Lab Assignment #2

Nicholas Noel, Liz Villa

Due February 3, 2023

Instructions

The purpose of this lab is to introduce you to writing *functions* in R. A function is a reproducible bit of code that produces output based on user-specified inputs rather than the "hard-coded" values in your script.

```
library(ISLR2)
library(ggplot2)
library(dplyr)
```

This lab assignment is worth a total of **20 points**.

Problem 1: Introduction to Functions

This problem is adapted from ISLR Chapter 4, Exercise 15.

Part a (0.5 pts)

In the chunk below, write a single line of R code that prints the result of raising the number 2 to the 3rd power.

```
print(2^3)
```

```
## [1] 8
```

Part b (Code: 0.5 pts, Testing: 0.5 pts)

Complete the R chunk below to create and test a function that prints the result of raising *any* number to the 3rd power. Make sure to delete the "eval = FALSE" in the chunk options after you get a chunk that works the way you expect!

```
## Cube function
Cube <- function(x){
  print(x^3)
}
```

It is always a good idea to test any function you write to make sure that it works as you expected.

What do you expect to get when you call the `Cube` function on a single number? Confirm that your function works as expected. Remember to remove `eval = FALSE` after you get the chunk to run the way it's supposed to.

```
Cube(2)
```

```
## [1] 8
```

What happens when you input a vector instead of a single number? Make a guess, then run the chunk below to see whether your guess was correct. If you get an output, remove `eval = FALSE`; if you get an error, keep `eval = FALSE` so that you can still knit without running the code.

```
Cube(c(2, 5, 10))
```

```
## [1]    8  125 1000
```

What happens when you input a character string instead of a number? Make a guess, then run the chunk below to see whether your guess was correct. If you get an output, remove `eval = FALSE`; if you get an error, keep `eval = FALSE` so that you can still knit without running the code.

```
Cube("Math 437")
```

Error shows that you need to be using numeric values.

Part c (Code: 0.5 pts, Testing: 0.5 pts)

Now, we will write another function that prints the result of raising *any* number to *any* exponent. This requires two arguments to the function: the base and the exponent.

It is good form to document all arguments to your function, either before the function or in the first few lines.

```
Power <- function(x, a){
  # x: the base of the power
  # a: the exponent
  print(x^a)
  }
```

What do you expect to get when you run this line of code? Did you get what you expected? If you get an output, remove `eval = FALSE`; if you get an error, keep `eval = FALSE` so that you can still knit without running the code.

```
Power(2, 3)
```

```
## [1] 8
```

What happens when you use a vector instead of a number for the first argument? What about for the second argument? What about for both arguments? Make a guess, then run the chunk below to see whether your guess was correct. If you get an output, remove `eval = FALSE`; if you get an error, keep `eval = FALSE` so that you can still knit without running the code.

```
Power(c(2, 5, 10), 3)
```

```
## [1]    8  125 1000
```

```
Power(2, c(2, 3, 4))
```

```
## [1]  4  8 16
```

```
Power(c(2, 5, 10), c(1, 2, 3))
```

```
## [1]    2   25 1000
```

What happens if you only give one argument? Make a guess, then run the chunk below to see whether your guess was correct. If you get an output, remove `eval = FALSE`; if you get an error, keep `eval = FALSE` so that you can still knit without running the code.

```
Power(2) # What does this do?
```

Part d (Code: 0.5 pts, Testing: 0.5 pts)

Usually we want to *return* the result of the function rather than *print* it, so that we can store the output in an R object.

```
Power_return <- function(x, a){
  # x: the base of the power
  # a: the exponent
  return(x^a)
  # Copy your code from part c here, but replace "print" with "return"
}
```

Let's confirm that we still get $2^3 = 8$ out when we run this line of code. Remember to remove `eval = FALSE` after you get the chunk to run the way it's supposed to.

```
Power_return(2, 3) # Does this still output 2^3 = 8?
```

```
## [1] 8
```

Explain the difference between these two lines of code. (Remember to remove `eval = FALSE` after you get the chunk to run the way it's supposed to.) Where did the 8 go when you ran the second line?

```
result1 <- Power(2, 3)
```

```
## [1] 8
```

```
result2 <- Power_return(2, 3)
```

The difference between the two lines of code is that result1 prints the output into the console whereas result2 stores the output into the R enviorment.

Part e (Code: 0.5 pts; Testing: 0.5 pts)

What if we *usually* want to cube a number, but we also want to give the user the flexibility to include a different exponent if necessary? We can use a *default* argument:

```
Power_default <- function(x, a = 3){
  # x: the base of the power
  # a (= 3): the exponent

  return(x^a)

}
```

Let's confirm that we still get $2^3 = 8$ out when we run this line of code.

```
Power_default(2, 3)
```

```
## [1] 8
```

What happens if you only give one argument? Make a guess, then run the chunk below to see whether your guess was correct. If you get an output, remove `eval = FALSE`; if you get an error, keep `eval = FALSE` so that you can still knit without running the code.

```
Power_default(2)
```

```
## [1] 8
```

By convention, arguments without a default value go at the beginning of the list of arguments and arguments with a default value go at the end.

Part f (Code and Testing: 0.5 pts)

We rarely write one-line functions. Usually we want to store the intermediate and final results in objects within the function, then use `return` to output the final result.

```
Power_result <- function(x, a = 3){
  # x: the base of the power
  # a (= 3): the exponent

  result <- x^a
  return(result)
}
```

Let's just do our usual test to make sure we keep getting $2^3 = 8$ out. Remember to remove `eval = FALSE` after you get the chunk to run the way it's supposed to.

```
Power_result(2, 3)
```

```
## [1] 8
```

Problem 2: From Concept to Code

When we write more complex functions, it is usually easiest to first create a high-level overview of what you want the function to do, then write *pseudocode* explaining step-by-step how you will accomplish it, and then finally turn your algorithm into code in the appropriate programming language.

In this problem, we are going to create a function to perform a two-sample permutation test as an alternative to a two-sample t-test.

Part a (Explanation: 1 pt)

In the high-level overview, we need to think about what we want the function to *output* and what *inputs* we have to give it. The code chunk below takes a data frame and outputs various properties of the indicated two-sample t-test.

```
t.test(formula = , data = , alternative = , var.equal = TRUE)
```

Explain what each of the four arguments (`formula`, `data`, `alternative`, `var.equal`) contributes towards telling R what to do in the pre-built `t.test` function.

The `formula` tells R what to do with the inputs to obtain the desired output generally in the format f(x) ~ x. The `data` argument tells R what data to use in the test. The `alternative` argument tells R what the alternative hypothesis is and needs to be input as text, either "t" for two-sided tests, "g" for greater, or "l" for less. The `var.equal` argument tells R if the variance is constant between samples.

Which arguments (if any) will we need to include when creating our custom permutation test function? What additional arguments (if any) will need to be included because we are using resampling-based methods instead of theory-based methods?

We will need to include the `formula`, `data`, and `var.equal`. Since we are using a resampling-based method, we will also need to include additional arguments to set the seed for reproducibility and to set the number of permutations for the resampling.

Part b (Explanation: 1 pt)

Although R returns an `htest` object when the `t.test` function is called, the `htest` object is basically just a `list` with some extra functionality attached. When we run our permutation test function, what should be included in our list of things to output?

The only values we absolutely need include the values `statistic` and `p.value`

Part c (Explanation: 3 pts)

Now let's write some *pseudocode*. Pseudocode is basically a step-by-step algorithmic description explaining how we're going to accomplish turning our input arguments into the output arguments. We don't actually write any code; the idea is to first informally define our plan before formally implementing it in the chosen programming language. I've started the algorithm below. Based on your understanding of permutation tests from lecture, finish Steps 4 and 5.

Step 1: Run the t-test on the original dataset and obtain the observed t-statistic value

Step 2: Create a vector to store the simulated t-statistics in

Step 3: Set a seed for reproducibility of the resampling

Step 4: For i in 1 to number of permutation resamples:

Step 4a: Randomly reorder the response variable

Step 4b: Run a t-test on the new data

Step 5: Obtain the p-value

Step 6: Create a list containing all the components of the output and then output it

This is actually more detail than is necessary; in real life you'd probably only include steps 1, 4, and 5 in your pseudocode.

Part d (Code: 3 pts)

Now we take our pseudocode and write a script. I've taken care of most of the tricky code, but you need to finish the script. Only Step 1 is fully complete. Once you've completed the other steps and your script runs, remember to delete `eval = FALSE`.

```
# Step 0: Initialize the arguments to your function
set.seed(9034)
data <- data.frame(group = rep(c("Group 1", "Group 2"), 50),
                   y = rnorm(100)) # simulated data

#data <- College %>% select(y = Outstate, group = Private)

formula <- y ~ group
alternative <- "t"
# add something to tell R to stop

# Initialize and assign values to any other arguments you identified in Part a

# Step 1: Run the t-test on the original dataset and obtain the observed t-statistic value

# Creating this permutation_df will allow us to ignore other variables in the data frame.
# This will make Step 4 much easier - see comments in Step 4a.
  permutation_df <- model.frame(formula = formula, data = data)

# We only care about the t-statistic, but we need to store it in a variable
 t_obs <- t.test(formula = formula, data = permutation_df,
                 alternative = alternative, var.equal = TRUE)$stat
# Step 2: Create a vector to store the simulated t-statistics in
  t_perm <- numeric(10000)
```

```
# Step 3: Set a seed for reproducibility of the resampling
set.seed(23)

# Step 4: For i in 1 to number of permutation resamples:
for (i in 1:10000){ # Complete the syntax

  # Step 4a
  permutation_df[[1]] <-sample(permutation_df[[1]])
  # Use the course notes as a template for doing this step
  # Note: Once we get to Part e and put this script inside a function environment,
  # using the actual name of the response variable requires much more advanced R.
  # Remember from Lab 1 that this is an alternative way to do the column indexing
  # and we set up permutation_df so that the first column is the response variable.

  # Step 4b
  t_perm[i] <- t.test(formula, permutation_df, var.equal = TRUE)$statistic
}
# Step 5
T_all <- c(t_obs, t_perm)
  p_left <- sum(T_all <= t_obs)/(length(t_perm) + 1)
  p_right <- sum(T_all >= t_obs)/(length(t_perm) + 1)

  # Use the switch function to compute the correct p-value
  p_value <- dplyr::case_when(alternative == "g" ~ p_right,
                              alternative == "l" ~ p_left,
                              alternative == "t" ~ 2*min(p_left, p_right),
                              TRUE ~ NaN )
# output NaN if alternative is anything else
# Step 6: Create a list containing all the components of the output and then output it
# Ive started the list, but you may need to finish it
  results <- list(obs = t_obs,
                  sim = t_perm,
                  p_val = p_value)
```

Assuming your code runs, you can check what's in `results` by viewing it or

```
str(results)
```

```
## List of 3
##  $ obs  : Named num 1.4
##   ..- attr(*, "names")= chr "t"
##  $ sim  : num [1:10000] -1.519 -0.686 -1.321 -0.4 0.269 ...
##  $ p_val: num 0.16
```

Part e (Code and Testing: 2 pts)

Finally, we define our function, include the variables we defined in Step 0 as the input arguments, use a `return()` statement at the end to return our output, and copy the remainder of the script into the function.

```
permutation_t_test <- function(formula, data, alternative = "t",
                               permutations = 10000, seed = 9034){
 set.seed(seed)

 permutation_df <- model.frame(formula = formula, data = data)
```

```
 t_obs <- t.test(formula = formula, data = data,
                 alternative = alternative, var.equal = TRUE)$stat

 t_perm <- numeric(permutations)


 for (i in 1:length(t_perm)){

  permutation_df[[1]] <-sample(permutation_df[[1]])


  t_perm[i] <- t.test(formula, permutation_df, var.equal = TRUE)$statistic}

 T_all <- c(t_obs, t_perm)

  p_left <- sum(T_all <= t_obs)/(length(t_perm) + 1)

  p_right <- sum(T_all >= t_obs)/(length(t_perm) + 1)

  p_value <- dplyr::case_when(alternative == "g" ~ p_right,
                              alternative == "l" ~ p_left,
                              alternative == "t" ~ 2*min(p_left, p_right),
                              TRUE ~ NaN )

  results <- list(obs = t_obs,
                  sim = t_perm,
                  p_val = p_value)

  return(results) # Last line of the function returns our output
}
```

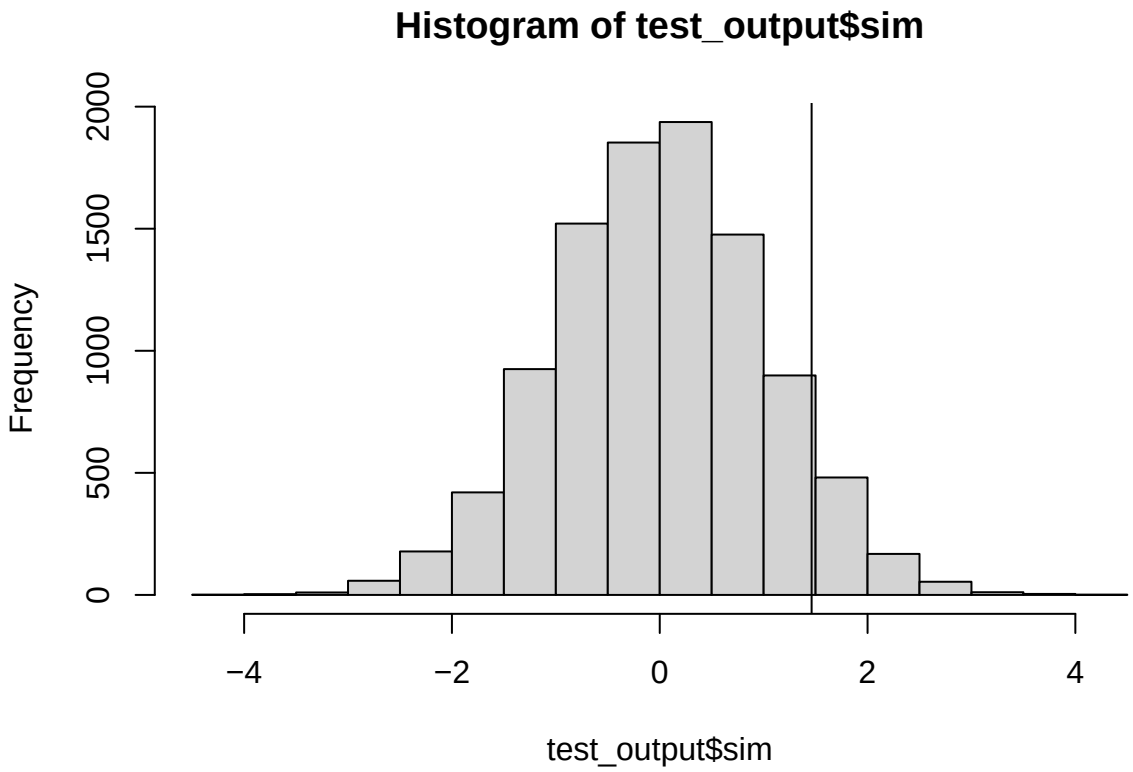
Test the function by completing and running the chunk below.

```
set.seed(9035)
df_new <- data.frame(group = rep(c("Group A", "Group B"), 50),
                     value = rnorm(100))
test_output <- permutation_t_test(formula = value ~ group,
                     data = df_new,
                     alternative = "g", seed = 9035)
# if you did not include default values for your new arguments, make sure to give them values in your f
```

Produce a histogram of the simulated t-statistics and add a vertical line at the observed t-statistic value (it is easiest to use the regular `hist` and `abline` functions). Using the histogram, confirm that the reported p-value seems reasonable.

```
hist(test_output$sim)

abline(v = test_output$obs)
```



Problem 3: Doing a Permutation t-Test

24 "alcohol-dependent" male patients at an alcohol treatment facility were randomly assigned to a traditional treatment program (Control) or a treatment program plus social skills training (SST). 23 patients faithfully reported their alcohol intake for a year (one patient in the SST group never reported). Their group assignment, and alcohol intake over the year (in centiliters of pure alcohol), is found in the *alcohol* dataset on Canvas.

```
alcohol <- read.csv("alcohol.csv")
```

In this problem, we will test the null hypothesis that there is no difference between the groups against the alternative that the Control Group (the default Group 1) had a higher alcohol intake.

Part a (Explanation: 0.5 pts)

Run the chunk below to plot a histogram showing the distribution of Intake in each group.

```
ggplot(data = alcohol, mapping = aes(x = Intake)) +
  geom_histogram(center = 1000, binwidth = 200) +
  facet_wrap(~Group)
```

What do the `center` and `binwidth` arguments do? What does `facet_wrap` do?

Center tells R what to make the midpoint of the x-axis. binwidth sets the size of the bins in the histogram. Facet wrap creates multiple graphs separated by the groups in a specified categorical variable.

Part b (Code: 0.5 pts)

Complete the code chunk below to find the mean and standard deviation of Intake in each group. Remember to remove `eval = FALSE` after you get the chunk to run the way it's supposed to.

```
alcohol %>% group_by(Group) %>% # what do we group by?
  summarize(mean_intake = mean(Intake), # how do we compute the mean of Intake
            sd_intake = sd(Intake)) # how do we compute the sd of Intake?
```

Part c (Code: 1 pt, Explanation: 1 pt)

Using the function you wrote in Problem 2, perform a permutation test of the null hypothesis that there is no difference between the groups against the alternative that the Control Group (the default Group 1) had a *higher* alcohol intake.

Store the output of your function in the variable `alcohol_t_test`, that is, run `alcohol_t_test <- permutation_t_test(...)` and replace the `...` with what you need to run the test. Perform the following sanity checks to make sure your code worked as intended:

```
alcohol_t_test <- permutation_t_test(Intake ~ Group, alcohol,
                                     permutations = 10000)
```

```
# Check to make sure you computed the t-statistic correctly
t.test(Intake ~ Group, data = alcohol, var.equal = TRUE)$statistic # remember to fill in the t.test arg
alcohol_t_test$obs # replace statistic.observed with the name of the variable in the output you stored
```

```
# Check to make sure you have the correct number of resamples
# Is this equal to the number of times you told R to do the permutation resampling?
length(alcohol_t_test$sim) # replace statistic.simulated with the name of the variable in the output yo
```

Based on the permutation test, using a 5% significance level, what should we conclude about the effectiveness of the SST program at reducing alcohol intake? Why?

```
bruh = alcohol_t_test$p_val
```

Since our p-value is 7.9992001×10^{-4} , using a 5% significance level we can conclude that we should reject the null hypothesis that there is no difference between the groups which implies that our alternative hypothesis is correct and the SST program is effective at reducing alcohol intake.

Part d (Code: 1 pt; Explanation: 1 pt)

Using the `College` dataset from the ISLR2 package, and the `permutation_t_test` function you created in Problem 2, perform a permutation t-test to determine if public schools (`Private = No`) charge *less* in out-of-state tuition (`Outstate`) than private schools (`Private = Yes`). Use 999 permutation resamples and seed 12345.

Store the output of your function in the variable `tuition_t_test`, that is, run `tuition_t_test <- permutation_t_test(...)` and replace the `...` with what you need to run the test.

```
#View(College)
tuition_t_test <- permutation_t_test(Outstate ~ Private, data = College,
                                     seed = 12345, permutations = 999)
```

Report the observed value of the t-statistic and the p-value. Additionally, create a histogram of the simulated t-statistic values and add a vertical line at the observed t-statistic value, like you did in Problem 2e. You may have to change the x-axis limits (e.g., `xlim` argument to the base `plot` function) to get the line to show up.

```
t.test(Outstate ~ Private, data = College, alternative = "t" )
```

```
##
##  Welch Two Sample t-test
##
## data:  Outstate by Private
```

```
## t = -23.249, df = 645.59, p-value < 2.2e-16
## alternative hypothesis: true difference in means between group No and group Yes is not equal to 0
## 95 percent confidence interval:
##  -5409.603 -4566.964
## sample estimates:
##  mean in group No mean in group Yes
##           6813.41          11801.69
```

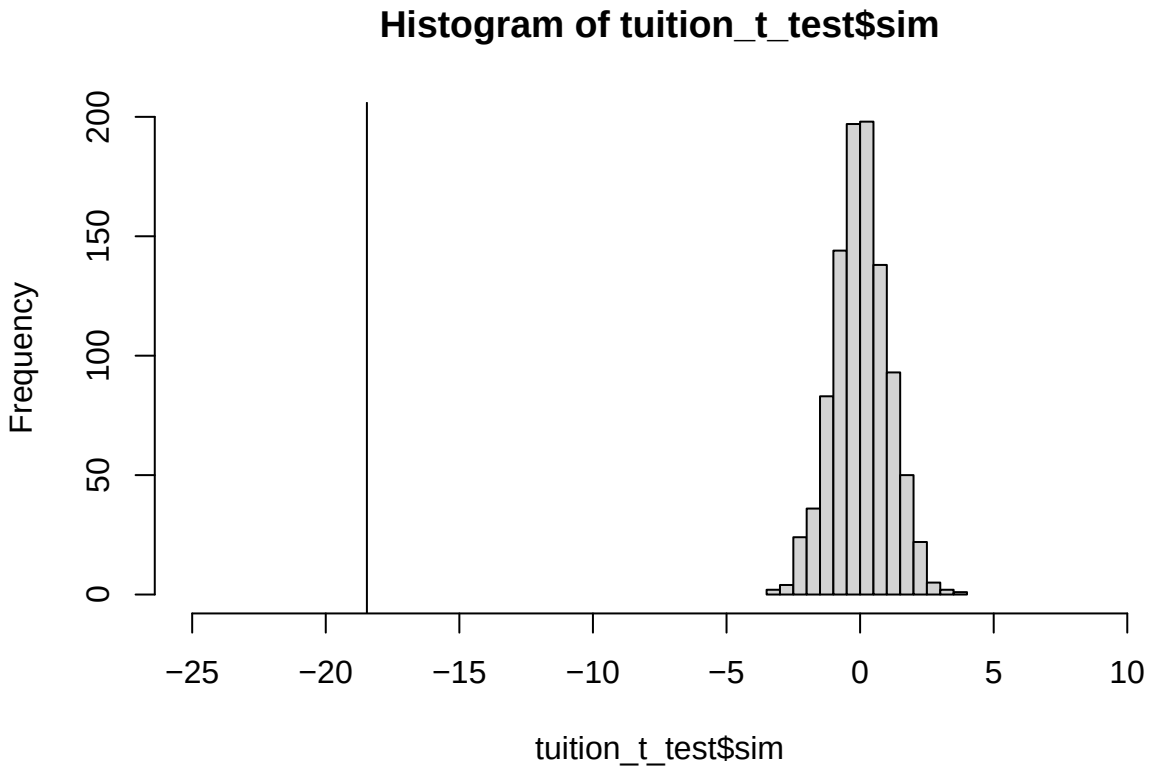
```
tuition_t_test$obs
```

```
##         t
## -18.46037
```

```
tuition_t_test$p_val
```

```
## [1] 0.002
```

```
hist(tuition_t_test$sim, xlim = range(-25,10))
abline(v = tuition_t_test$obs)
```



Based on the permutation test, using a 5% significance level, what should we conclude about the difference in average out-of-state tuition between public and private colleges? Why?

At a significance level of 5% we can reject the null hypothesis and should conclude that the average difference in average out of state tuition between public and private colleges is very significant with a test statistic of -23.249. In particular, out of state tuition of private colleges is significantly higher than those of public colleges.