R Lab 0 - R Programming Refresher

Advanced Topics in Causal Inference

Goals for this lab:
1. Review of R programming and good programming practice.
2. Create a basic function to generate data with $O = (W, A, Y)$ structure.
3. Implement g-computation, IPTW, and TMLE estimators inspired by the average treatment effect (ATE) of a point treatment intervention.

Next lab:
1. Explore different data structures, inspired by real-world experiments.
2. Simulate data generating process that gives rise to the data we observe.
3. Think of studies in which these data generating systems may occur.

1 Introduction and Motivation

Welcome to Advanced Topics in Causal Inference R Labs! This semester we'll be working in R to implement concepts we've learned in lectures and discussion sections. In this lab we'll provide a refresher of R programming. Here is a link to cheatsheets I find useful that you may enjoy studying on your own: https://www.rstudio.com/resources/cheatsheets/.

Section 2 will give you a refresher on R basics, including objects, subsetting, making plots, etc. Section 3 will walk you through how to generate data, and Section 4 will implement the 3 estimators we learned in Causal I to estimate the average treatment effect. You won't be turning in this lab, so feel free to complete as many or few sections as you feel will help you get a solid foundation before we start labs (to turn in) next week.

In future labs, please turn in relevant R code, text, and figures. Consider turning in your homework as a compiled R Markdown (RMD) file, which "knits" your code, text, and figures into a single PDF, Word, or html document. A .Rmd template with further instructions can be found on bCourses.

Figure 1: Welcome to Causal II labs!

2 R review and refresher

2.1 Help files

Help files in R are extremely useful. Coding in R isn't so much about memorizing functions; it's more useful to know how to look up commands. Knowing how to read the help documentation will make you a much more independent and creative programmer.

For example, type `?rnorm` in your console in the command line. This pulls up the help page for the `rnorm()` function. What does the function `rnorm()` do? The help file gives you a description of all things having to do with the normal distribution (you will see a few other functions listed, such as `pnorm()`, `dnorm()`, and `qnorm()`).

The *Usage* section lets you know what arguments go into which functions. If there's an equal sign (i.e., =), that means that argument has a default value, so you don't need to specify that argument. You only need to specify if you want something different than the default.

The *Arguments* section describes what each of the arguments are.

So, for example, `rnorm(n = 3, mean = 4, sd = 1)` generates 3 random numbers from a normal distribution with mean 4 and standard deviation 1. And `rnorm(n = 3)` generates 3 random numbers from a normal distribution with mean 0 and standard deviation 1. Notice we don't need to specify the `mean` and `sd` arguments in `rnorm()` if we want to generate numbers from a $Normal(\mu = 0, \sigma = 1)$ distribution because these are the default parameters (i.e., in the Usage section of the help file, the arguments `mean` and `sd` are equal to 0 and 1, respectively). Conversely, we always need to specify what `n` is because there's no default (i.e., no '=' sign after `n`).

2.2 Assigning values to R objects

We can assign results of computations to variables. Type each of the following lines either in an R script (and run each line), or in your console:

```
> x = (6-2)^.5/3^2
```

Here we are assigning a complicated arithmetic computation to the variable `x`. Notice what comes up in the workspace window when you assign the variable `x` to that value. Notice what comes up in the console when you run `x`:

```
> x
```

```
[1] 0.2222222
```

`identical()` is a function that outputs a logical vector telling us whether two variables are identical:

```
> X = 3
> identical(x, X)
```

```
[1] FALSE
```

As you can see here: R is case sensitive! Here we're catenating `x` and `X` into a vector called `y`:

```
> y = c(x, X)
> y
```

```
[1] 0.2222222 3.0000000
```

On your own (remember, use `?` to figure out what a function does and what arguments that go into a function):

1. Use the `rep()` function to make a vector that repeats the number "4" 5 times (i.e., 4 4 4 4 4). Assign that vector to the variable `z`.
2. Use the `seq()` function to make a vector that shows odd numbers from 1 to 10 (i.e., 1 3 5 7 9). Assign that vector to the variable `y`. Notice how we've overwritten the variable `y` we had before.
3. Use the `sample()` function to sample 17 of elements from `y` with replacement. Assign that vector to the variable `w`.
4. Add `z + y`.
5. Add `y + w`. What happened? Why? (Hint: use the `length()` function on `y` and `w`).
6. Remove `w` using the `rm()` function.

Solution:

```
> #1. use the rep() function to make vector z
> z = rep(x = 4, times = 5)

> #2. use seq() function to make vector y
> y = seq(from = 1, to = 10, by = 2)

> #3. use sample() function to make vector W
> w = sample(x = y, size = 17, replace = TRUE)

> #4. add z + y
> z + y

[1]  5  7  9 11 13

> #5. add y + w
> y + w

 [1]  6  4 12  8 14  4  8  8 12 18  2 10  6 16 16  6  4

> # why did we get a warning?
> length(y)

[1] 5

> length(w)

[1] 17
```

The vectors `y` and `w` do not have the same length (in contrast to question 4). In cases like these, `R` recycles the `y` vector for the remaining elements of the vector `w`.

If `w` had been a multiple of `y` (for example, if `w` was 20), we wouldn't have gotten a warning message - beware of recycling!

```
> #6. remove the variable w
> rm(w)
```

```
> #7. other example uses
> n_items <- 4
> for (i in seq(n_items)) {
+   print(paste(i, "bat, mwahaha"))
+ }
```

```
[1] "1 bat, mwahaha"
[1] "2 bat, mwahaha"
[1] "3 bat, mwahaha"
[1] "4 bat, mwahaha"
```

```
> die = 1:6
> manyRolls = sample(die, 100, replace=T)
> sixFreq = sum(manyRolls == 6)
> sixFreq / 100
```

```
[1] 0.15
```

```
>
```

2.3 Data types and subsetting

Subsetting just means picking out elements of a vector (or other objects). This will be useful for picking out certain variables or observations of a dataset. Let's reset `z` and `y` to the following (i.e., run these two lines):

```
> z = c(2, 4, 6, 7, 10)
> y = c(1, 3, 5, 7, 9)
```

Catenate `y` and `z`, and assign to `x`.

```
> x = c(y, z)
```

2.3.1 Subsetting

- Subsetting by *position*

  ```
  > x[3]
  ```

  ```
  [1] 5
  ```

Here, we are getting the 3^{rd} element of `x`. This is subsetting by *position*.

- Subsetting by *exclusion*

```
> x[-3]

[1]  1  3  7  9  2  4  6  7 10
```

Here, we are getting all but the 3^{rd} element of `x`. This is subsetting by *exclusion*.

- Subsetting by *name*

Write the following code. There's a "built-in" vector called `letters` in `R` that has all the letters in the alphabet:

```
> names(x) = letters[1:length(x)]
> x

 a  b  c  d  e  f  g  h  i  j
 1  3  5  7  9  2  4  6  7 10
```

Above, we're taking the first 10 (since that's the length of `x`) letters of the alphabet, and assigning those letters as names for the element in `x`. Below, we're getting all elements of `x` with the name `'c'`:

```
> x['c']

c
5
```

This is subsetting by *name*.

- Subsetting by *logical*

The following returns a vector of "logicals" telling us which elements of `y` are equal to 3:

```
> y==3

[1] FALSE  TRUE FALSE FALSE FALSE
```

The second element returned is `TRUE` because the second element of `y` is 3. Notice our use of the double equals sign (`==`). You can also do `>` for greater than, `<` for less than, `>=` for greater than or equal to, `<=` for less than or equal to. Below, we're subsetting the elements of `z` based on the elements of `y` are equal to 3:

```
> z[y==3]

[1] 4
```

This is subsetting by *logical*.

- Subsetting by *everything*:

```
> x[]

 a  b  c  d  e  f  g  h  i  j
 1  3  5  7  9  2  4  6  7 10
```

Here, we are getting everything from `x`.

On your own:

1. Get every other element of `x` (hint: use one of the functions you implemented above!)

2. Get elements with the name "a", "c", and "f"
3. Make every element of `x` equal to 4.

Solution:

```
> #1. every other element of x
> x[seq(from = 1, to = 10, by = 2)]


a c e g i
1 5 9 4 7


> #2. elements with a, c, f
> x[c("a", "c", "f")]


a c f
1 5 2


> #3. set all elements of x to 4
> x[] = 4
> x


a b c d e f g h i j
4 4 4 4 4 4 4 4 4 4
```

Incorporating what we have learned so far, you may be in a situation where you want to randomly sample observations from your data in order to evaluate the fit of your model, you could do something like:

```
> train_ind <- sample(seq_len(nrow(my_data)), size = 0.75 * nrow(my_data))
> train <- my_data[train_ind, ]
> test <- my_data[-train_ind, ]
> train_X <- subset(train, select = -y)
> train_Y <-train$y
> test_X <- subset(test, select = -y)
> test_Y <-test$y
```

2.4 Matrices and dataframes

2.4.1 Matrices

Matrices are "shaped" vectors, in which all items have to be of the same type. Let's reset `x` to be a vector with numbers from 1 to 10:

```
> x = 1:10
```

Now let's construct a matrix, `mat`, which uses the elements of `x`, and makes a "shape" out of them so that it has 2 columns and 5 rows:

```
> mat = matrix(x, nrow = 5)
> mat
```

```
     [,1] [,2]
[1,]    1    6
[2,]    2    7
[3,]    3    8
[4,]    4    9
[5,]    5   10
```

What are the dimensions of the matrix, `mat`?

```
> dim(mat)
```

```
[1] 5 2
```

Let's subset out the second row of `mat`:

```
> mat[2,]
```

```
[1] 2 7
```

Now let's subset out the first column of `mat`. Notice how it returns a vector.

```
> mat[,1]
```

```
[1] 1 2 3 4 5
```

On your own:

1. Get the element that's the third row and first column of the `mat` matrix.
2. Assign names to the `mat` matrix so that it looks like this:

   b c

   a 1 4

   b 3 4

   c 5 4

   d 7 4

   e 9 4

   Hint: Use `rownames(mat)` and `colnames(mat)`.

Solution:

```
> #1. 3rd row, 1st column
> mat[3, 1]
```

```
[1] 3


> #2. add row names and column names to mat
> rownames(mat) = letters[1:5]
> colnames(mat) = letters[2:3]
```

2.4.2 Dataframes

A data frame is a collection of columns that can be of different types. This is what you typically think of as a dataset. In your group projects, you may need to load a data file read/manipulate it in `R`. If you haven't done so already, download the CAquakes.csv file from the Lab 0 folder in bCourses. Notice which folder on your computer you've saved it to, or move it to a folder that makes sense for this assignment. First, set your working directory:

```
> setwd("/Users/mccoyd/Box/252E_2021/RLabs/RLab0")
```

This is where CAquakes.csv lives on my computer. Change the filepath according to where CAquakes.csv lives on your computer and make sure to put quotes around the filepath. Read in the CAquakes .csv file:

```
> CAquakes = read.csv(file = "CAquakes.csv")
```

Note, the above will only work if the working directory is set to where CAquakes.csv lives on your computer. Also notice how the dataframe CAquakes has appeared in your workspace. Let's get the data type of CAquakes:

```
> class(CAquakes)
```

```
[1] "data.frame"
```

We see `R` read it in as a data frame. Type `View(CAquakes)` in your console. Inspect your data:

```
> dim(CAquakes) # dimensions of your data frame
```

```
[1] 383   4
```

```
> head(CAquakes)  # this gets the first 6 elements of CAquakes
```

```
      Date Latitude Longitude   M
1 18001011     36.8    -121.5 5.5
2 18001122     32.9    -117.8 6.3
3 18030000     34.2    -118.1 5.5
4 18030525     32.8    -117.1 5.5
5 18060325     34.4    -119.7 5.5
6 18080621     37.8    -122.6 5.5
```

```
> summary(CAquakes)
```

```
      Date             Latitude       Longitude            M
 Min.   :18001011   Min.   :31.50   Min.   :-125.9   Min.   :5.500
 1st Qu.:18890676   1st Qu.:34.15   1st Qu.:-121.8   1st Qu.:5.600
 Median :19261024   Median :36.50   Median :-119.6   Median :5.900
 Mean   :19235315   Mean   :36.37   Mean   :-119.7   Mean   :5.992
 3rd Qu.:19585801   3rd Qu.:38.30   3rd Qu.:-117.2   3rd Qu.:6.200
 Max.   :19991016   Max.   :42.31   Max.   :-114.5   Max.   :7.900
```

The `summary()` function gives you information on your object, depending on what kind of object it is. What kind of information does the `summary()` function give for a dataframe?

Here are 3 ways of extracting the first 6 elements of Latitude column vector:

```
> head(CAquakes$Latitude)

[1] 36.8 32.9 34.2 32.8 34.4 37.8

> head(CAquakes[["Latitude"]])

[1] 36.8 32.9 34.2 32.8 34.4 37.8

> head(CAquakes[,2])

[1] 36.8 32.9 34.2 32.8 34.4 37.8
```

On your own:

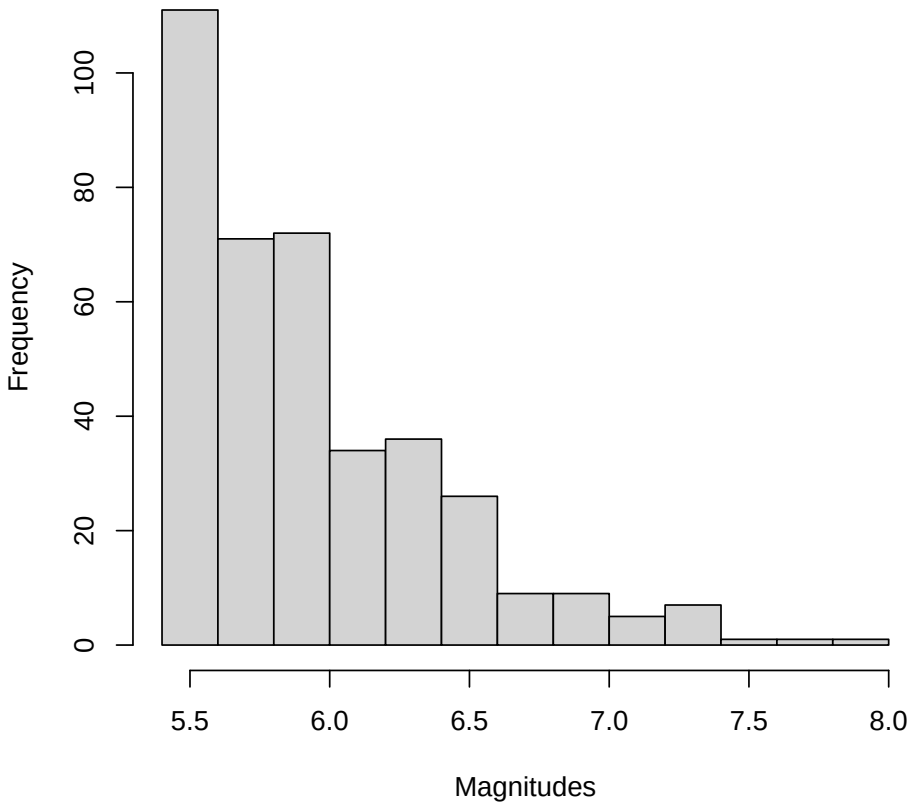
1. Using the `CAquakes` data, get the mean Longitude.
2. Look up the function `hist()` and plot the magnitudes (M) of the earthquakes. Try changing the main title and x-axis title to something more informative than the default.
3. Use a scatterplot to visualize how earthquake magnitude behaves as latitude increases (*Hint*: look up and use the `plot()` function). Again, try changing the titles.

Solution:

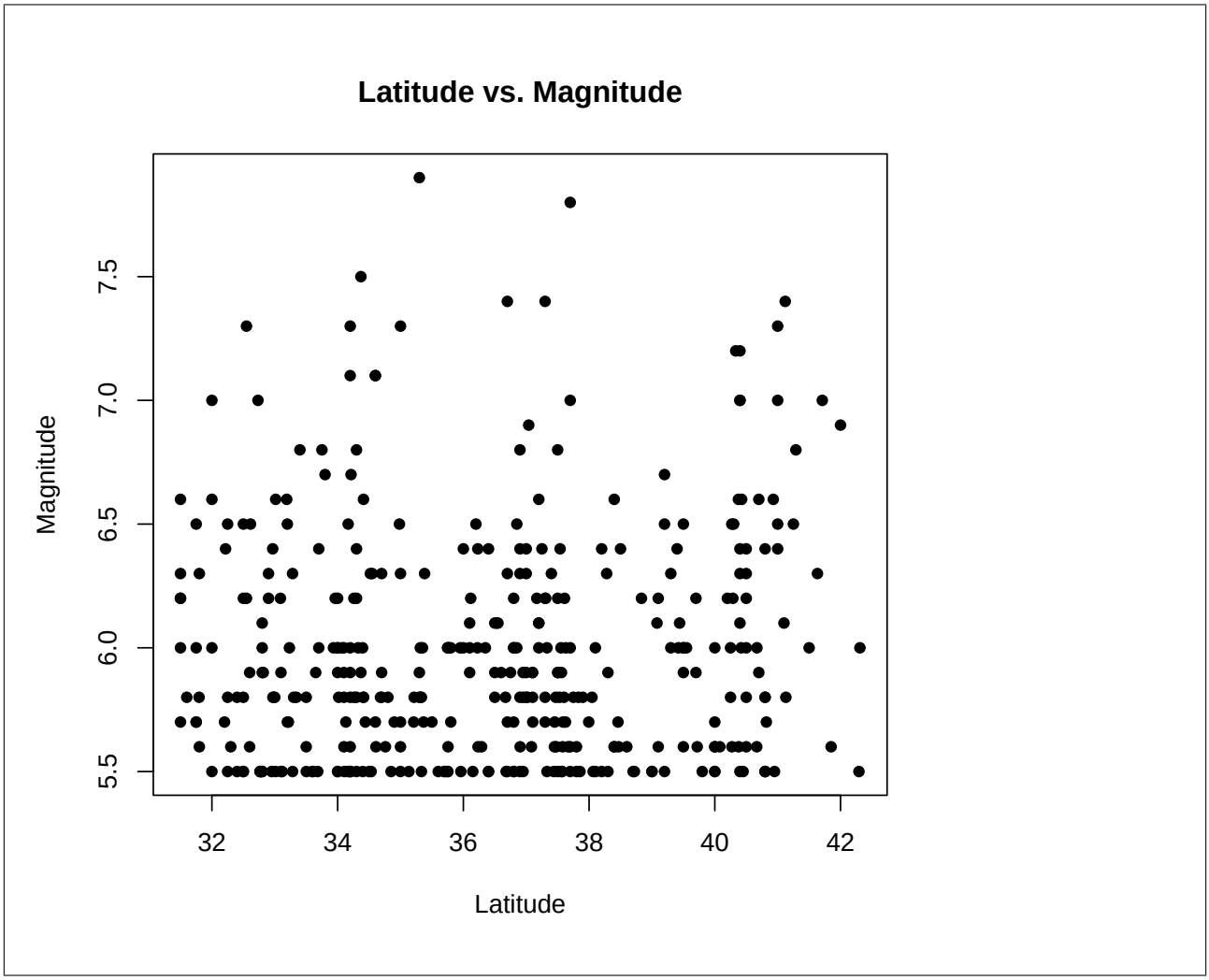
```
> #1. avg longitude
> mean(CAquakes$Longitude)

[1] -119.7381

> #2. make hist of magnitudes
> hist(x = CAquakes[["M"]], main = "Magnitudes of CA earthquakes", xlab = "Magnitudes")
```




```
> #3. make plot of latitutde vs magnitude
> plot(x = CAquakes$Latitude, y = CAquakes$M,
+      xlab = "Latitude", ylab = "Magnitude",
+      main = "Latitude vs. Magnitude", pch = 16)
```



3 Creating functions and generating data

There are many built-in functions in `R` (for example, above we saw the `rep()`, `seq()`, and `rnorm()` functions, to name a few). However, it's also possible to write your own functions. We will often create functions in these labs, in particular, to generate many copies of variables that comprise a dataset. The general syntax for a function is as follows:

```
> examplefun = function(x) {
+
+   y = x + 3 # what the function does
+
+   return(y) # what we want the function to return back (output)
+ }
```

`examplefun` is the name of the function (we make that up), and `x` is the argument (input). `examplefun()` takes in as input `x`, and outputs `y`, which is the input plus 3. For example, if input 4 for `x`, we anticipate `examplefun()` to spit out 7 (i.e., 4 + 3):

```
> examplefun(x = 4)
```

```
[1] 7
```

On your own:

1. Set the seed to 252.
   Note: Setting the seed to a value (generally) ensures that running the same chunk of code twice will return the exact same output, even if the code involves "random" numbers. This will also help your GSIs help you with your labs/homework assignments/projects.

2. Create a function, `Uniform()`, that generates n i.i.d. random variables uniformly distributed between 0 and x.
   Hint: the function should take in 2 arguments as input: 1) n, the number of uniform variables to generate, and 2) x, the maximum value the uniform variable can take. *Extra hint:* look up the `runif()` function to randomly generate values from the uniform distribution.

3. Create a function, `generate_data()`, that generates a dataframe with n i.i.d. random variables that follow these distributions:

$$\begin{aligned} U_{W1} &\sim Uniform(0,1) \\ U_{W2} &\sim Normal(\mu = 1, \sigma^2 = 2^2) \\ U_A &\sim Uniform(0,1) \\ U_Y &\sim Uniform(0,1) \end{aligned}$$

   Hint: the skeleton of your function should look like this (fill in the commented out parts with your own code):

```
> generate_data = function(n) {
+
+   # generate n Uniform(min = 0, max = 1) variables here, set equal to U.W1
+   # generate n Normal(mean = 0, max = 1) variables here, set equal to U.W2
+   # generate n Uniform(min = 0, max = 1) variables here, set equal to U.A
+   # generate n Uniform(min = 0, max = 1) variables here, set equal to U.Y
+
+ }
```

   where `generate_data` takes in the single argument `n`, the number of observations. Try using the `Uniform()` function you created in the previous step to generate the uniform random variables.

4. Add to the function `generate_data()` so that it also creates 4 new variables based on the following functions that relate them:

$$\begin{aligned} W1 &= \mathbb{I}\big[U_{W1} < 0.45\big] \\ W2 &= 0.75^* U_{W2} \\ A &= \mathbb{I}\big[U_A < expit(-1 + 2.6^* W1 + 0.9^* W2)\big] \\ Y &= \mathbb{I}\big[U_Y < expit(-2 + A + 0.7^* W1)\big] \end{aligned}$$

   Hint: Use the `as.numeric()` function for indicator variables.
   Example: $\mathbb{I}[X > 0]$ can be coded as `as.numeric(X>0)`.

Extra hint: Recall that the *expit* function is the inverse of the logistic function:

$$logit(x) = log\left(\frac{x}{1-x}\right)$$
$$expit(x) = \frac{1}{1+e^{-x}}$$

In `R`, the *expit* function is called `plogis()`.

5. Within `generate_data()`, create a dataframe using either the `data.frame()` function or the `cbind()` function consisting of only $W1, W2, A$, and Y and let that be your output (i.e., what you `return`) for the function `generate_data()`.

6. **Create 1000 copies of** $W1, W2, A,$ **and** Y using `generate_data()` and set the output of the function equal to the object `ObsData`.

7. Show the first 6 lines and summary statistics of `ObsData`, using the `head()` and `summary()` functions, respectively.

Solution:

```
> #1. set the seed
> set.seed(252)


> #2. function that creates n i.i.d. uniform random variables
> Uniform = function(n, x) {
+
+  return(runif(n = n, min = 0, max = x))
+
+ }


> # 3 - 5
> # function that creates n i.i.d. copies of O
> generate_data = function(n) {
+
+   # exogenous variables
+   U.W1 = Uniform(n, x = 1)
+   U.W2 = rnorm(n, mean=1, sd=2)
+   U.A = Uniform(n, x = 1)
+   U.Y = Uniform(n, x = 1)
+
+   # endogenous variables
+   W1 = as.numeric( U.W1 < 0.45)
+   W2 = 0.75*U.W2
+   A = as.numeric( U.A < plogis(-1+2.6*W1+0.9*W2))
+   Y = as.numeric( U.Y < plogis(-2+A+0.7*W1))
+
+   # dataframe with endogenous variables
+   O = data.frame(W1, W2, A, Y)
+
+   return(O)
+ }
```

```
> # 6. run generate_data and set equal to ObsData dataframe
> ObsData = generate_data(1000)


> # 7. inspect ObsData
> head(ObsData)


  W1         W2 A Y
1  0 -2.2677937 0 0
2  0  1.8953489 1 1
3  1  1.9008567 1 0
4  0  1.3315402 1 0
5  0  1.1802323 0 0
6  1 -0.4424049 1 0


> summary(ObsData)


       W1              W2                A               Y
 Min.   :0.000   Min.   :-4.0304   Min.   :0.000   Min.   :0.000
 1st Qu.:0.000   1st Qu.:-0.2148   1st Qu.:0.000   1st Qu.:0.000
 Median :0.000   Median : 0.7559   Median :1.000   Median :0.000
 Mean   :0.426   Mean   : 0.7803   Mean   :0.614   Mean   :0.268
 3rd Qu.:1.000   3rd Qu.: 1.7987   3rd Qu.:1.000   3rd Qu.:1.000
 Max.   :1.000   Max.   : 5.5581   Max.   :1.000   Max.   :1.000
```

4 Implementation of g-computation (simple substitution estimator), IPTW, and TMLE estimation of average treatment effect of a point treatment

Using `ObsData` we created in the previous section, we will implement g-computation, IPTW, and TMLE estimators using SuperLearner to generate estimates inspired by the average treatment effect (e.g., $\mathbb{E}[Y_{a=1} - Y_{a=0}]$). Recall from Causal I labs (note: not to worry if you don't remember much about these estimators – we will review them [including how to implement them] later on in this course):

1. **Simple substitution estimator based on the G-Computation formula:**

$$\hat{\Psi}_{SS}(\hat{\mathbb{P}}) = \frac{1}{n}\sum_{i=1}^{n}\left(\hat{\bar{Q}}(1, W_i) - \hat{\bar{Q}}(0, W_i)\right)$$

   where $\hat{\mathbb{P}}$ is the empirical distribution and $\hat{\bar{Q}}(A, W)$ is the estimate of the conditional mean outcome given the exposure and baseline covariates $\bar{Q}_0(A, W) \equiv \mathbb{E}_0(Y|A, W)$.
   - Consistency of the simple (non-targeted) substitution estimator depends on consistent estimation of the conditional mean outcome $\bar{Q}_0(A, W)$.

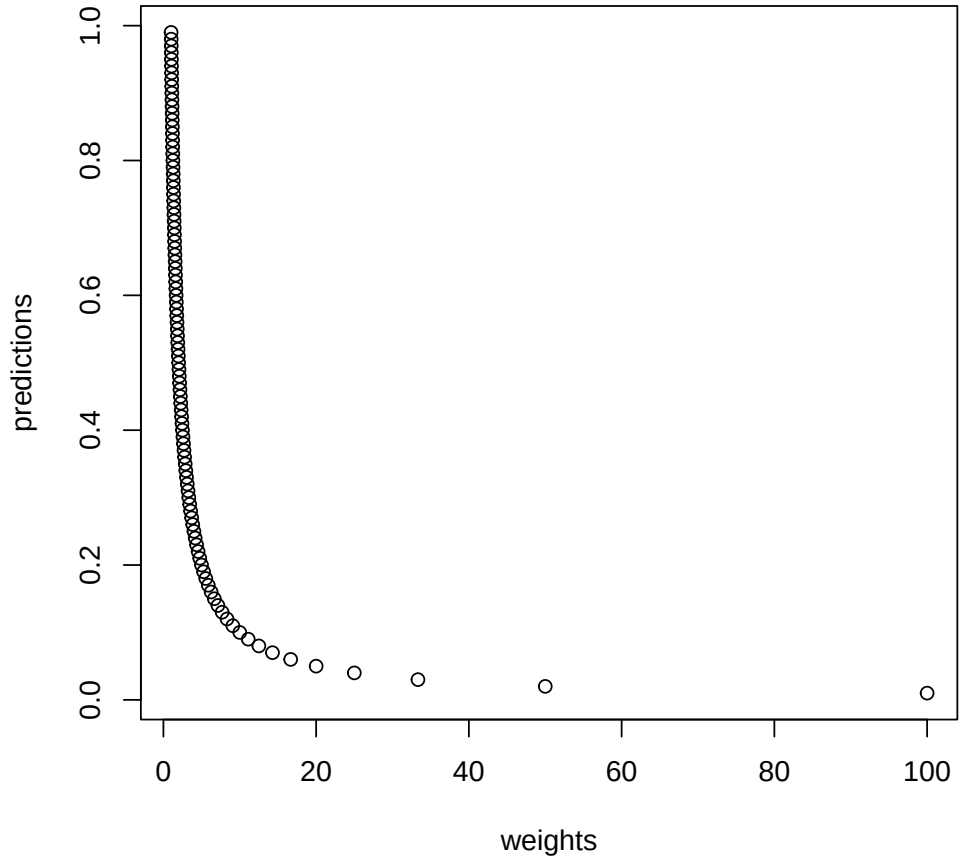
2. **Standard (unstabilized) inverse probability weighted estimator (IPTW)**:

$$\hat{\Psi}_{IPTW}(\hat{\mathbb{P}}) = \frac{1}{n}\sum_{i=1}^{n}\left[\frac{\mathbb{I}(A_i = 1)}{\hat{g}(1|W_i)} - \frac{\mathbb{I}(A_i = 0)}{\hat{g}(0|W_i)}\right] Y_i$$

   where $\hat{g}(1|W_i) = \hat{\mathbb{P}}(A_i = 1|W_i)$ is an estimate of the exposure mechanism.
   - Consistency of IPTW estimators depends on consistent estimation of the exposure mechanism $g_0(1|W) \equiv$

$\mathbb{P}_0(A = 1|W)$.

```
> predictions <- seq(0.01, 0.99, 0.01)
> weights <- 1/ predictions
> plot(weights,predictions)
```



3. **Targeted maximum likelihood estimation (TMLE):**

$$\hat{\Psi}_{TMLE}(\hat{\mathbb{P}}) = \frac{1}{n}\sum_{i=1}^{n}\left(\bar{Q}_n^*(1, W_i) - \bar{Q}_n^*(0, W_i)\right)$$

where $\bar{Q}_n^*(A, W)$ denotes the *targeted* estimate of the conditional mean outcome, given the exposure and baseline covariates $\bar{Q}_0(A, W)$.
- Implementation requires estimation of both the conditional mean function $\bar{Q}_0(A, W)$ and the exposure mechanism $g_0(A|W)$.
- Double robust estimators are consistent if either $\bar{Q}_0(A, W)$ or $g_0(A|W)$ is estimated consistently.

Reminder: An estimator is *consistent* if the point estimates converge (in probability) to the estimand as sample size $n \to \infty$.
Reminder 2: The notation of a subscript n or hat denotes that the value is an estimate (i.e., it comes from the

empirical distribution $\mathbb{P}_n$ or $\hat{\mathbb{P}}$). The notation of subscript 0 denotes that it is a true value (i.e., the value is comes from the true population distribution, $\mathbb{P}_0$). Again, we will review this later in the course :)

Warning: in practice you must go through the entire roadmap to infer any causality. Here, we are simply demonstrating how to implement the estimators.

1. Load the `SuperLearner` and `tmle` or `ltmle` packages by using the `library()` function, and set the seed to 252.
2. Set the object `n` to the number of observations in `ObsData`.
   Hint: use the `nrow()` function.
3. Create a dataframe, `X`, that contains only $W1$, $W2$, and A of `ObsData`
   Hint: use the `subset()` function.

   ```
   > X = subset(ObsData, select = c(W1, W2, A))
   ```

4. Make new dataframes `X1` and `X0` that are copies of `X`. For the dataframe `X1`, set $A = 1$; for the dataframe `X0`, set $A = 0$. For example:

   ```
   > X1$A = 1 # set A = 1 in X1 dataframe. Repeat for X0 but with A = 0.
   ```

5. Stack `X`, `X1`, and `X0` by using the `rbind()` function. Set the stacked data equal to the object `newdata`.
6. Set `SL.library = c('SL.glm', 'SL.mean', 'SL.glm.interaction')`. This is the SuperLearner library we'll use to estimate the outcome regression and treatment mechanism needed for the estimators.
7. **Call `SuperLearner()` to estimate $Q_0(A, W1, W2) = \mathbb{E}_0[Y|A, W1, W2]$, the outcome regression**. Specify the arguments `Y = ObsData$Y`, `X = X`, `newX = newdata`, `SL.library = SL.library`, and `family = 'gaussian'`. Set the SuperLearner object equal to the `Qbar`.
8. Obtain the predicted/expected Y given:
   (a) the observed A and covariates

   ```
   > QbarAW = Qbar$SL.predict[1:n]
   ```

   (b) $A = 1$ and the observed covariates

   ```
   > Qbar1W = Qbar$SL.predict[(n+1): (2*n)]
   ```

   (c) $A = 0$ and the observed covariates

   ```
   > Qbar0W = Qbar$SL.predict[(2*n+1): (3*n)]
   ```

9. **Evaluate the simple substituation estimator** by taking the mean of `Qbar1W` and subtracting it from the mean of `Qbar0W`:

$$\begin{aligned}\hat{\Psi}_{SS}(\hat{\mathbb{P}}) &= \frac{1}{n}\sum_{i=1}^{n}\left[\hat{\mathbb{E}}(Y_i|A=1, W1_i, W2_i) - \hat{\mathbb{E}}(Y_i|A=0, W1_i, W2_i)\right] \\ &= \frac{1}{n}\sum_{i=1}^{n}\left[\bar{Q}_n(1, W1_i, W2_i) - \bar{Q}_n(0, W1_i, W2_i)\right]\end{aligned}$$

10. Create a dataframe with only the covariates. Call this dataframe `W`.
11. **Call `SuperLearner()` to estimate $g_0(A|W1, W2) = P_0(A|W1, W2)$, the treatment mechanism**. Specify the arguments `Y = ObsData$A`, `X = W`, `SL.library = SL.library`, and `family = 'binomial'`. Set the SuperLearner object equal to the `gHatSL`.
12. Generate the predicted probability of:
    (a) $A = 1$ given covariates

```
> gHat1W = gHatSL$SL.predict
```

(b) $A = 0$ given covariates

```
> gHat0W = 1 - gHatSL$SL.predict
```

13. **Implement the IPTW estimator** by taking a weighted average of Y, with weights equal to $w_i = \frac{1}{g_n(A_i|W1_i,W2_i)}$:

$$\begin{aligned}\hat{\Psi}_{IPTW}(\hat{\mathbb{P}}) &= \frac{1}{n}\sum_{i=1}^{n}\frac{\mathbb{I}(A_i=1)}{g_n(A_i|W1_i,W2_i)}Y_i - \frac{1}{n}\sum_{i=1}^{n}\frac{\mathbb{I}(A_i=0)}{g_n(A_i|W1_i,W2_i)}Y_i \\ &= \frac{1}{n}\sum_{i=1}^{n}\mathbb{I}(A_i=1)^*\hat{w}_i^*Y_i - \frac{1}{n}\sum_{i=1}^{n}\mathbb{I}(A_i=0)^*\hat{w}_i^*Y_i\end{aligned}$$

14. **Implement the TMLE estimator** by calling `tmle()` or `ltmle()`.

- If you're using the `tmle()` package, specify the arguments `Y = ObsData$Y`, `A = ObsData$A`, `W = W`, `Q.SL.library = SL.library`, `g.SL.library = SL.library`.
- If you're using the `ltmle()` package, specify the arguments `data=ObsData`, `Anodes='A'`, `Ynodes='Y'`, `abar=list(1,0)`, `SL.library=SL.library`.

Solution:

```
> # 1. load SuperLearner and tmle packages
> library(SuperLearner)
> library(tmle)
> library(ltmle)


> # 2. set n equal to 1000
> n = nrow(ObsData)


> # 3. subset W, A and set to X
> X = subset(ObsData, select = c(W1, W2, A))


> # 4. create dataframes with A=1 and A=0
> X1 = X0 = X
> X1$A = 1
> X0$A = 0


> # 5. create newdata by stacking
> newdata = rbind(X,X1,X0)


> # 6. set SuperLearner library
> SL.library = c("SL.glm", "SL.mean", "SL.glm.interaction")
```

From Super Learning: An Application to the Prediction of HIV-1 Drug Resistance:

Theoretical results show that the super learner will asymptotically outperform any of the candidate estimators it employs as long as the number of candidate learners is polynomial in sample size (or, if one of the candidate estimators it employs achieves a parametric rate of convergence, the super learner will converge at an almost parametric rate). These results suggest that the investigator pays a very small price

for considering multiple alternative learners. Currently, most researchers employ one, or at most a handful, of learning algorithms to answer prediction questions. A better approach would be to apply as many candidate learners as are feasible given time and computing limitations, and choose among them using the super learner

```
> # 7. call superlearner to estimate E[Y|A,W]
> Qbar = SuperLearner(Y=ObsData$Y, X=X, newX=newdata, SL.library=SL.library)


> # 8.
> # expected Y given observed A,W
> QbarAW = Qbar$SL.predict[1:n]
> # expected Y given A=1 and W
> Qbar1W = Qbar$SL.predict[(n+1): (2*n)]
> # expected given A=0 and W
> Qbar0W = Qbar$SL.predict[(2*n+1): (3*n)]


> # 9.
> ### Simple substitution estimator of the ATE ###
> PsiHat.SS = mean(Qbar1W - Qbar0W)


> # 10. create dataframe of covariates
> W = subset(X, select = c(W1, W2))


> # 11. call superlearner to estimate g_0(A|W)
> gHatSL = SuperLearner(Y=ObsData$A, X=W, SL.library=SL.library, family="binomial")


> # 12.
> # generate the predicted prob A=1, given baseline cov
> gHat1W = gHatSL$SL.predict
> # generate the predicted prob of A=0, given baseline cov
> gHat0W = 1 - gHatSL$SL.predict


> # 13.
> ### IPTW estimator of the ATE ###
> IPTW_comps <- cbind(ObsData$A, gHat1W,ObsData$A/gHat1W,  gHat0W, (1-ObsData$A)/gHat0W, ObsData$Y)
> colnames(IPTW_comps) <- c("A", "Predict A = 1", " A = 1 Weight", "Predict A = 0", "A = 0 weight", "Ou
> head(IPTW_comps)


  A Predict A = 1  A = 1 Weight Predict A = 0 A = 0 weight Outcome
1 0    0.04424412      0.000000    0.95575588     1.046292       0
2 1    0.67537229      1.480665    0.32462771     0.000000       1
3 1    0.96175256      1.039768    0.03824744     0.000000       0
4 1    0.55196392      1.811713    0.44803608     0.000000       0
5 0    0.51698380      0.000000    0.48301620     2.070324       0
6 1    0.74813608      1.336655    0.25186392     0.000000       0


> PsiHat.IPTW = mean(ObsData$A*ObsData$Y/gHat1W) -
+   mean((1-ObsData$A)*ObsData$Y/gHat0W)
```

```
> # 14.
> ### TMLE of ATE ###
> PsiHat.TMLE_tmle = tmle(Y = ObsData$Y, A = ObsData$A, W = W,
+                         Q.SL.library = SL.library, g.SL.library = SL.library)
> PsiHat.TMLE_ltmle = summary(ltmle(data=ObsData, Anodes='A', Ynodes='Y',
+                           abar=list(1,0), SL.library=SL.library))
> PsiHat.SS


[1] 0.1139461


> PsiHat.IPTW


[1] 0.114629


> PsiHat.TMLE_ltmle$effect.measures$ATE


$long.name
[1] "Additive Treatment Effect"

$estimate
[1] 0.08336901

$std.dev
[1] 0.05282814

$pvalue
[1] 0.1145386

$CI
            2.5%     97.5%
[1,] -0.02017223 0.1869103

$log.std.err
[1] FALSE


> PsiHat.TMLE_tmle$estimates$ATE


$psi
[1] 0.08220612

$var.psi
[1] 0.002418168

$CI
[1] -0.01417662  0.17858887

$pvalue
[1] 0.09458152
```

Causal I Lab 0 by Alex Leudtke