_data<-rbind(county_data, temp_df)                                   # 17
}                                                                         # 18
filename<-paste0(folder_processed_data, "/unemp_data.rda")                # 19
save(unemp_data, file=filename)                                           # 20
```

To work through this first let's think what we are trying to achieve. The links for the downloads are all in the form of https://www.bls.gov/lau/laucntyZZ.xlsx, where ZZ is two digits representing the year. These ZZ values run from "90"to "99" for years 1990-1999 and "00" to "19" for years 2000-2019.

Let's work through the above code line by line:

- Lines #1 & #2: load required packages.
- Line #3: Declare unemp_data will be a data.frame. Right now it is empty, but we will add to it.
- Line #4: Define a vector with elements 90-99 and 0-19. (Which will correspond to the years that we will pull)
- Line #5 : We have a vector for years, however if we notice in the url names we need this vector to include a leading 0 in front of the single 'character' digits (ie "01" instead of "1"). But we don't want a leading 0 in front of the double 'character' digits (ie We DON'T want "090"). Go back to the String Manipulation section if you need to refresh on this.
- Line #6 : Calculate the length of years and save as years_l
- Line #7 : See String Manipulation if defining the for loop does not make sense.
- Line #8 : We are creating the character string for the url for the download link. Since they all take the form of https://www.bls.gov/lau/laucntyZ Z.xlsx, we can use one element of our years vector at a time. (See String Manipulation for explanation on paste0)
- Line #9 : This is of similar spirit to line #4, but this is defining the path/filename of the excel file we will save.
- Line #10 : This line is just telling R to download the file at that url, save it to the defined location/name, and to read it as a non-raw text form. (see Downloading/Reading Data if unclear.)
- Lines #11-#16: See Downloading/Reading Data for a direct explanation.
- Line #17: rbind appends data. Hence since all of our data has the same format and has a variable indicating the year, we can simply append.
- Line #19: Create the filepath (to our processed_data folder) where we will save the file, what the file name and type is.
- Line #20: save the combined data to the location/name we defined above.

We now have our data cleaned and saved for our next lesson when we will start to work with it more!

Chapter 3

Tidyverse

The goal of this Tidyverse section is to learn

- Subsetting Datasets
- Merging Datasets
- Summarizing
- Plotting

In this section we will be continuing to be working with the BLS data we downloaded in the Introduction.

Tidyverse provides us with some useful tools for data manipulation and cleaning. I include examples and descriptions of the commands I have most frequently used.

3.1 Filtering with Pipes

```
indiana<-                                    # 1  # 1
    unemp_data %>% filter(State_fips=="18")  # 2  # 2
```

What is going on in this line of code? Let's start with the second line. The operator %>% is called a pipe. This comes from a package called magittr which is included within tidyverse. Pipes are very handy for neatly writing longer sequences of code. Here we begin with a pretty simple example. We can think of pipes as saying "then". That is it takes the argument before it and uses it as input in the following command. Here we are using a filter command, so we are saying take our data.frame county_data, then filter or "give us" the values where State_fips is equal to "18" (this is fips code for Indiana. Go Hoosiers!). So in summary we are just filtering out indiana data and saving it as a new data frame (this is line 1). Note: this operation does NOT impact county_data, the

only way we 'over write' a data frame is to explicitly tell it to. For example the following WOULD overwrite county_data.

```
unemp_data<-unemp_data %>% filter(State_fips=="18")  # 1  # 1
```

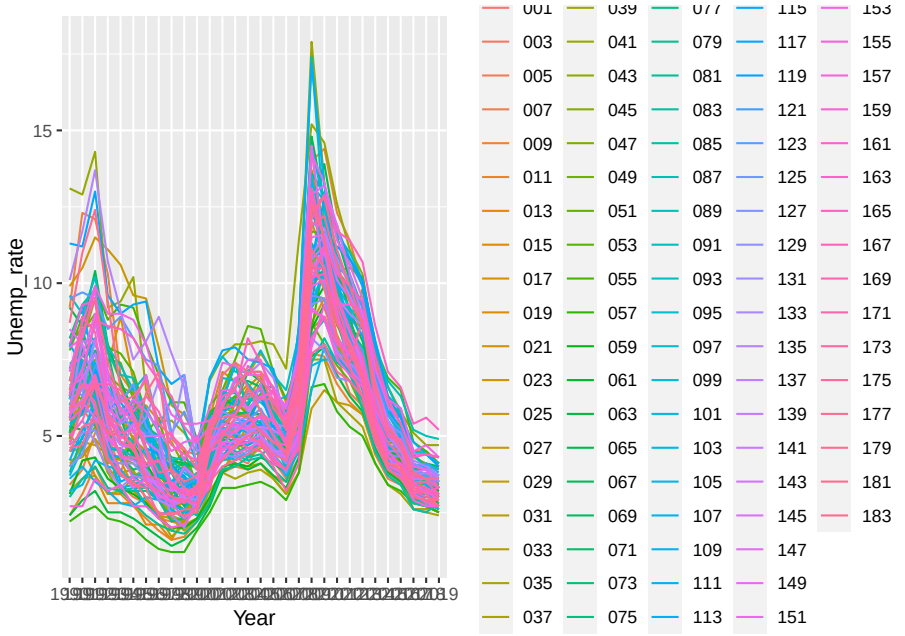
3.2 Plotting Data

In this lesson we will be plotting the data gathered in previous lessons. ggplot2 is a very popular package for plotting and is now included as part of the tidyverse package. It allows us to make many different kinds of plots and customize them. As it is so popular there are many guides for it. Tidyverse's website summarizes some very useful resources.

As ggplot2 has so many options, we will just be barely skimming the surface in this introduction. Let's start with a minimal command. First let's load in ggplot2 filter our data down into just Indiana.

```
library(ggplot2)                                          # 1  # 1
                                                          # 2  # 2
indiana <- unemp_data %>% filter(State_fips=="18")        # 3  # 3
```

```
ggplot(data=indiana, aes(x=Year, y=Unemp_rate,    # 1  # 1
    colour=County_fips, group=County_fips))+      # 2  # 2
  geom_line()                                     # 3  # 3
```



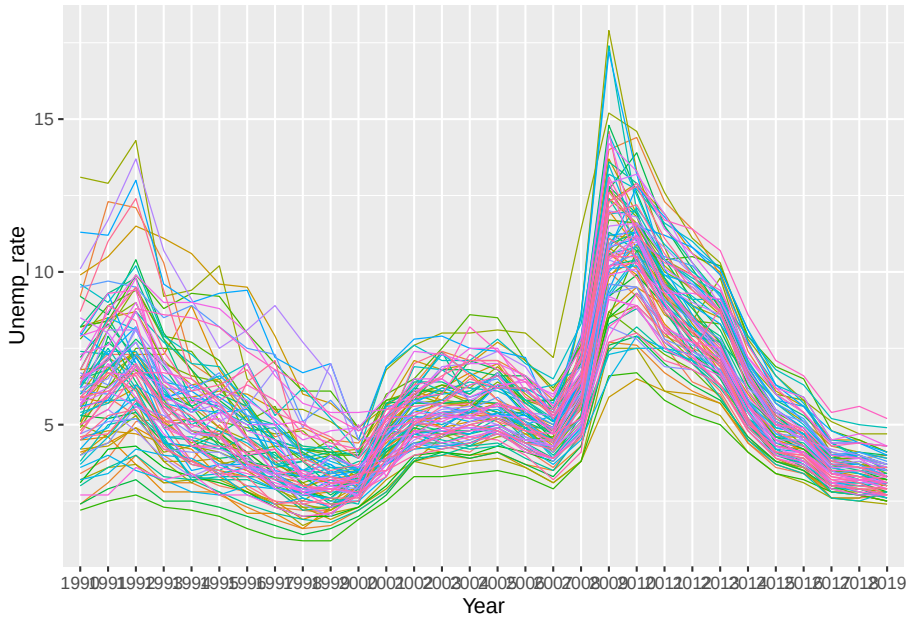
- Line #1: Tells ggplot we will be using the data.frame indiana. aes is short

for aesthetics. This is where we tell ggplot how to map our data into the plots. Here I want ggplot to plot Year on the x-axis, Unemp_rate on the y-axis.

- Line #2: A continuation of the aesthetic mapping, his line is saying plot separately for each County_fips separately, giving them a different color.
- Line #3: Tells ggplot to plot the data as lines.

First thing after this plot, the legend is excessive. Let's just suppress it for now. To do this we can define the legend.position in theme as none. The theme command allows for a lot of customization with the plot. If you want to see all options click here. Let's also change the thickness of the lines, we can do this with the alpha argument.

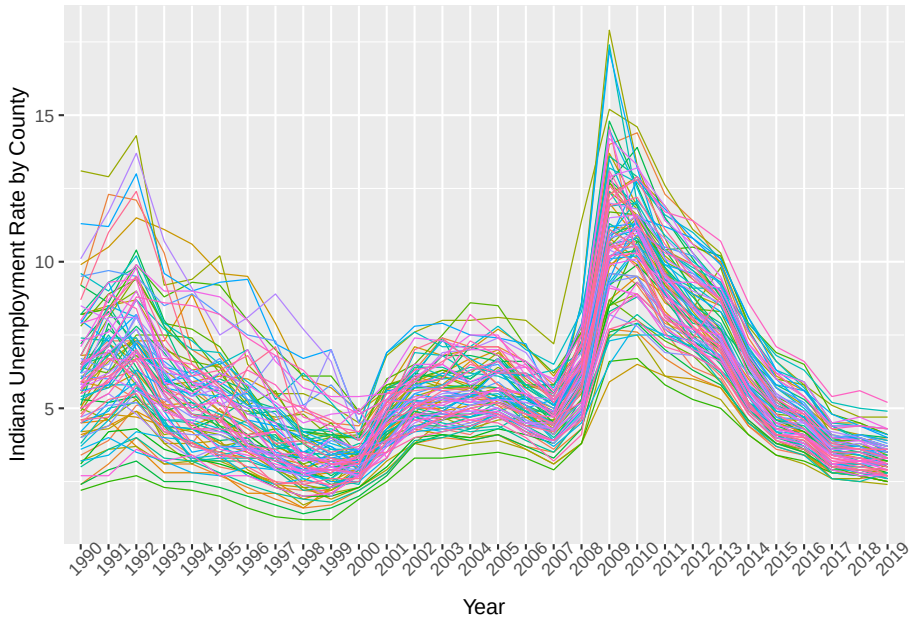
```
ggplot(data=indiana, aes(x=Year, y=Unemp_rate,    # 1  # 1
    colour=County_fips, group=County_fips))+      # 2  # 2
  geom_line(size=0.3)+                            # 3  # 3
  theme(legend.position="none")                   # 4  # 4
```



This already is looking a lot better. But can we do something about the labels on the x axis, they are overlaying on each other. We could make the text smaller but then it would be difficult to see, so let's just rotate them. Also let's make the y-axis name a bit more meaningful.

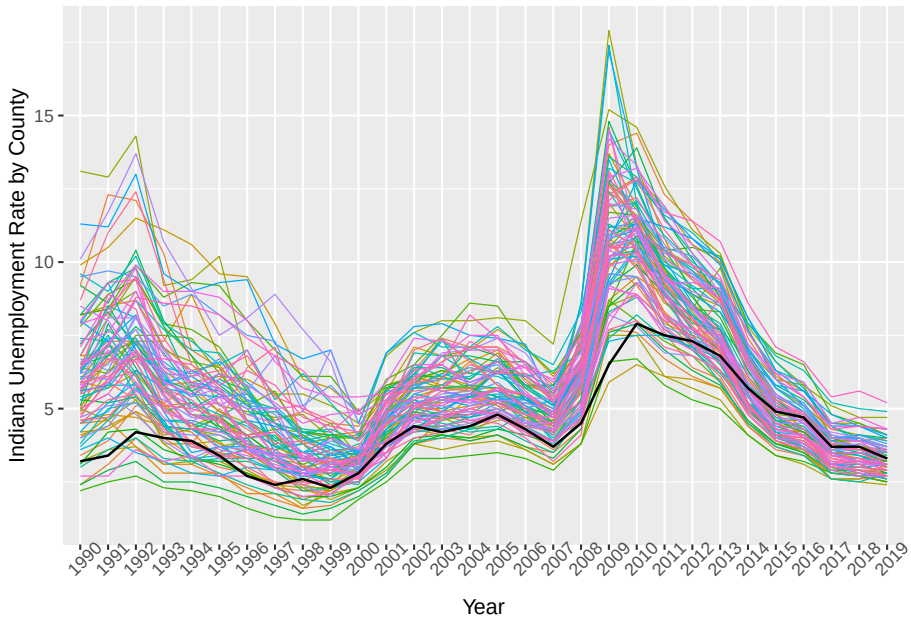
```
ggplot(data=indiana, aes(x=Year, y=Unemp_rate,    # 1  # 1
    colour=County_fips, group=County_fips))+      # 2  # 2
  geom_line(size=0.3)+                            # 3  # 3
  theme(legend.position="none",                   # 4  # 4
```

```
    axis.text.x=element_text(angle=45)) +        # 5  # 5
  ylab('Indiana Unemployment Rate by County')    # 6  # 6
```



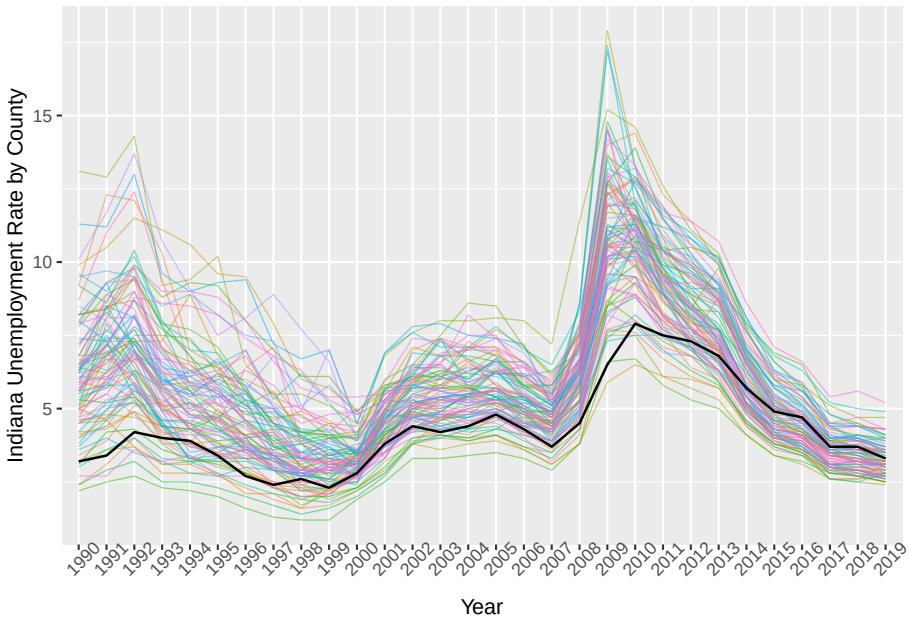
Now we're getting somewhere! But what if we want to emphasize the county Bloomington is in, Monroe County?

```
monroecty<-indiana %>% filter(County_fips=="105")      # 1   # 1
                                                       # 2   # 2
ggplot(data=indiana, aes(x=Year, y=Unemp_rate,         # 3   # 3
    colour=County_fips, group=County_fips))+           # 4   # 4
  geom_line(size=0.3)+                                 # 5   # 5
  geom_line(data=monroecty, aes(x=Year, y=Unemp_rate,  # 6   # 6
    colour=County_fips, group=1),                      # 7   # 7
    size=0.6, colour='black') +                        # 8   # 8
  theme(legend.position="none",                        # 9   # 9
    axis.text.x=element_text(angle=45)) +              # 10  # 10
  ylab('Indiana Unemployment Rate by County')          # 11  # 11
```



We can make the black line for Monroe County 'pop' a bit more too. Let's change the opacity of the other counties lines. To do this we can set a value, alpha. This takes values 0 to 1 where smaller values are more transparent.

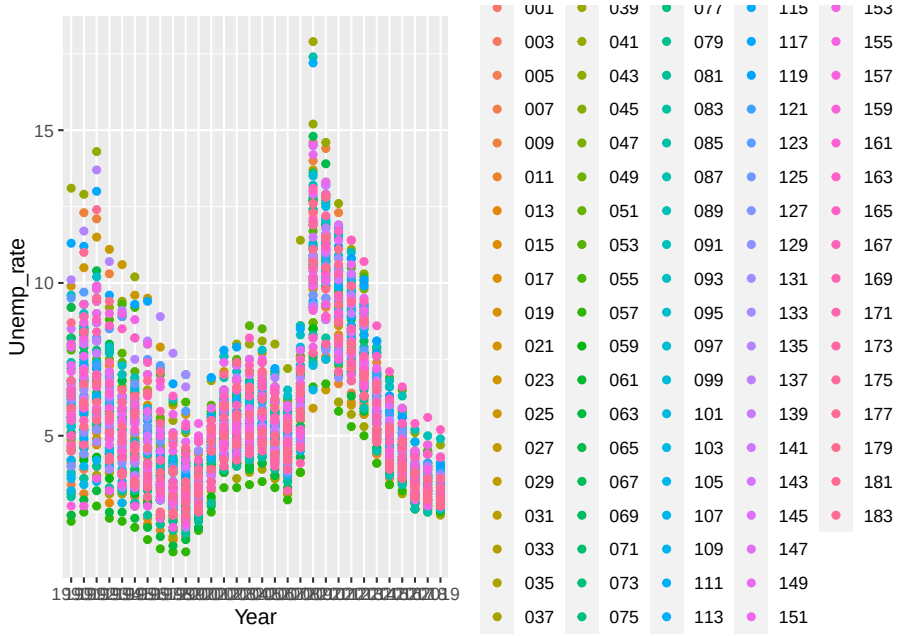
```
ggplot(data=indiana, aes(x=Year, y=Unemp_rate,          # 1  # 1
    colour=County_fips, group=County_fips))+            # 2  # 2
  geom_line(size=0.3, alpha=0.5)+                       # 3  # 3
  geom_line(data=monroecty, aes(x=Year, y=Unemp_rate,   # 4  # 4
    colour=County_fips, group=1),                       # 5  # 5
    size=0.6, colour='black') +                         # 6  # 6
  theme(legend.position="none",                         # 7  # 7
    axis.text.x=element_text(angle=45)) +               # 8  # 8
  ylab('Indiana Unemployment Rate by County')           # 9  # 9
```



3.2.1 Scatter plot (geom_point)

What if we want points to insteaed of lines for each year?

```
ggplot(data=indiana, aes(x=Year, y=Unemp_rate,    # 1  # 1
  colour=County_fips, group=County_fips))+        # 2  # 2
  geom_point()                                    # 3  # 3
```



3.3 Manipulating within Groups

Our next chunk of code gets a bit more complex, let's take it peice by peice though.

```
indiana %>% group_by(Year) %>%          # 1  # 1
  summarise(Average=mean(Labor_force))  # 2  # 2
```

Let's again take it from the end and work our ways backwards through it. The 3rd line above is calculating a new variable within a group that is called 'Average'. The group (line 2) that we are calculating within is each Year for our Indiana data.frame. That is this snippet of code will give us one value back for each year, the average Labor Force level across Indiana's counties.

Year	Average
1990	30593.65
1991	30357.00
1992	31058.60
1993	32046.84
1994	33308.01
1995	34078.00
1996	33800.80
1997	33919.72
1998	33947.47
1999	33965.91
2000	33982.50
2001	34140.32
2002	34469.34
2003	34597.76
2004	34432.72
2005	34841.75
2006	35164.35
2007	34866.21
2008	35131.49
2009	34717.45
2010	34512.97
2011	34586.91
2012	34454.88
2013	34656.59
2014	35052.00
2015	35508.30
2016	36178.11
2017	36243.96
2018	36756.59
2019	36819.25

This group_by command paired with summarise is quite useful. For example going back to our original data.frame, county_data. Suppose we wanted to do what we just did but for all states. That is calculate within each state and year combination. With the group_by command this is quite simple! (I print the first 100 lines of output below)

```
unemp_data %>%                            # 1  # 1
  group_by(Year, State_fips) %>%          # 2  # 2
  summarise(Average=mean(Labor_force))    # 3  # 3
```

```
## `summarise()` has grouped output by 'Year'. You can override using the `.groups` arg
```

Year	State_fips	Average
1990	01	28468.343
1990	02	10323.923
1990	04	119079.067
1990	05	15091.147
1990	06	261007.414
1990	08	27557.641
1990	09	227471.000
1990	10	120074.667
1990	11	328924.000
1990	12	96480.269
1990	13	20653.579
1990	15	137730.500
1990	16	11193.045
1990	17	58304.049
1990	18	30593.652
1990	19	14664.212
1990	20	12119.305
1990	21	14619.667
1990	22	29216.547
1990	23	39563.625
1990	24	107919.000
1990	25	229027.357
1990	26	55509.072
1990	27	27573.690
1990	28	14438.854
1990	29	22737.852
1990	30	7198.929
1990	31	8871.215
1990	32	39407.176
1990	33	62009.100
1990	34	193117.000
1990	35	21630.061
1990	36	142149.565
1990	37	34765.130
1990	38	5985.226
1990	39	61586.670
1990	40	19786.338
1990	41	41408.833
1990	42	86992.045
1990	44	105072.600
1990	45	37903.326
1990	46	5260.303
1990	47	25207.768
1990	48	33932.858
1990	49	28352.138
1990	50	21768.214
1990	51	24452.541
1990	53	64751.974
1990	54	13857.600
1990	55	35970.542
1990	56	10259.348
1990	72	14517.090
1991	01	28665.134

Average is just one statistic we may want to calculate, how about percentiles, minimum, maximum, standard deviation. our summarise command will accept more than one argument, we just have to separate by a comma. Let's also save this as a new data.frame for use in future lessons.

```
indiana_laborforce<-                          # 1   # 1
  indiana %>% group_by(Year) %>%              # 2   # 2
  summarise(Min=min(Labor_force),             # 3   # 3
    p10th=quantile(Labor_force, c(0.1)),      # 4   # 4
    p25th=quantile(Labor_force, c(0.25)),     # 5   # 5
    p50th=quantile(Labor_force, c(0.5)),      # 6   # 6
    p75th=quantile(Labor_force, c(0.75)),     # 7   # 7
    p90th=quantile(Labor_force, c(0.9)),      # 8   # 8
    Max=max(Labor_force),                     # 9   # 9
    Average=mean(Labor_force),               # 10  # 10
    StDev=sd(Labor_force))                   # 11  # 11
```

Year	Min	p10th	p25th	p50th	p75th	p90th	Max	Average	StDev
1990	2629	6639.3	9617.50	14858.5	30089.50	60784.5	424053	30593.65	52989.19
1991	2541	6380.5	9480.00	14713.0	30131.00	62309.9	421789	30357.00	52670.08
1992	2550	6284.2	9615.50	15169.5	31117.00	63623.2	429239	31058.60	53551.02
1993	2607	6200.6	10037.75	15652.5	32034.75	64549.9	438377	32046.84	54683.97
1994	2681	6453.1	10480.50	16472.0	33808.25	66506.3	453457	33308.01	56427.35
1995	2714	6543.2	10994.75	16648.5	34588.00	67364.5	459603	34078.00	57219.27
1996	2724	6522.6	10827.50	16551.0	34597.25	66487.5	453728	33800.80	56670.37
1997	2698	6585.3	10966.00	16671.5	34786.50	66195.3	453928	33919.72	56870.28
1998	2701	6629.7	11008.75	16660.0	34150.50	65887.6	452412	33947.47	56827.82
1999	2691	6496.7	10986.75	16958.5	33968.50	65196.0	448732	33965.91	56434.89
2000	2985	7053.5	10816.50	16984.5	34996.50	63718.6	455135	33982.50	56818.02
2001	2982	7085.7	10709.00	17127.5	34674.75	64832.0	459505	34140.32	57318.59
2002	3051	7040.7	10802.75	17244.5	34974.75	66572.3	462330	34469.34	57807.67
2003	3092	6941.0	10709.50	17199.0	34514.75	67751.5	464331	34597.76	58056.22
2004	3109	6815.2	10547.00	17083.5	33431.50	68476.2	459140	34432.72	57583.09
2005	3152	7075.8	10702.75	17357.5	33606.00	69985.6	459492	34841.75	57925.42
2006	3107	6883.1	10620.50	17326.5	34554.50	72792.0	461468	35164.35	58484.53
2007	3026	6816.7	10447.50	16881.0	34304.50	73128.1	460046	34866.21	58342.88
2008	3044	6816.2	10488.00	16673.5	34065.50	75072.0	463200	35131.49	58819.13
2009	3088	6758.4	10434.25	16452.0	33700.25	74434.1	457524	34717.45	58083.80
2010	3263	6440.5	10308.00	16622.0	33921.75	74406.0	452836	34512.97	57806.16
2011	3210	6459.9	10184.25	16630.5	33804.00	75620.5	456011	34586.91	58199.53
2012	3181	6336.2	10058.50	16453.0	32924.00	76458.0	458377	34454.88	58388.80
2013	3165	6325.9	9900.50	16289.5	33053.00	78149.5	463346	34656.59	58983.65
2014	3216	6401.2	9851.50	16524.0	33215.50	80113.3	467169	35052.00	59579.47
2015	3178	6341.8	9924.75	16908.0	33960.00	82327.7	472330	35508.30	60275.44
2016	3226	6444.8	9930.50	17093.5	35149.00	85223.0	481434	36178.11	61517.07
2017	3153	6512.5	9783.75	17119.0	35516.50	85487.6	483607	36243.96	61840.36
2018	3189	6561.9	9943.50	17334.5	36211.00	86905.8	488698	36756.59	62659.34
2019	3201	6615.6	9812.75	17243.0	36470.50	87146.9	492967	36819.25	63114.77

3.4 Defining New Variables

How about if we don't want to calculate summary statistics within a group, but just want to calculate a new variable from each observation? Consider if we had unemployed and labor force levels but did not have unemployment rate, how could we go about calculating it with mutate?

```
indiana<-                                                          # 1  # 1
  indiana %>%                                                      # 2  # 2
  mutate(unemp_rate_calc=round((Unemployed/Labor_force)*100, digits=1))  # 3  # 3
```

Note since we are defining our new data frame as indiana, the one we are manipulating, we are in this case overwriting the indiana data.frame. Since the

unemployment rate in our file was listed as percent and rounded to the nearest tenth, I did the same for our calculated. (digits=1 means one decimal place)

Now let's compare the first rows of the given unemployment rate with the one we just calculated.

```
## # A tibble: 6 x 2
##   Unemp_rate unemp_rate_calc
##        <dbl>           <dbl>
## 1        6.5             6.5
## 2        5.1             5.1
## 3        4.8             4.8
## 4        3.4             3.4
## 5        9.2             9.2
## 6        2.4             2.4
```

3.5 Dropping Variables

How can we go about dropping or keeping certain variables?

Say we wanted to drop the unemp_rate_calc, Labor_force, and Employed?

```
temp<-                                                  # 1  # 1
  indiana %>%                                           # 2  # 2
  select(-c(unemp_rate_calc, Labor_force,  Employed))   # 3  # 3
head(temp)                                              # 4  # 4
```

```
## # A tibble: 6 x 7
##   LAUS_code    State_fips County_fips County_name    Year  Unemployed Unemp_rate
##   <chr>        <chr>      <chr>       <chr>          <chr>      <dbl>      <dbl>
## 1 CN180010000~ 18         001         Adams County,~ 1990         998        6.5
## 2 CN180030000~ 18         003         Allen County,~ 1990        8317        5.1
## 3 CN180050000~ 18         005         Bartholomew C~ 1990        1632        4.8
## 4 CN180070000~ 18         007         Benton County~ 1990         156        3.4
## 5 CN180090000~ 18         009         Blackford Cou~ 1990         643        9.2
## 6 CN180110000~ 18         011         Boone County,~ 1990         483        2.4
```

Now what if we wanted to keep just State_fips, County_fips, Year, and Unemp_rate?

```
temp<-                                                  # 1  # 1
  indiana %>%                                           # 2  # 2
  select(c(State_fips, County_fips, Year, Unemp_rate))  # 3  # 3
head(temp)                                              # 4  # 4
```

```
## # A tibble: 6 x 4
##   State_fips County_fips Year  Unemp_rate
##   <chr>      <chr>       <chr>      <dbl>
## 1 18         001         1990         6.5
```

```
## 2 18          003          1990          5.1
## 3 18          005          1990          4.8
## 4 18          007          1990          3.4
## 5 18          009          1990          9.2
## 6 18          011          1990          2.4
```

That is if we include the "-" before our variables, we are telling dplyr through our select argument to drop the variables we list. While if we don't have the "-", we are telling dplyr to keep only these variables.

3.6 Merging Datasets

What if we have 2 data sets, both county level. How could we combine them?

3.6.1 Poverty Estimates

We will be merging poverty estimates for the US counties in 2019. A table prepared by the USDA can be found on their website here. Let's download this and put it into our original_data folder. Now let's open it and see what the data looks like. One of the first things we can notice is the first 2 lines, United States and Alabama. That is there is county and state level observations in addition to the county level ones. We need to keep this in mind before we merge with our unemployment data.

Try to figure out how to read this data into R by yourself. If you need to review, go back to Downloading/Reading Data. Remember using R Studio to assist with this is probably the simplest way until we get very comfortable with the commands.

Here is the command I use for it:

```
PovertyEstimates <-                                              # 1  # 1
  read_excel(paste0(folder_raw_data, "/PovertyEstimates.xls"),   # 2  # 2
  skip = 4)                                                      # 3  # 3
```

Make sure we also have our county unemployment data read in. We can observe both of these data.frames now. The way I would clean this data is the following (of course depends on the question/results you are trying to obtain):

- unemployment data:
  1. restrict to 2019 data (as our other county data is just 2019)
  2. Drop Year variable (as we only have one year now)
  3. Drop Puerto Rico (State FIPS 72)
  4. combine state_fips and county_fips into a FIPStxt variable (this is a variable in our poverty data which we will later be using as an ID to merge data on.)
- poverty data:
  1. filter out state/country level observations, leaving just county level.

Here is the code I use for this:

```
unem_2019<-county_data %>%
  filter(Year==2019, State_fips!=72) %>% #restrict sample to only year 2019, drop PR
  select(-Year) %>% #drop the variable Year
  mutate(FIPStxt=paste0(State_fips, County_fips)) #Create variable to merge on

pov_2019 <- PovertyEstimates %>%
  filter(substr(FIPStxt, str_length(FIPStxt)-2, str_length(FIPStxt))!="000")
```

Now if we observe both data frames we are left with what we want to merge. We have defined a variable FIPStxt which we would like to join observations on. That is for a given FIPStxt ID in our unemployment data, we would like to find the row with that FIPStxt ID in our poverty data, then add the variables in the poverty data to our unemployment data. Then do this for all iterations of our FIPStxt ID.

There are join commands for this in the dplyr package. The clearest join command is full_join. This matches data frames based on common variable names (or defined variable names to match on.) I print the first 3 rows below.

```
clean_full<-                          # 1  # 1
  unem_2019 %>% full_join(pov_2019)   # 2  # 2
```

```
## Joining, by = "FIPStxt"
```

LAUS_code	State_fips	County_fips	County_name	Labor_force	Employed	U
CN0100100000000	01	001	Autauga County, AL	26172	25458	
CN0100300000000	01	003	Baldwin County, AL	97328	94675	
CN0100500000000	01	005	Barbour County, AL	8537	8213	

3.6.2 left_join

While the above full_join command is probably the 'safest' route since it prints all of the variables and rows of both data frames, knowing more about other joins can save us some time.

For example, consider above when we first loaded in our poverty data. It had observations at state and country wide levels in addition to the county level data we wanted. We would ideally want a command that took in the county unemployment data, matched the poverty data based on FIPStxt, and then just exclude any extras from the poverty data. We can do this with left (or right) join.

```
unclean_left<-                              # 1  # 1
  unem_2019 %>% left_join(PovertyEstimates)  # 2  # 2
```

```
## Joining, by = "FIPStxt"
```

LAUS_code	State_fips	County_fips	County_name	Labor_force	Employed	Unemployed
CN0100100000000	01	001	Autauga County, AL	26172	25458	714
CN0100300000000	01	003	Baldwin County, AL	97328	94675	2653
CN0100500000000	01	005	Barbour County, AL	8537	8213	324

In this case we are starting with our unemployment data, and then telling R to match all possible values by our common variable (FIPStxt here), then we don't care about any poverty estimate observations that can't be matched. However what about unemployment observations that can't be? We simply fill these values with NA's. For example, if we DON'T exclude Puerto Rico, it would be in our unemployment data, but not in our poverty data. Let's see what happens:

```
unem_2019_pr<-county_data %>%                          # 1  # 1
  filter(Year==2019) %>%                               # 2  # 2
  select(-Year) %>%                                    # 3  # 3
  mutate(FIPStxt=paste0(State_fips, County_fips))      # 4  # 4
pr<-                                                   # 5  # 5
  unem_2019_pr %>%                                     # 6  # 6
  left_join(PovertyEstimates) %>%                      # 7  # 7
  filter(State_fips==72)                               # 8  # 8
```

```
## Joining, by = "FIPStxt"
```

LAUS_code	State_fips	County_fips	County_name	Labor_force	Employed	Unemplo
CN7200100000000	72	001	Adjuntas Municipio, PR	4115	3485	6
CN7200300000000	72	003	Aguada Municipio, PR	11743	10553	1
CN7200500000000	72	005	Aguadilla Municipio, PR	14350	12881	1

3.6.3 Additional Resources

In addition to full and left join shown here, there is also inner join and right join. Details of these can be found here.

Chapter 4

Exporting Results

The goal of this section is to learn how to take the figures/tables we created in the tidyverse section and export them directly into a form we can read into LaTex.

Specifically:

- Xtable
- ggsave

4.1 Saving Plots

See ggplot section in the tidyverse section previously to learn how to generate a plot. Once we have generated a plot we like, let's save it.

To do this we will use the ggsave command.

```
ggsave(file=paste0(folder_figures, "/indiana_unemp.png"))      # 1  # 1  # 1
```

This will take the last graphic in memory and save it in the location we specify. We can also tell ggplot to make the plot specific dimensions which I highly recommend doing. If you do not specify dimensions, they can be dependent on the size of your viewer in RStudio.

```
ggsave(file=paste0(folder_figures, "/indiana_unemp.png"),   # 1  # 1  # 1
  height = 5, width = 7)                                    # 2  # 2  # 2
```

4.2 Tables to LaTex

Let's learn how to export our summary table we created in the dplyr section. Just as a review here is how we created it.

```
indiana_laborforce<-                          # 1   # 1   # 1   # 1
  unemp_data %>%                              # 2   # 2   # 2   # 2
  filter(State_fips=="18") %>%                # 3   # 3   # 3   # 3
  group_by(Year) %>%                          # 4   # 4   # 4   # 4
  summarise(Min=min(Labor_force),             # 5   # 5   # 5   # 5
    p10th=quantile(Labor_force, c(0.1)),      # 6   # 6   # 6   # 6
    p25th=quantile(Labor_force, c(0.25)),     # 7   # 7   # 7   # 7
    p50th=quantile(Labor_force, c(0.5)),      # 8   # 8   # 8   # 8
    p75th=quantile(Labor_force, c(0.75)),     # 9   # 9   # 9   # 9
    p90th=quantile(Labor_force, c(0.9)),      # 10  # 10  # 10  # 10
    Max=max(Labor_force),                     # 11  # 11  # 11  # 11
    Average=mean(Labor_force),                # 12  # 12  # 12  # 12
    StDev=sd(Labor_force))                    # 13  # 13  # 13  # 13
```

Now let's load in the package xtable. This package allows us to print data.frames into a tex table environment. Let's first inspect the output for just running the base function with minimal options.

```
  xtable(indiana_laborforce)  # 1  # 1  # 1  # 1
```

```
## % latex table generated in R 4.1.0 by xtable 1.8-4 package
## % Thu Jul 15 11:46:59 2021
## \begin{table}[ht]
## \centering
## \begin{tabular}{rlrrrrrrrrrr}
##   \hline
##  & Year & Min & p10th & p25th & p50th & p75th & p90th & Max & Average & StDev \\
##   \hline
## 1 & 1990 & 2629.00 & 6639.30 & 9617.50 & 14858.50 & 30089.50 & 60784.50 & 424053.00
##   2 & 1991 & 2541.00 & 6380.50 & 9480.00 & 14713.00 & 30131.00 & 62309.90 & 421789.
##   3 & 1992 & 2550.00 & 6284.20 & 9615.50 & 15169.50 & 31117.00 & 63623.20 & 429239.
##   4 & 1993 & 2607.00 & 6200.60 & 10037.75 & 15652.50 & 32034.75 & 64549.90 & 438377
##   5 & 1994 & 2681.00 & 6453.10 & 10480.50 & 16472.00 & 33808.25 & 66506.30 & 453457
##   6 & 1995 & 2714.00 & 6543.20 & 10994.75 & 16648.50 & 34588.00 & 67364.50 & 459603
##   7 & 1996 & 2724.00 & 6522.60 & 10827.50 & 16551.00 & 34597.25 & 66487.50 & 453728
##   8 & 1997 & 2698.00 & 6585.30 & 10966.00 & 16671.50 & 34786.50 & 66195.30 & 453928
##   9 & 1998 & 2701.00 & 6629.70 & 11008.75 & 16660.00 & 34150.50 & 65887.60 & 452412
##   10 & 1999 & 2691.00 & 6496.70 & 10986.75 & 16958.50 & 33968.50 & 65196.00 & 44873
##   11 & 2000 & 2985.00 & 7053.50 & 10816.50 & 16984.50 & 34996.50 & 63718.60 & 45513
##   12 & 2001 & 2982.00 & 7085.70 & 10709.00 & 17127.50 & 34674.75 & 64832.00 & 45950
##   13 & 2002 & 3051.00 & 7040.70 & 10802.75 & 17244.50 & 34974.75 & 66572.30 & 46233
##   14 & 2003 & 3092.00 & 6941.00 & 10709.50 & 17199.00 & 34514.75 & 67751.50 & 46433
##   15 & 2004 & 3109.00 & 6815.20 & 10547.00 & 17083.50 & 33431.50 & 68476.20 & 45914
##   16 & 2005 & 3152.00 & 7075.80 & 10702.75 & 17357.50 & 33606.00 & 69985.60 & 45949
##   17 & 2006 & 3107.00 & 6883.10 & 10620.50 & 17326.50 & 34554.50 & 72792.00 & 46146
##   18 & 2007 & 3026.00 & 6816.70 & 10447.50 & 16881.00 & 34304.50 & 73128.10 & 46004
```

```
##   19 & 2008 & 3044.00 & 6816.20 & 10488.00 & 16673.50 & 34065.50 & 75072.00 & 463200.00 & 3513
##   20 & 2009 & 3088.00 & 6758.40 & 10434.25 & 16452.00 & 33700.25 & 74434.10 & 457524.00 & 3471
##   21 & 2010 & 3263.00 & 6440.50 & 10308.00 & 16622.00 & 33921.75 & 74406.00 & 452836.00 & 3451
##   22 & 2011 & 3210.00 & 6459.90 & 10184.25 & 16630.50 & 33804.00 & 75620.50 & 456011.00 & 3458
##   23 & 2012 & 3181.00 & 6336.20 & 10058.50 & 16453.00 & 32924.00 & 76458.00 & 458377.00 & 3445
##   24 & 2013 & 3165.00 & 6325.90 & 9900.50 & 16289.50 & 33053.00 & 78149.50 & 463346.00 & 34656
##   25 & 2014 & 3216.00 & 6401.20 & 9851.50 & 16524.00 & 33215.50 & 80113.30 & 467169.00 & 35052
##   26 & 2015 & 3178.00 & 6341.80 & 9924.75 & 16908.00 & 33960.00 & 82327.70 & 472330.00 & 35508
##   27 & 2016 & 3226.00 & 6444.80 & 9930.50 & 17093.50 & 35149.00 & 85223.00 & 481434.00 & 36178
##   28 & 2017 & 3153.00 & 6512.50 & 9783.75 & 17119.00 & 35516.50 & 85487.60 & 483607.00 & 36243
##   29 & 2018 & 3189.00 & 6561.90 & 9943.50 & 17334.50 & 36211.00 & 86905.80 & 488698.00 & 36756
##   30 & 2019 & 3201.00 & 6615.60 & 9812.75 & 17243.00 & 36470.50 & 87146.90 & 492967.00 & 36819
##    \hline
## \end{tabular}
## \end{table}
```

We could then copy and paste this output into latex. However, we can tell xtable and R to save a tex file with this in it. Notice the pathing used. This is a latex table so we are keeping ourselves organized by 1) giving the tex file a useful name and 2) saving it within our tables folder. This may seem extra tedious now but come generating hundreds of plots and tables, organazation can be key to keeping everything straight.

```
  print.xtable(                                                  # 1  # 1  # 1  # 1
    xtable(indiana_laborforce),                                  # 2  # 2  # 2  # 2
    file=paste0(workingdir, "tables/indiana_laborforce.tex"))    # 3  # 3  # 3  # 3
```

{{% callout note %}} Try building this table in a pdf through LaTex {{% /callout %}}

Let's now customize this table more. First, let's get rid of row names as they are not meaningful for us here. Also, if we want to later fix this table in a certain place in a tex document we want a table placement H.

```
print.xtable(                                                  # 1  # 1  # 1  # 1
  xtable(indiana_laborforce),                                  # 2  # 2  # 2  # 2
  table.placement="H",                                         # 3  # 3  # 3  # 3
  include.rownames=FALSE,                                      # 4  # 4  # 4  # 4
  file=paste0(workingdir, "tables/indiana_laborforce.tex"),    # 5  # 5  # 5  # 5
  )                                                            # 6  # 6  # 6  # 6
```

Let's add a bit more customization to this table.

```
print.xtable(                                                          # 1  # 1  # 1  # 1
  xtable(                                                              # 2  # 2  # 2  # 2
  indiana_laborforce,                                                  # 3  # 3  # 3  # 3
  digits=0,                                                            # 4  # 4  # 4  # 4
  label= "tab:indiana_laborforce",                                     # 5  # 5  # 5  # 5
  align = c("|c|","|c|","c","c","c","c","c","c","c","c","c|")),        # 6  # 6  # 6  # 6
```

```
  table.placement="H",                                          # 7  # 7  # 7  # 7
  include.rownames=FALSE,                                       # 8  # 8  # 8  # 8
  file=paste0(folder_tables, "/indiana_laborforce.tex"))        # 9  # 9  # 9  # 9
```

- Line #4: Round all digits to the nearest integer.
- Line #5: add a label to the table which you can then reference to in your main tex document.
- Line #6: if we want to have a custom alignment for our table. (Note: this command requires 1 more column of arguments than there is columns in your table. This is due to it printing the row names. However if you later define include.rownames as FALSE as I do here. It will get rid of the first column of row names.)

There are many other packages that can assist in creating tex tables. We will later use stargazer for reporting regression results. I like using xtable due to it's ease to customize. Please leave me a note below if you have any favorite packages for creating tex tables.