Coding Bootcamp Part 1

January 16, 2023

1 EEP/IAS 118 - Coding Bootcamp, Part 1

1.1 Introduction to R and Jupyter Notebooks

1.1.1 Section for Week 1 (No in-person, synchronous instruction)

1.1.2 Accompanying Recording available on bCourses under 'Media Gallery'

Note: The Recording is from 2022; as you follow through the bootcamp and the video, any small discrepancies will be pointed out via blue text.

1.2 EEP/IAS 118 Coding

This term we'll be using the programming language **R** for all our data visualization and econometric analysis. As we'll see this semester, **R** is an extremely powerful open-source system that lets us do just about anything we can think of with our data.

There are a couple of different ways to run **R**, and in the first two weeks we'll teach you two main approaches:

1. Using **R** within *Jupyter Notebooks* and **RStudio** hosted remotely on Berkeley's **Datahub**
2. Installing and running **R** with **RStudio** offline on your personal computers

For the sake of this class, option **1.** is the simplest in terms of the analysis we'll be doing this semester, but for independent research and future work, option **2.** is worth knowing as well.

1.3 Jupyter Notebook Basics

We're going to start off in week 1 with option 1 by learning how to use Jupyter Notebooks to run R. Next week we'll show you how to access RStudio through Datahub and walk you through installing R and RStudio on your personal computers.

Jupyter notebooks are an interactive computing environment that lets us combine both text elements and active R code. Importantly for us, notebooks let us

- Write and run R code through our web browsers, and
- Add narrative text that describes our code and the output from said code

This means that we can use R without having to install any software on our personal computers, and can ignore errors that might pop up from local conflicts or issues with packages on personal machines. Further, it lets us answer written exercises and coding problems all in one place. This is especially convenient when it comes to submitting homework assignments, as you can save all your code, output, and writeup and save it all directly in the same pdf.

When you clicked the Bootcamp Part 1 link on bCourses, you were taken to a folder in your web browser. This folder is hosted on a remote web server on Berkeley's Datahub and should look something like this:

Datahub Folder

This is the main folder for course materials, and we'll add a bunch of files and folders over the course of the semester for lectures, daily assignments, and problem sets.

Today we are doing **Coding Bootcamps**, so you will want to click on that folder.

You should see the following files:

Coding Bootcamps Folder

- The file path at the top (`ENVECON-118-SP23/1_CodingBootcamps`) shows us that we navigated to the *Coding Bootcamps* folder within the main course folder. You can navigate back to a specific folder at any time by clicking on its name in the file path (always make sure to save your work before leaving a notebook!).
- *Coding Bootcamp Session Part 1.ipynb* is our notebook for today. You can open it by clicking on the file name. There's also an accompanying recording working through the notebook available at this link (also accessible on bCourses under **Media**)
- The other *Coding Bootcamp Session Part X.ipynb* files are our other Bootcamp notebooks for you to work through later on in the course (we'll point out in section when we'll cover tools for a given problem set)
- *autos.csv* is a data file in comma separated value (csv) format
- *autos.dta* is the same data file, but formatted for Stata in .dta format
- *sleep75.dta* and *wateruse.dta* are two other .dta files we'll use in later bootcamps
- *Readme.md* is a readme file for viewing elsewhere - ignore these whenever they appear

We'll be using both csv and dta data this term, so we'll practice reading in both formats.

(There is also a folder of images…don't worry about those!)

1.3.1 Running Jupyter Notebooks

Double clicking on the *Coding Bootcamp.ipynb* notebook will open up the notebook in another browser window. The notebook that opens should look … well, like this!

At the top of the page you'll find the menubar and toolbar:

From this menu we can run our code, display our text, save our notebook, and download a pdf or other format of our notebook.

Clicking on **File** in the menubar lets us save the notebook and build a checkpoint (you can also use the picture of a floppy disk just below **File** to save your notebook). **Revert to Checkpoint** lets you go back to a previously saved version of the notebook (useful in case you lose track of code changes and want to go back to a previous working version).

Print Preview lets you view the notebook as it would look when printed. You can then hit `control + p` and select **Save to PDF** or **Adobe PDF** to save a pdf copy of the notebook.

You can also use the **Download As > PDF via HTML** command to export a pdf copy of the notebook.

Note: make sure to save your notebooks often to preserve your work in case of connectivity loss!

When you submit your pdf version of assignments on gradescope, you are responsible for making sure your pdf displays everything you want the graders to see (i.e. both codes and outputs; text and codes are wrapping to the next line rather than exiting the page to the right.)

1.3.2 Editing Cells

This and all notebooks are comprised of a linear collection of boxes, called *cells*. For the sake of this class, we'll be working with two types of cells: *Markdown* cells for text, and *Code* cells for writing and executing **R** code.

Markdown cells support plain text as well as markdown code, html, and LaTeX math mode. For this class, plain text answers are totally fine. If you want to dive into Markdown formatting, this cheat sheet has information on formatting text, building tables, and adding html. If you want to add pretty math equations, see this LaTeX Math Mode Guide.

Select a cell by right clicking on it. A grey box with a blue bar to the left will appear around the cell, like so:

You are now in **command mode**. In command mode you can see the cell type (whether it is markdown for text or code for **R** commands) in the toolbar but can't edit the content of the cell.

To edit the content of cells, double click on the cell to enter **edit mode**. When in edit mode, the inner box surrounding the cell will become highlighted and you are now able to edit the contents of the cell. (Note: Unlike in the video, the bar on the left margin may not turn green in edit mode anymore.)

A text (markdown) cell in edit mode should look like this:

and a code (R) cell will look like this:

Use edit mode to type in all your text and code. When you are done with your paragraph or want to try out your code, it's time to run the cell.

A Cell in edit mode

Below this line is a (mostly) blank text cell. Try selecting this cell by right clicking on it, then double click to enter edit mode. Replace the text with something.

A (mostly) blank text cell

Note: sometimes on Problem Sets or in Section you will encounter cells that are protected and can't be edited (like this one). Don't worry, we've done this to make sure that question prompts don't get accidentally deleted. Below these you'll find text/code cells that you can edit.

1.3.3 Running Cells

When you are ready to run a cell (i.e. done typing your text in a cell and want to display it in formatted mode, or want to run **R** code), hit `shift` + `enter`, `control` + `enter`, or hit `Run` in the toolbar.

Running a text cell will exit edit mode, format the cell's text, and select the next cell down.

You can also run a cell from edit or command mode - try this by right clicking on the above cell that you were typing in, and hitting `shift` + `enter`.

We'll see later that running code cells will oftentimes add output to our notebook. This will be how we follow what we're doing in **R** and how we'll get our output.

You can reopen a cell you've run just as before by double clicking on it, letting you modify your code/text and rerun it.

1.4 R

R is our programming language for statistical programming. It is open source and free, handles lots of different types of data, and has tons of cool packages that allow it to tackle all sorts of tasks (regression analysis, plotting, working with spatial data, web scraping, creating applications, machine learning to name a few). It uses the exact same language and syntax regardless of whether we're in a Jupyter Notebook, in RStudio through Datahub, or working with Rstudio offline on our own computers.

1.4.1 R in Code Cells

Thanks to Datahub, R is running in the background of our Jupyter notebooks. As a result, we can type R code into code cells, run the cell, and get results all in one place. Code cells are easy to tell apart from text cells as they have `In [ ]:` to the left, and when selected you'll see the toolbar switch to *Code*. For now we'll tend to add a line or two of code to a cell at a time as that lets us break things up and makes it easier to debug errors, but you can theoretically add as many lines of code to a single cell as you'd like.

Let's try it: select the code cell below, type `2 + 2`, and run the cell.

```
[ ]:
```

R took our code as input, and spit out the result - in this case the value of our summation, 4.

Important Note: before you save a homework pdf to submit, make sure you've run all your code cells! You won't get any output if you don't run a code cell, and for full credit we need to see the output (unless we specify otherwise).

An easy way to run all the cells at once is by going to the menubar and using **Cell > Run All**.

1.4.2 R Basics

Note of caution: **R** is a stickler for typos. Precise syntax is essential. Commas, parentheses, brackets, and other punctuation must be in the correct place. R is also case-sensitive, so upper and lower case matter! Any deviation from required syntax will lead your code to either fail or produce unintended (and incorrect) results. You will spare yourself a lot of aggravation if you take the time to go slowly and carefully as you're getting started until you have gotten more familiar with the commands and their required syntax.

1.4.3 Packages, Libraries, and Paths

One of R's best features is that it's based on packages. Base R already loads some general packages when you boot it up, but loading other packages makes R incredibly powerful. Nearly any time you do any econometrics in R, you'll be using functions contained within a package.

Every time we want to use a function contained in a package, we must first *call* that package using the `library()` function. For example, if we want to use the `read_dta()` function to load a dta file, we can load the **haven** package by typing and running the command `library(haven)` in the below cell:

[ ]:

The package was loaded correctly in the background. Since there was no output to display nor an error to show, we received no output. We can confirm that the package was loaded correctly by running the `sessionInfo()` function below (note the capital I!).

[ ]:

This shows us information about our **R** version, local settings, active packages, ones we've manually loaded, and others that are loaded through our system.

If we had written the package name wrong, and instead tried to load a nonexistent package "hevon", we would receive an error in the notebook. Try to incorrectly load the package by running `library(hevon)` in the below code cell.

[ ]:

1.4.4 Paths

When we start loading in data files we will need to account for the location of files in our file paths. **R** handles paths by basing itself in the *working directory*, and all file paths are defined relative to that working directory.

We can view the working directory by using the `getwd()` function.

[ ]:

The above path is the exact folder you were linked to from bCourses. When calling files, we will start our paths in this folder. If the file is located in the same folder as our Jupyter notebook (i.e. the working directory), we do not need to include all the "/home/rstudio…" path, and instead can just reference the file by name. If the file we want to reference (let's call it *enviro.csv*) is in a subfolder (say called *Data*), we will need to include that subfolder in our path: "Data/enviro.csv"

We'll deal more with working directories once we move into **RStudio** - for now all the files we need will be in the same folder as our template, so we can call the files directly by name without worrying about complicated folder structures.

1.4.5 Loading Files in R

We're going to read in our data, but before we can read in our dataset we should refresh ourselves on what the file is called. To do this, we can use the `list.files()` command below to see all the

files in our current working directory.

[ ]:

The file we're interested in first is the **autos.dta** file. Since this is a stata-formatted file (with the .dta extension), we'll need to use the `read_dta()` function in the **haven** package (which we already loaded above). To use the function we need to include the filename with quotes in the parentheses: `read_dta("autos.dta")`. The quotes tell R that whatever is contained in the quotes is a character string, and not the name of an object saved in memory.

Here we run into an important feature about **R**: when we load a dataset or want to store an object to memory (such that we can do things with it later), we have to *assign it to a name*. To do this, we use the syntax

```
name <- function(arguments)
```

The arrow tells us we are assigning the output of the function (given the function's arguments/inputs) to *name*. The arrow is read as "gets," so if we wanted to store a vector of integers between 1 through 10 as `integers` using the command

```
integers <- 1:10
```

We read it as "integers 'gets' 1 through 10."

Okay, let's load in the **autos** data and save it to the object `carsdata` by running `carsdata <- read_dta("autos.dta")`:

[ ]:

The dataset has now been stored to memory under the name `carsdata`. R is pretty flexible in terms of what we could have named the dataset - you can use names that include periods (`cars.data`) or underscores (`cars_data`) or capital letters and numbers, or even all the above (`CaRs_D.aTa2020`). Best practice is to keep things simple and name them in a way that conveys what the object is (hence calling our automobile data `carsdata`). Remember that the GSIs are going to be reading your code when grading problem sets, and they appreciate names that are intuitive (and devoid of profanity).

To view the data, we could just type the name of the dataset into a code cell and hit run… but be aware that this will print the **entire** dataset. This isn't bad if you have a small dataset, but soon we'll be working with large datasets with thousands of rows. It's a real pain to have to scroll through thousands of rows of data (and takes up whole pages in our final pdf!), but fortunately there's an easy way to view only a few lines of the dataset. We can use the `head()` command to see only a few observations. Try using `head(carsdata)` below to view the top few rows of the data we just loaded.

Note: by default, head returns the first 6 rows of data, but you can get more/less rows by adding a comma and including the optional argument `n =` *with the number of rows you want to view - i.e.* `head(carsdata, n = 12)`*. You can also use* `tail()` *in the same way to view the last few rows of the data.*

[ ]:

The very first bit of information tells us the format of the data (data frame). The first row tells us the variable names, and the second row tells us the variable type (character string or type of numeric variable). Each row in the table is a different observation - here giving us info on a car make and model, its price, mpg, and other characteristics.

If we want to see just the names of the variables, we can use the aptly-named `names()` function on our data:

[ ]:
```
```

If we want to interact with just one column of our data frame, we can refer to it using the `$` command. This is useful if we're interested in, say, obtaining the mean of only miles per gallon (*mpg*) for the cars in our sample with the `mean()` function.

We can also combine this with `head()` to view just the first six rows of the column *mpg*.

Try running the below code cell to see these in action.

[ ]:
```
head(carsdata$mpg)
mean_mpg <- mean(carsdata$mpg)
```

Note that we don't see any output from our mean calculation because we saved it to memory as the object `mean_mpg`. In order to view its value, we can print the object just by its name. Type `mean_mpg` and run the code cell below.

[ ]:
```
```

Other File Formats In this class we'll primarily use `.dta` and `.csv` files. When we load a .csv file, we'll use the `read.csv()` command, once again using quotation marks around the file name in the function and assigning it to a name.

For example, we could load the csv version of the automobile data that's in our folder with the name *carsdata* using `carsdata2 <- read.csv("autos.csv")`. Note the use of a period in the function name versus the underscore when loading dta files!

[ ]:
```
```

1.4.6 Manipulating Data Frames

We have a lot of flexibility to select certain observations, certain variables, or certain values within our data frame. We can also perform a lot of operations to variables - change their values or create new variables. One of the packages we'll be using for this is the **tidyverse** package. It's actually a collection of packages designed for data science, and includes a bunch of useful functions that we'll use throughout this term.

To start, let's load **tidyverse** and use it to perform transformations on our dataset.

[ ]:
```
library(tidyverse)

carsdata_low_price <- filter(carsdata, price <= 10000)
head(carsdata_low_price)
```

```
carsdata_low_price <- arrange(carsdata_low_price, desc(price))
head(carsdata_low_price)

carsdata_low_price <- select(carsdata_low_price, make, price, mpg, weight)
head(carsdata_low_price)
```

Phew, that was a lot of stuff! Let's back up and work through it.

The first thing we did after loading the **tidyverse** package was to `filter()` our data frame. This function has two main arguments. The first is the name of the object to filter (we're using our carsdata), and the second is the condition on which to filter. Here we are selecting only the observations with price less than or equal to $10,000, and dropping all rows that do not meet this condition in the price variable. We save this filtered data to the new `carsdata_low_price` object. Note that you could overwrite the existing `carsdata` with the filtered version by just assigning it to that name once again.

Next, we arrange the new `carsdata_low_price` data frame in descending order of price using the `arrange()` function and overwrite our `carsdata_low_price` with this new arrangement.

You can arrange in ascending order by omitting the `desc()` around the variable of interest (i.e. `arrange(carsdata_low_price, price)`) or arrange on several variables by listing them in order (i.e. `arrange(carsdata_low_price, price, weight)`).

Finally, we use `select()` to keep only a few variables of interest - here we choose to keep just vehicle make, price, miles per gallon, and weight - and once again overwrite our `carsdata_low_price` object.

Now we'll create new versions of our variables using the `mutate()` command. `mutate()` is a workhorse function that lets us alter existing variables and add new variables to our dataframe. Later on we'll learn a technique that will simplify the syntax for mutate and make it even easier to do multiple transformations all in a row.

If I wanted to rescale price to be in units of $1,000, we can do this by typing

[ ]:
```
carsdata_low_price <- mutate(carsdata_low_price, price_thousand = price/1000)
head(carsdata_low_price)
```

Here we are adding the variable `price_thousand` to the dataframe `carsdata_low_price` that measures price in thousands of dollars. R uses most basic math symbols in the ways you would expect: `+`, `-`, `*`,\, and `^` (be careful with order of operations and use parantheses when necessary).

We can also create new variables that are interactions of existing variables. For example, if we wanted to know the ratio of price (in dollars) to weight (in pounds), we could obtain this with `mutate(carsdata_low_price, price_per_lb = price/weight)`.

1.4.7 Summarizing Variables

There are a number of different ways for us to obtain summary information about our variables in **R**. While we saw earlier one way to directly get the mean of a variable, we could instead have used the `summarise` command, which lets us obtain a number of different statistics - either one at a time, or multiple together

[ ]:
```
avg_mpg <- summarise(carsdata_low_price, mean(mpg))
avg_mpg

multi_stats <- summarise(carsdata_low_price, max(mpg), min(price), n())
multi_stats
```

`avg_mpg` gives us the average miles per gallon rating for the `carsdata_low_price` data, while `multi_stats` contains the maximum mpg, the minimum price, and the number of rows in the dataframe - all in its own dataframe.

`summarise()` is useful because we can use it to select specific summary statistics we're interested in or to create truly custom summary stats.

If instead we just wanted a bunch of basic stats, we can use the `summary()` command. You can use this on the dataframe to get info on all the variables, or on a specific variable (recall the use of `$` we talked about earlier).

[ ]:
```
summary(carsdata_low_price)
```

[ ]:
```
summary(carsdata_low_price$price)
```

What if we want to get custom summary statistics across a set of variables? In that case we can use the `across()` function within `summarise()` to save a few lines of code.

In place of our usual summary statistics equations, we now add in `across()` which has two arguments. The first, `.cols`, is the set of columns we want to summarise across (in this case, *price*, *weight*, and *gear_ratio*). The second, `.fns`, is the list of functions we want to use for summary statistics. We specify these functions within a `list` object, where we first provide a name of a summary stat (i.e. "avg") and set that equal to the name of the function we want to use. In this case, we want to create a summary stat called "avg" that uses `mean()`, so we can write `avg = mean`.

Similarly, `SD = sd` will add columns to our summary stats table for each of our three named variables where we use the `sd()` function to calculate their means.

[ ]:
```
summarise(carsdata,
          across(.cols = c(price, weight, gear_ratio),
                 .fns = list(avg = mean, SD = sd))
          )
```

And we can see that **R** 1) organizes the summary stats first by the variable and then by the summary function, and 2) appends the summary stat name from our list to the end of the variable's name.

This gives us a convenient way to produce a larger set of summary statistics for a lot of variables at once, without having to manually enter a separate line over and over again for each new variable!

One challenge is that, if we summarise across a lot of variables, our notebook might hide some of the columns from view. To get around this, you can either split up the table into multiple tables with fewer variables each, or put the summarise call inside of `t()` to transpose it and flip it into a single column.

1.4.8 Plotting in R

Today we're going to learn how to use the basic plot function in R. In the coming weeks we're going to learn **ggplot2**, a fantastic graphics package that is so great at producing graphics that it basically justifies the use of R all on its own. It's syntax is a little bit more involved though, so we're going to start off with some simpler functions.

First, we can generate a histogram of vehicle prices using the `hist()` command.

[ ]:
```
hist(carsdata_low_price$mpg,
    main = "Fuel Economy Distribution",
    xlab = "Miles per Gallon")
```

The first argument to `hist()` is the variable to plot, `main` is the main title, and `xlab` is the text label for the x-axis.

We can also add a blue line of width three at the average mpg by running the `abline()` function right after our histogram:

[ ]:
```
hist(carsdata_low_price$mpg,
    main = "Fuel Economy Distribution",
    xlab = "Miles per Gallon")
abline(v = avg_mpg, col = "blue", lwd = 3)
```

We can also make a scatterplot of vehicle price and fuel efficiency using the `plot()` function.

[ ]:
```
plot(carsdata_low_price$mpg, carsdata_low_price$price,
    main = "Price vs. Fuel Efficiency",
    xlab = "Miles per Gallon",
    ylab = "Price ($)")
```

Since a scatterplot requires two variables, the first argument to `plot()` is the x variable and the second argument the y variable. `ylab` is the text label for the y-axis.

There seems to be a negative correlation here: the higher MPG, the lower the price. Is this potentially a luxury SUV effect? Low willingness to pay for fuel efficiency?

1.4.9 Regression

Running regressions with **R** is quite easy. Later in the course we'll get into some more complex regression commands, but for now we'll stick with simple linear regression using the `lm()` command.

First, let's take a step back and see how mpg and price are correlated in the data. We can obtain correlations of two variables using the `cor()` function.

[ ]:
```
mpg_weight_cor <- cor(carsdata_low_price$mpg, carsdata_low_price$price)
mpg_weight_cor
```

What if we used a simple OLS regression instead? With regression functions like `lm()`, the first argument is the regression formula. Here we specify `mpg ~ weight`, so we're telling R to describe *mpg* as a function of *weight*. In practice, this is estimating the equation

$$mpg_i = \beta_0 + \beta_1 weight_i + \epsilon_i$$

where *mpg* is the dependent variable and *weight* the independent variable/covariate. β_0 and β_1 are our intercept and slope coefficients, respectively - we'll spend a bunch of time learning about and interpreting these coefficients throughout the class. ϵ_i is an error term.

The second argument is the name of the dataset.

[ ]:
```
mpg_weight_regression <- lm(mpg ~ weight, data = carsdata_low_price)
mpg_weight_regression
```

`mpg_weight_regression` is now an lm class object, which contains our estimated regression coefficients as well as a bunch of other useful stuff, most notably residuals and fitted values. If we use the `summary()` function on an lm object, we get back a lot of information:

[ ]:
```
summary(mpg_weight_regression)
```

We can also obtain the fitted values and residuals by calling them with `name$fitted.values` and `name$residuals`, respectively. Use the below code to look at the first few fitted values and the average value of the regression residuals.

[ ]:
```
head(mpg_weight_regression$fitted.values)
mean(mpg_weight_regression$residuals)
```

1.5 Help with R

Google is your best friend. While I will introduce you to some R commands during section, there may be times where you are having trouble with a function's syntax or need a new function to perform a certain task. Asking Google, using the R Documentation site as well as R Project Package Reference Manuals/Vignettes for help with functions and syntax will get you quite far. You can also load the documentation for a function by typing `?function()` into a code cell. For example, try typing `?select()` into the below code cell:

[ ]:
```

```

If something is unclear, searching for the R task and perusing answers on StackExchange can be a helpful resource. We will try our best to make sure homework assignments mention the functions/packages needed for a new task, and that we've at least seen them in section or lecture.

1.6 Closing Notebooks and Shutting Down your Server

When you are all done with your notebook, make sure to shut it down. Shutting down notebooks will prevent save conflicts and reduce the likelihood of any server issues.

After saving, shut down your active notebook by going to **File > Close and Halt**. This will close the notebook tab.

To shut down your entire server, from the directory click on **Control Panel** in the upper right corner

Then, select **Stop My Server** followed by **Logout**.

[ ]: