R Data Manipulation Tutorial: Carbon Capture Indicator

Tristan Misko

7 October 2022

Introduction

I've gotten a couple of questions about how to get started learning some of the more high-powered tools like R and python and how to know when to use these tools over something simple like Google Sheets. In most cases, I would recommend using Google Sheets when you can, but sometimes (as in the example below) it would be incredibly tedious to try to do the analysis by hand.

In this document, I work through an example for which producing the indicator by hand or in Google Sheets would be very difficult. I collect and clean the data for the Carbon Capture Indicator in the Land Category of the Sustainability Pillar.
This example is great to learn from because it covers most of the issues you might run into and uses a lot of the main tools you'd need to know to be able to manipulate and clean data to produce a useable indicator in R. I've written this tutorial to be accessible to anyone on the SSPI team, but if anything is unclear, please reach out to me so I can fix it.

I'll probably also upload a regressions tutorial when I rerun the outcome variable regressions in a few days, so look out for that.

Crash Course on R Fundamentals

Computers store objects in memory and run programs which can manipulate those objects into new objects. Our goal is to learn some useful commands and workflows for effectively manipulating dataframe objects. The language we'll be using to communicate which manipulations we'd like the computer to execute is called R.[1]

R is a programming language designed for data manipulation and analysis, and it is typically run in RStudio, an Integrated Development Environment (IDE) designed for data manipulation. RStudio will provide us a nice location to edit and run our code and to see the outputs of our manipulations in real time. To start up, you'll download R and RStudio following the instructions at the end of the document.[2]

R files (files ending in `.R`) store lines of R code. Running such a file executes the code, performing whatever actions author of the file has told R to do.

R Markdown files (files ending in `.rmd`) are super nifty special files which contain chunks of code you can run like an R file along with writing used to communicate what's going on in the code or the larger analysis.[3] When you "knit" an `.rmd` file, the code in the chunks runs, producing whatever data manipulations, tables, and graphs you've instructed R to do, then the written document is compiled and returned. You're actually reading the output of an R Markdown file right now. For the SSPI, `.rmd` files are great for communicating

[1] We'll be using R for a few reasons. First, I and a number of your teammates have some familiarity in R for data analysis, so there will be people around who can support on your way up the learning curve. Secondly, R is a free and open source software used by analysts in academia and industry, so it's a valuable skill to develop. Third, I think R is more beginner friendly than python for data analysis.

[2] Follow this installation guide for more help: https://moderndive.netlify.app/1-getting-started.html

[3] `.r` is to `.py` as `.rmd` is to `.ipynb`

the idea of the kinds of more complex analysis we'd be using R for in a way that's readable for people who aren't used to writing code.

R contains objects and functions. Typical objects for us will be values, vectors, and dataframes:

- Values are numbers, stings of characters, etc.
- Vectors are ordered collections of values of the same type (i.e. all strings, all numbers)
- Dataframes are table-like collections of data values organized into columns (each column is a vector)

We will call R functions on these different types of objects to achieve the results we're looking for. Functions take in **arguments** (objects to act on) and **return** objects.

Below, we'll see many examples of the types of objects described above and we'll get to know many of the standard functions used to manipulate these objects.

One last note: R has a weird notation for variable assignment. Most languages would say `x = 5` to make a variable `x` which has value `5`. You can do this in R, but convention is to use the `<-` operator in R to assign variables. That is, instead of saying `x = 5` in R, you'd say `x <- 5`. Weird, right?

Setting Up the Document

At the top of an R Markdown file, we set up the file we're going to use. We call the special `library` function to import standard libraries used in R. A library is a collection of code and documentations written by someone for other users. This code defines some useful objects and functions we will use below. Calling library will set up this outside code so we can use it as a base atop which we will do our own work.

The last step is to set up our working directory. This is the folder on our computer in which we have our data stored. Every time you import data in R, you'll have to remember to set your working directory. If ever you don't know what your working directory is, you can call `getwd()` to figure it out. To set your own working directory, find the filepath to the folder with your data on your computer, and call the `setwd` function on a string containing the filepath.[4]

```
knitr::opts_chunk$set(echo = TRUE) # this line tells RMD to display the code
library(dplyr) # standardlibrary for manipulating data in R
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
library(ggplot2) # standard library for plotting data in R
library(tidyr) # standard library we'll use to "spread" the data (see below)
library(rmarkdown) # a library for making pretty tables for the document
# set the working directory (the folder on my computer where i'm keeping the data and stuff)
setwd("/Users/tristanmisko/Documents/Berkeley/ECON/SSPI\ Research/Tutorials/2022-10-07\ R\ Data\ Manipul
```

[4] Filepaths look different on MacOS and Windows, so watch out for that!

Importing the Data

To start, we import the list of SSPI countries so that we don't have to type the name of each country by hand. I created a nifty little `.csv` file for this from the main data file online. You can easily create `.csv` file using `File > Download > Comma Separated Values (.csv)` on Google Sheets.

To import data, we call the `read.csv` function. The string `"SSPI_countries.csv"` is the filename of the data we want to import. The `read.csv` function takes the argument `"SSPI_countries.csv"` and returns a dataframe object containing the data from that file. We "pass" the dataframe returned by `read.csv` to the variable `SSPI_countries` using the assignment operator `<-`. Now, whenever we type `SSPI_countries` below, R knows we're talking about the dataframe we pulled down.

```
#"<-" is R's funny notation for "=".  I think "=" also works but I'm an R purist
SSPI_countries <- read.csv("SSPI_countries.csv")
# look at the first few rows of SSPI_countries dataframe object
head(SSPI_countries)
```

```
##    X   Country COU X.1 X.2 X.3 X.4
## 1 NA Argentina ARG  NA  NA  NA  NA
## 2 NA Australia AUS  NA  NA  NA  NA
## 3 NA   Austria AUT  NA  NA  NA  NA
## 4 NA   Belgium BEL  NA  NA  NA  NA
## 5 NA    Brazil BRA  NA  NA  NA  NA
## 6 NA    Canada CAN  NA  NA  NA  NA
```

Notice that the `SSPI_countries` dataframe has a lot of whitespace in it because I was sloppy in making the `SSPI_countries.csv` file. To get rid of that, we're going to `select` the second and third column of the dataframe. We use the weird R notation `c(2,3)` to create (hence the `c`) the vector containing 2 and 3. For our purposes, a vector is just a list of values. Each column and row of a dataframe is a vector, and there are special ways to access each of those in R. More on that later.

The function `select` takes two arguments: our dataframe `SSPI_countries` and the vector containing 2 and 3. Calling `select` on the vector containing 2 and 3 tells R to select the data that is in the second and third column of our dataframe, and to return a dataframe containing only that data in a new dataframe.[5] We then assign that returned value to the variable `SSPI_countries`.

If that seems a little weird at first, that's because it is. Variable names behave differently on the left and right sides of the assignment operator `<-`. Remember: *everything to the right is evaluated (i.e. all the calculations and function calls and returns) before anything on the left side.* In the first line of code below, evaluating the right side means that we access the original dataframe `SSPI_countries` in memory (i.e. the one with all the whitespace), we do our selection (which returns a new dataframe without any of the whitespace), then we tell R to make `SSPI_countries` "point at" the clean new dataframe instead of the old one. Below this line, the variable `SSPI_countries` has in a sense "forgotten about" the old dataframe with all the whitespace—it's no longer in the computer's memory—and we only remember the clean new dataframe. We make use of this kind of operation (called "overwriting a variable") constantly in this file, so it's worth taking a second to make sure that's clear before moving on.

```
# use the select function from the dplyr library to eliminate the whitespace
SSPI_countries <- select(SSPI_countries, c(2,3))
#see how we removed the whitespace
head(SSPI_countries)
```

```
##     Country COU
## 1 Argentina ARG
```

[5] R uses 1-indexing for various reasons. That is, list indices are $1, 2, 3, \ldots$. Most programming languages use 0-indexing (lists go $0, 1, 2, \ldots$) for various other reasons, and programmers get very upset about this kind of stuff. For us, it's not a big deal, but if you're used to python or something like that, just remember R is weird.

```
## 2 Australia AUS
## 3   Austria AUT
## 4   Belgium BEL
## 5    Brazil BRA
## 6    Canada CAN
```

Now we're going to pull in the data we need for the indicator. We use the `nrow` and `ncol` functions to see that the dataframe has 7329 rows and 14 columns. We view the names of the columns with `colnames` and decide that we really only need the columns `"Area"`, `"Year"`, and `"Value"`. We then use our friend the `select` function to pick only these columns. One nice thing about `select` is that it can use indices (like `c(2,3)` above) or column names from our dataframe (like below), and R will figure out which we mean.
We save our selection over the original `carbon` dataframe and look at the result.

```
# import the data
carbon <- read.csv("FAOSTAT_data_en_9-28-2022.csv")
# check the dimensions
c(nrow(carbon), ncol(carbon))
```

```
## [1] 7329   14
```

```
# check column names to select the ones we're going to need
colnames(carbon)
```

```
##  [1] "Domain.Code"      "Domain"           "Area.Code"        "Area"
##  [5] "Element.Code"     "Element"          "Item.Code"        "Item"
##  [9] "Year.Code"        "Year"             "Unit"             "Value"
## [13] "Flag"             "Flag.Description"
```

```
#select the right columns
carbon <- select(carbon, c("Area", "Year", "Value"))
head(carbon)
```

```
##      Area Year   Value
## 1 Algeria 1990 31.1396
## 2 Algeria 1991 30.9831
## 3 Algeria 1992 30.8266
## 4 Algeria 1993 30.6701
## 5 Algeria 1994 30.5136
## 6 Algeria 1995 30.3571
```

We'll also pull in the forest extent data using the same method. We can overwrite the `colnames` of the dataframe with whatever we want by reassigning `colnames(forest)` to a vector of the appropriate length containing our desired names. We can use `select` with a negative condition too: below we `select` the columns which are not (`!`) `FL_2010`, `FL_2016`, `FL_2017`, and `FL_2019`.

```
# import the forest data
forest <- read.csv("fra2020-extentOfForest.csv")
#look at the column names
colnames(forest)
```

```
##  [1] "X"                    "Forest...1000.ha.."   "Forest...1000.ha...1"
##  [4] "Forest...1000.ha...2" "Forest...1000.ha...3" "Forest...1000.ha...4"
##  [7] "Forest...1000.ha...5" "Forest...1000.ha...6" "Forest...1000.ha...7"
## [10] "Forest...1000.ha...8"
```

```
#rename them looking at the online database so we can tell what we're talking aobut
colnames(forest) <- c("Area", "FL_1990", "FL_2000", "FL_2010", "FL_2015", "FL_2016", "FL_2017", "FL_2018
# select only the potentially relevant years
forest <- forest %>% select(!c("FL_2010", "FL_2016", "FL_2017", "FL_2019"))
```

```
head(forest)
```

```
##                    Area FL_1990 FL_2000 FL_2015 FL_2018 FL_2020
## 1                          1990.00 2000.00 2015.00 2018.00 2020.00
## 2           Afghanistan 1208.44 1208.44 1208.44 1208.44 1208.44
## 3 Albania (Desk study)  788.80  769.30  789.19  788.90  788.90
## 4               Algeria 1667.00 1579.00 1956.00 1930.00 1949.00
## 5        American Samoa   18.07   17.73   17.28   17.19   17.13
## 6               Andorra   16.00   16.00   16.00   16.00   16.00
```

Carbon Stock Data

Cleaning and Filtering the Data

Next up, we're going to filter down the `carbon` data to only the SSPI countries using the `SSPI_countries` dataframe we imported earlier. To achieve this, we use the aptly named function `filter` from the `dplyr` library. The `filter` function requires a logical vector to work, i.e. for each row in the data it needs to know whether that row is in (`TRUE`) or out (`FALSE`).

For our purposes, we want to know whether the `Area` value in the dataframe (which indicates the country name) is in our list of SSPI Countries. We use the special vector operator `%in%`, which checks the element on the left of `%in%` is contained in the vector on the right of `%in%`. We make the vector `countries` using the special `$` notation for dataframes. The notation `SSPI_countries$Country` tells R to look into the `SSPI_countries` dataframe, grab the column `Country`, and stick the result into a vector, which we've chosen to call `countries`.
This vector has `nrow(SSPI_countries)`=49 entries, one for each row of the dataframe.

There are a couple of subtleties to watch out for when doing this with country names. Usually we prefer to use the three letter country codes standardized by the UN when we can because **some countries have multiple names**. In this data, there are no country codes, so we have to make do with names. The usual multiple name suspects in the SSPI are Korea, Rep. (aka. South Korea, Republic of Korea, or simply Korea), Russian Federation (often just listed as Russia), the United States (which sometimes goes by United States of America, US, and USA depending on the dataset), and Slovak Republic (aka. Slovakia). It's always important to check that you've got all the countries once you've done a big filtering operation. To do this, we tell R to list out all of the unique `Area` names after filtering down to the countries whose names are contained in the vector `countries`.

```
# make a vector containing the SSPI_country Country names
countries <- SSPI_countries$Country
# list out the unique country names in the filtered datset
unique(filter(carbon, Area %in% countries)$Area)
```

```
##  [1] "Argentina"           "Australia"           "Austria"
##  [4] "Belgium"             "Brazil"              "Canada"
##  [7] "Chile"               "China"               "Colombia"
## [10] "Denmark"             "Estonia"             "Finland"
## [13] "France"              "Germany"             "Greece"
## [16] "Hungary"             "Iceland"             "India"
## [19] "Indonesia"           "Ireland"             "Israel"
## [22] "Italy"               "Japan"               "Latvia"
## [25] "Lithuania"           "Luxembourg"          "Mexico"
## [28] "Netherlands"         "New Zealand"         "Norway"
## [31] "Poland"              "Portugal"            "Russian Federation"
## [34] "Saudi Arabia"        "Singapore"           "Slovenia"
```

```
## [37] "South Africa"          "Spain"                 "Sweden"
## [40] "Switzerland"           "United Arab Emirates" "Uruguay"
```

We can see that we're missing seven countries. As we find below, we have the usual suspects and a few others have alternate spellings. For these, we look through the data and see how they are encoding the names, then we simply add those names to our `countries` vector so that when we run the filter again, `%in%` will return `TRUE` instead of `FALSE`.

```
# check what the right names for the SSPI countries are
unique(carbon$Area)
# add the name used in the data to the list
countries <- c(countries, "Republic of Korea",
               "United Kingdom of Great Britain and Northern Ireland",
               "United States of America", "Slovakia", "Türkiye", "Czechia")
# filter again and check to see that we have 48/49 countries
unique(filter(carbon, Area %in% countries)$Area)
```

Now only Kuwait is missing. This makes sense: there aren't any forests in Kuwait! We'd deal with this issue in a meeting with the Professor.

Now we can run our final filter on the data, overwriting the original `carbon` dataframe with only the data we need. We have now reduced our original 7329×14 dataframe full of irrelevant countries and variables to a much more managable 732×3 dataframe containing only the countries and variables we actually want. This process gets referred to as "cleaning" or "selecting" and "filtering" the data.

```
# run the filter for countries
carbon <- filter(carbon, Area %in% countries)
# run the filter for years
carbon <- filter(carbon, Year <= 1999 | Year >= 2015)
# check out how many observations we have left
nrow(carbon)
```

```
## [1] 732
```

Grouping and Averaging the Data

Now we're ready to compute 1990s averages. General strategy: define a new variable called `modYear` which equals "1990s" if the year is in the 1990s, otherwise it's just the year. We can then call `group_by` and `summarize`, two special and especially confusing `dplyr` functions, which we'll use to group the observations into buckets based on their `modYear` and `Area`. For years outside of the 1990s, there will only be one observation in the bucket: we only have observation for each country in each year, so the average of that one observation will just be the value for the given year. For years in the 1990s, there are ten observations for each country in the 1990s, all with a value of `"1990s"` for modYear, so computing the average across a group for a given `Area` and `modYear == "1990s"` will return the average of all ten observations in the group.

This step is definitely not trivial, and there are many potential approaches.
This one might not be the clearest, but it's important to get some exposure to `group_by` and `summarize` because they are some of the most useful functions for data manipulation in R (well, actually in `dplyr`).

```
# define the modYear function which takes in a vector of years and returns the vector
# of years with all 1990s values replaced by "1990s"
modYear <- Vectorize(function(year){
    if (year <= 1999 & year >= 1990){
        return("1990s")
    } else {
        return(as.character(year))
    }
```

```
}, "year")

# apply the modYear function to the Year column of carbon, and store the results
# in a new column of carbon called modYear
carbon$modYear <- modYear(carbon$Year)
#look at the the table of carbon to check that we got the expected result
head(carbon)
```

```
##        Area Year    Value modYear
## 1 Argentina 1990 3664.384   1990s
## 2 Argentina 1991 3653.088   1990s
## 3 Argentina 1992 3641.791   1990s
## 4 Argentina 1993 3630.495   1990s
## 5 Argentina 1994 3619.198   1990s
## 6 Argentina 1995 3607.902   1990s
```

```
# use the group_by and summarize operators to make a new dataframe which does
# the averaging and removes the irrelevant years
carbon <- carbon %>% group_by(Area, modYear) %>% summarize(Value = mean(Value, rm.na = T))
```

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

```
head(carbon)
```

```
## # A tibble: 6 x 3
## # Groups:   Area [1]
##   Area      modYear Value
##   <chr>     <chr>   <dbl>
## 1 Argentina 1990s   3614.
## 2 Argentina 2015    3216.
## 3 Argentina 2016    3210.
## 4 Argentina 2017    3202.
## 5 Argentina 2018    3194.
## 6 Argentina 2019    3187.
```

Above we used the `dplyr` pipe operators `%>%`, which might look scary at first but which are actually quite nice. The way to read the last line to the right of the `<-` assignment operator is "take the `carbon` dataframe, send it into the `group_by` function, grouping observations which have the same `Area` and `modYear` value, then send that grouped data into the `summarize` function, which makes a new grouped dataframe which has the group identifier variables `Area` and `modYear` and assigns `Value` (in the new grouped dataframe) to be the `mean` of `Value` across the group (where `Value` is taken from the original `carbon` dataframe).

Looking at `head(carbon)`, we can now see that we have a 1990s average and all the years following 2015. Sweet!

Spreading to Do Computations

We now want to compute percent changes, which is going to require us to reorganize the dataframe. In particular, we want to have a single row for each `modYear` so that we can easily compute the averages for all countries at once using column operations.

To achieve this, we're going to use the `spread` function from the `tidyr` library. [6] Specifically, we are going

[6] When people say R is an "ugly" language, this is what they mean. Basic things like spreading data require the use of an outside libary, and the names of functions across the libraries are not consistent and sometimes conflict. A "pretty" language would have all this built-in and standard, but that can add complexity. It's all tradeoffs :/

to spread `Value` across a different column for each value of `modYear`.

In the language that the `spread` function uses, `Value` is the "value" and `modYear` is the "key". The idea of using "key-value pairs" to identify objects and define functions is one of the most important in computer science. Within each `Area`, we have a set of keys (each observation of `modYear`) which each correspond to a single `Value`, and we're going to spread the keys across the columns by calling `spread(dataframe, key, value)`.

```
# run the spread function and store the value as carbon
carbon <- spread(carbon, modYear, Value)
head(carbon)
```

```
## # A tibble: 6 x 8
## # Groups:   Area [6]
##   Area      `1990s`  `2015`  `2016`  `2017`  `2018`  `2019`  `2020`
##   <chr>       <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
## 1 Argentina   3614.  3216.   3210.   3202.   3194.   3187.   3179.
## 2 Australia   8393.  8292.   8292.   8293.   8292.   8292.   8292.
## 3 Austria      351.   399.    401.    403.    405.    406.    408.
## 4 Belgium       NA     71.8    71.8    71.8    71.8    71.8    71.8
## 5 Brazil     57421. 52505.  52367.  52210.  52151.  52060.  51981.
## 6 Canada     20586. 19591.  19579.  19576.  19574.  19572.  19570.
```

A quick look at the data shows that we've almost got everything we want, but there's an issue with Belgium and Luxembourg: they don't have a 1990s average! What's going on? To find out, we've got to go back and look at our original data, but that's all been overwritten. [7]

What might we do? Well, we can import the original data again into a separate dataframe and look at Belgium and Luxembourg values. Paging through the data is one solution, but I'm going to use some `regex` to find the issue.

We use `regex` whenever we are trying to match strings. The term `regex`, a portmanteau for "regular expressions," refers to a collection of procedures that dates back to the early days of computer science.[8] Fun fact: I've actually used `regex` in every research project that I've done, so it's definitely a useful little topic to master.

For our purposes, we'll use the function `grepl(pattern, string)`, which checks whether or not `string` contains `pattern`. There are a lot of special ways to make patterns to account for very complex cases, but for us we'll just use the country names we're trying to find. In R, `grepl` is vectorized, which means that we can seach across a vector of strings (like the column `fix_bnl$Area`) and the function will return a vector of the same length containing `TRUE`s and `FALSE`S in the appropriate positions.[9]

```
# import the data into a new dataframe
fix_bnl <- read.csv("FAOSTAT_data_en_9-28-2022.csv")
# define a logical vector that is true if the Area name contains Belgium or Luxembourg and
# stor it as a column in fix_bnl dataframe
fix_bnl$contains_bnl <- grepl("Belgium", fix_bnl$Area) | grepl("Luxembourg", fix_bnl$Area)
# select the relevant columns and then filter for the columns which contain_bnl
fix_bnl <- fix_bnl %>% select(c("Area", "Year", "Value", "contains_bnl")) %>%
    filter(contains_bnl)
# look at the data
head(fix_bnl)
```

```
##      Area Year   Value contains_bnl
```

[7] For you reading this document, you can actually still see the original data in the paged output at the top of the sheet. Can you spot the issue there before reading on?

[8] A bit of `regex` history https://www.youtube.com/watch?v=NTfOnGZUZDk

[9] When defining the `modYear` function above, I manually vectorized using `Vectorize` it so it could do a whole column at once. Any user-defined function can be vectorized using `Vectorize(userdefined-function, "vectorized_variable")`

```
## 1 Belgium 2000 59.7300         TRUE
## 2 Belgium 2001 60.5957         TRUE
## 3 Belgium 2002 61.4614         TRUE
## 4 Belgium 2003 62.3271         TRUE
## 5 Belgium 2004 63.1927         TRUE
## 6 Belgium 2005 64.0584         TRUE
```

We can see that Belgium and Luxembourg get lumped together for the 1990s. This now presents us with a few options, which we would likely have a meeting with the Professor and potentially an expert.

1. We could throw out the countries and say we don't have data, but we could almost certainly do better.
2. We could go looking for better data, but we're probably not going to find any. This is a UN dataset with tons of information for most countries. If the UN couldn't find the data, you probably can't either.
3. We could just give Belgium and Luxembourg the same combined values as their 1990s average, but this would penalize both countries because they're comparing to too high a baseline.
4. We could try to **impute** a value making a few reasonable assumptions about the data. This is the approach we take below.

Imputing Values for Belgium and Luxembourg

So, we've got combined Belgium-Luxembourg data through 1990-1999. To make a 1990s average for Belgium and Luxembourg, the approach that seems best to me is to make the average 1990s average for the combined data, then to divide that average between Belgium and Luxembourg based on the 2000 proportions, which is the first year for which we have data. This might not be exactly right, but if carbon stock in forest area is proportional to land area, then it should be pretty close to right. Mostly, it seems to be about as good as we can do with the data we have, so we'll procede with caution.

To determine whether this is even a good idea, we might look to see whether the ratio between Belgium and Luxembourg is roughly constant over time, which would justify this kind of step. To do this, we're going to group the observations by `Year` (using `group_by`) and `summarize` the quotient of the `first Value` in the group (the `Value` corresponding to Belgium in the given group `Year`) and the `last Value` in the group (corresponding to Luxembourg).

```
ratios <- fix_bnl %>% group_by(Year) %>% summarize(ratio = first(Value)/last(Value))
head(ratios)
```

```
## # A tibble: 6 x 2
##    Year ratio
##   <int> <dbl>
## 1  1990     1
## 2  1991     1
## 3  1992     1
## 4  1993     1
## 5  1994     1
## 6  1995     1
```

Looking at the output of this grouping, which we've stored in the grouped dataframe `ratios`, we can see that the ratio is declining over time, but not by that much. This means that Luxembourg's carbon stock in forests has been increasing faster than Belgium's over the last 20 years.

With this information, we have to make a decision. Do we want to use the average ratio over time or the ratio in 2000?

- Using the average ratio would imply that we think that the ratio in the 1990s could have been higher or lower than it was in 2000, and we don't really have any better information.

- Using the 2000 ratio would imply that we think there's a pattern over time and that using 2000 carries more information about what the likely ratio was through the 1990s than the average through the 2000s and 2010s.

I personally think the second option is better because of the aforementioned steady decline in the ratio over time, but because they're both between 8 and 9 it won't really matter that much. We could quantify that intuition using a sensitivity test if we wanted, but for now I'm going to run with the 2000 ratio.

To find the appropriate weights, we're going to go back to the `fix_bnl` dataframe and find the 2000 value for Belgium and Luxembourg using `filter`. Assigning the result to `bl2000`, we can now pull out the `Value` column into a vector and divide this vector by the `sum` across the vector to obtain a new vector containing the proportions of the 2000 total across Belgium and Luxembourg held by Belgium and Luxembourg respectively.

```
# filter the fix_bnl dataframe to be only year 2000
bl2000 <- fix_bnl %>% filter(Year == 2000)
# check to make sure we filtered correctly
head(bl2000)
```

```
##         Area Year   Value contains_bnl
## 1    Belgium 2000 59.7300         TRUE
## 2 Luxembourg 2000  6.9126         TRUE
```

```
# compute the weights: noitice when we divide and multiply vectors by constants,
# all the elements of the vector get divided/multiplied by that constant.
w <- bl2000$Value/sum(bl2000$Value)
# look at the weights: Belgium is much bigger, so it gets higher weights (checks out)
w
```

```
## [1] 0.8962736 0.1037264
```

Next, we compute the 1990s average. By now you probably know the drill: `filter` down the data, pull out the column, record the mean.

```
# filter data down to combined Belgium-Luxembourg values ranging 1990 to 1999 only
bnl_1990s <- fix_bnl %>% filter(Area == "Belgium-Luxembourg")
# compute the mean value across the 1990s for combined Belgium-Luxembourg
bnl_1990s_mean <- mean(bnl_1990s$Value)
# print out the Belgium-Luxembourg 1990s mean
bnl_1990s_mean
```

```
## [1] 59.54124
```

Finally we find the imputed values by multiplying the 1990s mean by the weights:

```
# multiply the weight vector w by the 1990s mean
imputations <- bnl_1990s_mean*w
# print out the result
imputations
```

```
## [1] 53.365239  6.176001
```

As a final sanity check, we look back at the 2000 values to see that our imputations aren't so far off. Sure enough, the 2000s values are a bit larger for Belgium and Luxembourg than the 1990s average, but we also know that the combined Belgium-Luxembourg average was growing through the 1990s, so these numbers seem pretty good.

We now want to stick these values into our `carbon` dataframe in the appropriate slots. To do this, we'll use a concept called *logical indexing* or *logical subsetting*. One nice feature of R vectors is that you can select items from a vector v using a logical vector of the same length as v. This is *incredibly* useful when working with big datasets, especially for filling in missing values.

What we want to do is assign our imputed values to the correct positions in the `carbon` dataframe, i.e. the positions which currently have an NA value. To find the elements which are currently `NA`, we use `is.na` called on our vector (which, recall, is the 1990s column of the `carbon` dataframe). This returns a logical vector of the same length as `carbon`'s columns which we can use to do subsetting.

As in most programming languages, subsetting in R is carried out using `[]`. For example, if `a <- c(2,3,4)`, then `a[1]` is `2` and `a[c(TRUE, FALSE, TRUE)]` is the vector `c(2,4)`.[10]. Doing logical subsetting for our NAs then putting in an assignment statement will overwrite the elements selected in the logical subsetting with whatever you assign them to be on the right hand side.

```
#our logical vector: backticks are used to specify numeric column names
is.na(carbon$`1990s`)
```

```
##  [1] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [25] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [37] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
# using our logical vector to select the appropriate elements of our vector
carbon$`1990s`[is.na(carbon$`1990s`)]
```

```
## [1] NA NA
```

```
# using our logical vector to overwrite the appropriate elements of our vector
carbon$`1990s`[is.na(carbon$`1990s`)] <- imputations
# check to see that our fix worked (huzzah!)
head(carbon)
```

```
## # A tibble: 6 x 8
## # Groups:   Area [6]
##   Area      `1990s`  `2015`  `2016`  `2017`  `2018`  `2019`  `2020`
##   <chr>       <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
## 1 Argentina  3614.   3216.   3210.   3202.   3194.   3187.   3179.
## 2 Australia  8393.   8292.   8292.   8293.   8292.   8292.   8292.
## 3 Austria     351.    399.    401.    403.    405.    406.    408.
## 4 Belgium      53.4    71.8    71.8    71.8    71.8    71.8    71.8
## 5 Brazil    57421.  52505.  52367.  52210.  52151.  52060.  51981.
## 6 Canada    20586.  19591.  19579.  19576.  19574.  19572.  19570.
```

Now the hole in the data is fixed!

Forest Data

We now `filter` the forest data to eliminate the non-SSPI countries. I'll go much quicker in this section. See if you can follow my code.

```
# create a logical vector to see which countries are missing
check_inclusion <- unique(countries) %in% unique(filter(forest, Area %in% countries)$Area)
# present the missing countries in a human readable format
head(arrange(data.frame(Area = unique(countries), check_inclusion), Area))
```

```
##        Area check_inclusion
## 1 Argentina            TRUE
## 2 Australia            TRUE
```

[10] Really it's the vector containing 2 and 4. The command `c(2,4)` is what the user would use to create such a vector, but it isn't the name of/notation for the vector itself. Unfortunately, there's not really a standard written notation for an R vector, so we're stuck with boring footnotes and bad notation. Just another way in which R is "ugly"

```
## 3   Austria            TRUE
## 4   Belgium            TRUE
## 5    Brazil            TRUE
## 6    Canada            TRUE
```

We're missing a three countries: UAE, Kuwait, and Saudia Arabia. Once again, we see that the Middle Eastern countries which are mostly in desert climates don't fit well with our forest-based indicator. We make a new dataframe and `rbind` (i.e. rowbind) it to the end of our real dataframe, then run the final filtering, checking that we get all 49 countries.

```
#make an additional dataframe, defining the column names and the vectors they contain
additional_data <- data.frame(Area = c("Kuwait", "Saudi Arabia", "United Arab Emirates"),
                              FL_1990 = c(0,0,0),
                              FL_2000 = c(0,0,0),
                              FL_2015 = c(0,0,0),
                              FL_2018 = c(0,0,0),
                              FL_2020 = c(0,0,0))
# check out our new extra dataframe
head(additional_data)
```

```
##                  Area FL_1990 FL_2000 FL_2015 FL_2018 FL_2020
## 1              Kuwait       0       0       0       0       0
## 2        Saudi Arabia       0       0       0       0       0
## 3 United Arab Emirates       0       0       0       0       0
```

```
# rowbind the additional zeros into the original dataframe
forest <- rbind(forest, additional_data)
# see that we've done what we expected
head(forest)
```

```
##                   Area FL_1990 FL_2000 FL_2015 FL_2018 FL_2020
## 1                      1990.00 2000.00 2015.00 2018.00 2020.00
## 2          Afghanistan 1208.44 1208.44 1208.44 1208.44 1208.44
## 3 Albania (Desk study)  788.80  769.30  789.19  788.90  788.90
## 4              Algeria 1667.00 1579.00 1956.00 1930.00 1949.00
## 5       American Samoa   18.07   17.73   17.28   17.19   17.13
## 6              Andorra   16.00   16.00   16.00   16.00   16.00
```

```
# run the filter and overwrite forest
forest <- forest %>% filter(Area %in% countries)
# check that the number of rows is right
nrow(forest)
```

```
## [1] 49
```

Merging `carbon` and `forest`

Ultimately, we want everything in a single dataframe so that we can do computations easily. We will execute a `merge`, matching each observation (i.e. each row) based on the `Area` column. The strings must match exactly, otherwise the `merge` won't work. Luckily, the country names match up because both dataframes are from the same source, FAO.

```
# create the merged dataframe
carbon_capture <- merge(carbon, forest)
# check that it looks right
head(carbon_capture)
```

```
##        Area       1990s       2015       2016       2017       2018       2019
## 1 Argentina  3613.55003  3215.5095  3209.6664  3201.7202  3194.3614  3186.7459
## 2 Australia  8393.23167  8291.7874  8291.5412  8292.9967  8292.2356  8292.2356
## 3   Austria   351.40600   398.5982   400.5607   402.5640   404.5314   406.4627
## 4   Belgium    53.36524    71.8251    71.8251    71.8251    71.8251    71.8251
## 5    Brazil 57421.23813 52504.7962 52367.1630 52209.5630 52150.8713 52059.7671
## 6    Canada 20585.75230 19591.2107 19578.5594 19576.4728 19574.3868 19572.3002
##         2020   FL_1990   FL_2000   FL_2015   FL_2018   FL_2020
## 1  3179.0320  35204.00  33378.00  29097.00  28791.00  28573.00
## 2  8292.2356 133882.20 131814.10 133094.50 134005.10 134005.10
## 3   408.4360   3775.67   3838.14   3881.19   3891.97   3899.15
## 4    71.8251    677.40    667.30    689.30    689.30    689.30
## 5 51981.1735 588898.00 551088.60 503884.80 499051.40 496619.60
## 6 19570.2141 348272.93 347801.97 347115.71 347002.07 346928.10
```

And just like that, we've got everything we need! We'll just do a quick cleanup, renaming the columns with `colnames` so that we know what's what and dropping the 2015, 2016, 2017, and 2019 columns from the `carbon` dataframe using `select`.

```
# drop unnecessary columns
carbon_capture <- carbon_capture %>% select(!c("2016","2017","2019"))
# rename the carbon columns so we know what they are
colnames(carbon_capture)[2:5] <- c("C1990s", "C2015", "C2018","C2020")
# view the resulting dataframe to make sure everything's looking good
head(carbon_capture)
```

```
##        Area      C1990s      C2015      C2018      C2020   FL_1990   FL_2000
## 1 Argentina  3613.55003  3215.5095  3194.3614  3179.0320  35204.00  33378.00
## 2 Australia  8393.23167  8291.7874  8292.2356  8292.2356 133882.20 131814.10
## 3   Austria   351.40600   398.5982   404.5314   408.4360   3775.67   3838.14
## 4   Belgium    53.36524    71.8251    71.8251    71.8251    677.40    667.30
## 5    Brazil 57421.23813 52504.7962 52150.8713 51981.1735 588898.00 551088.60
## 6    Canada 20585.75230 19591.2107 19574.3868 19570.2141 348272.93 347801.97
##     FL_2015   FL_2018   FL_2020
## 1  29097.00  28791.00  28573.00
## 2 133094.50 134005.10 134005.10
## 3   3881.19   3891.97   3899.15
## 4    689.30    689.30    689.30
## 5 503884.80 499051.40 496619.60
## 6 347115.71 347002.07 346928.10
```

Finishing Up the Computations

Finally, we can do the computations for the indicator. From the indicator table, we know we are trying to get the "percentage change in the ratio of carbon stock in living biomass over forest land from a 1990s average." First, we compute an approximate 1990s average for forest land by averaging the 1990 and 2000 value. Next, we compute the ratios for 1990s and 2018. Finally, we compute the percentage change in the ratios. All that work above for a couple lines of code down here.

```
# compute an average of forest land for 1990s
carbon_capture$FL_1990s <- (carbon_capture$FL_1990 + carbon_capture$FL_2000)/2
# compute the ratio of carbon stock per forest land for the 1990s
carbon_capture$r1990s <- carbon_capture$C1990s/carbon_capture$FL_1990s
# compute the ratio of carbon stock per forest land for 2018
```

```
carbon_capture$r2018 <- carbon_capture$C2018/carbon_capture$FL_2018
# compute the percentage change in ratios
carbon_capture$pchg_1990s_to_2018 <- (carbon_capture$r2018 - carbon_capture$r1990s)/carbon_capture$r199
```

Exporting the Data

Once you've done your computations, use `write.csv` to create a new `.csv` file in the working directory with your data in a sharable format. Huzzah! We have the raw data for an indicator ready to go.

```
# write the dataframe to a new csv file which we can upload
write.csv(carbon_capture, "carbon_capture.csv")
```

Getting Your Hands Dirty

Getting Started with R

1) Download R and RStudio and follow the setup steps.
2) Immediately change the Editor Theme to "Chaos" in the Appearance tab of Preferences so that you don't sear your retinae with the white background.

3) In the console, type `"install.packages(c("dplyr", "ggplot2", "tidyr", "rmarkdown"))`, which will pull down all of the packages we need for R from CRAN (an internet repository) and store them on your machine. If you don't do this, then calling, say, `library(dplyr)` will send your computer on a wild goose chase looking for a package you don't have installed.
4) Download the `r-for-SSPI-tutorial-1` directory from the google drive, and store it in a folder you know the filepath to. You see my filepath is long because it's stored under many layers of folder organization. If you're having trouble, try putting it on your Desktop, which usually has a fairly simple filepath.
5) Set your working directory to the folder you've just downloaded by changing the filepath in the `setwd` function in line 17 to whatever the right filepath is for your machine. If you're working on Windows, maybe do some googling to see what the correct formatting is for non-Unix (which is to say silly—don't @ me) filesystems.

A Note on Why We're Doing This

High powered tools like R and python are great because they are **reproducable** and **extensible**. For a research team like ours, reproducability is key because we want to be able to see what we've done and catch any mistakes along the way.
Writing code to do our cleaning/analysis (as opposed to doing it in Google Sheets or Excel) means that we have a recorded set of instructions to reproduce the analysis and we can easily check to see if we've made any mistakes along the way. The exercises below will demonstrate why extensibility is great.

Exercises

1. Compute the percentage increase from 1990s to 2020 and store it in a separate column called `pchg_1990s_to_2020` using the same technique we used to compute `pchg_1990s_to_2018` above.
2. Repeat this analysis to include the SSPI Extended Countries. How do we change which countries are included in the analysis? What file needs to be changed? Use this file to get an output dataset which

has the original SSPI 50 countries (actually 48 for us) and the 18 extended countries. Troubleshoot any issues that crop up with country names, missing data, etc.

3. Right now we are using a 1990s (1990-1999) average and comparing it to 2018 data.
   If we want to get a comparable indicator for 2020, we might want to consider using the average from (1992-2001) as the baseline. Change the code I've provided so that the only `modYear`s we group by are `"1992-2001"` and `"2020"`, and return a new version of the final dataframe with this new version.

Find time to work as a group on these exercises and help each other! If you're having trouble, try googling it. Usually hundreds of other people have had the same issue.

If you're really stuck, email me at tjmisko@berkeley.edu with your questions. Finally, remember, you can always download this file again if you feel like it's gotten messed up.