Lab 5: Writing Functions And Debugging

INSERT YOUR NAME HERE (INSERT YOUR UW NETID HERE)

Due by 23:59pm on Feb 22, 2024

Total Points: 50

Part 1. Writing Functions in R (I) (3+1+1 pts)

1. The soft-threshold function is defined as

$$g_\lambda(x) = \begin{cases} \text{sign}(x) \cdot (|x| - \lambda) & |x| \geq \lambda, \\ 0 & |x| < \lambda. \end{cases}$$

   Write an R function that takes two arguments, numeric vector `x` and a threshold value `lambda`. Your function should return the value of $g_\lambda(x)$ and work for vector input `x` of any length.

```
# Your code goes here
```

2. Set $\lambda = 4$, demonstrate your soft-threshold function on the vector `c(-5, -3, 0, 3, 6)`.

```
# Your code goes here
```

3. Set $\lambda = 2$, demonstrate your function on the vector `c(-8, -5, -3, 0, 3, 5, 7)`.

```
# Your code goes here
```

Part 2. Writing Functions for the Median-of-Means Estimator (7+5+3+2+3+3 pts)

Recall from Lab 2 Part 3-2 that we computed the means of every 10 elements from a vector of length 100 with random points from the binomial distribution $\text{Bin}(m = 300, p = 0.25)$ and then output the median of these 10 mean values. Indeed, we utilized the "Median-of-Means" estimator to estimate the mean of the binomial distribution $\text{Bin}(m = 300, p = 0.25)$ based on its random sample. The idea of this estimator is to divide the random sample with n observations into several subsamples (or buckets) with size K and then compute the mean within each bucket. Then, for all the means from these buckets (there are $\lfloor n/K \rfloor$ in total), we will return the median of them as the final output of this estimator. One advantage of the "Median-of-Means" estimator over the usual mean/average estimator of the random sample is its robustness to the outliers; recall Lab 2 Part 3-3.

In this question, we are going to write our own function to implement the "Median-of-Means" estimator.

1. Generate a random vector `rand_vec` with length `n=160` from the binomial distribution $\text{Bin}(m = 300, p = 0.25)$. Then, we compute the "Median-of-Means" estimator on `rand_vec` as follows:

- Assign a bucket size variable with value 10 (i.e., `bucket_size = 10`).
- Compute the number of buckets `num_bucket` using the integer division `n %/% bucket_size`.
- Leverage the `rep_len()` function to generate a vector with the same length as `rand_vec`, whose elements are `1:num_bucket`, and then coerce it into a factor vector `group_vec`.
- Use the `tapply()` function to compute the mean within each factor level.
- Finally, compute the median of those mean values from the result of the `tapply()` function. Output this final median value, which will be the "Median-of-Means" estimator on the `rand_vec`.

```r
set.seed(123)  ## Don't change this line. It makes the result reproducible.
# Your code goes here
```

2. Change the length of a random vector to $n = 206$, i.e., generate a new random vector `rand_vec2` with length `n=206` from the binomial distribution $\text{Bin}(m = 300, p = 0.25)$. Redo the steps in the above subquestion to compute the "Median-of-Means" estimator on this new vector `rand_vec2`.

- Answer in words: Are the sizes of those subsamples (or buckets) the same in this case? Why does our code still work even when the length of the random vector `rand_vec2` is not a multiple of the bucket size? (Hint: Recall Slide 52 in Lecture 3. What R mechanism plays a role here?)

```r
set.seed(123)  ## Don't change this line. It makes the result reproducible.
# Your code goes here
```

3. Now, we will abstract the above procedure into a function called `MedianOfMean()` with two input arguments:

- `x` stands for the input random vector on which we want to estimate the mean (Abstracting the `rand_vec` above).
- `bucket_size` is the bucket size (i.e., the number of observations in each subsample).

The return value of the function `MedianOfMean()` is the "Median-of-Means" estimator on `x`. Then, evaluate your function `MedianOfMean()` on the random vector `rand_vec2` in subquestion 2 with bucket size 10 and check if the outputs are the same. (Hint: Change the `rand_vec` in subquestion 1 to `x` in the body of the function. Be careful about whether other variable names need to change or assign within the function.)

```r
# Your code goes here
```

4. We should also let the user know when the length of the input vector `x` is not a multiple of the bucket size `bucket_size`. Modify the above `MedianOfMean()` function so that, as a side effect, it prints out the following two messages when the length of the input vector `x` is not a multiple of the bucket size `bucket_size`.

- "The length of the input vector `x` is not a multiple of the `bucket_size` parameter."
- "There are some buckets that have one more observation than others."

Then, evaluate your function `MedianOfMean()` on the random vector `rand_vec2` in subquestion 2 with bucket size **12**.

```r
# Your code goes here
```

5. Based on the `MedianOfMean()` function in subquestion 4, return not just the "Median-of-Means" estimator on `x` but also the bucket size and the bucket affiliation of `x` (i.e., the factor vector that we create for using `tapply()`). (Hint: Create a list with three components:

- `MoM_est`: the final value of the "Median-of-Means" estimator on `x`;
- `bucket_size`: the bucket size as the input argument;
- `group_aff`: the factor vector that indicates the bucket affiliation of `x`.)

Then, evaluate your function `MedianOfMean()` on the random vector `rand_vec2` in subquestion 2 with bucket size **12**.

```r
# Your code goes here
```

6. Since the input parameter `x` must be a numeric vector, add a check to the function `MedianOfMean()` in subquestion 5 to ensure that the input `x` has length greater than 1 and is of the data type `numeric`. (Hint: Use the `stop()` function when the conditions are not met and print out the message `"\n The input vector should be of its length greater than 1 and of the numeric data type."`.)

Then, evaluate your function `MedianOfMean()` on the vector `letters` with bucket size **12**.

```
# Your code goes here (Don't change the option `error=TRUE` in this chunk!!)
```

Part 3: Exploring Function Environments (4+2+4 pts)

1. Copy and paste your `MedianOfMean()` function in the Part 2-6 to the below chunk. Assign a value 10 to the variable `bucket_size` in the global environment (simply by writing `bucket_size = 10` outside of your function body). Then, evaluate your function `MedianOfMean()` on the random vector `rand_vec2` in Part 2-2 with bucket size **12**.

- Answer in words: Is the final value of the "Median-of-Means" estimator equal to the one in Part 2-3 or Part 2-5? Explain why?
- Print out the value of `bucket_size` in the global environment. Is it equal to 10 or 12?

```
# Your code goes here
```

2. We are going to explain the difference between `=` and `<-` in this subquestion. The equal sign `=` and assignment operator `<-` are often used interchangeably in R, and some people (including myself) will often consider that a choice between the two is mostly a matter of stylistic taste. To my surprise, this is not the full story. Indeed, `=` and `<-` behave very differently when used to set input arguments in a function call. As we showed above, setting, say, `bucket_size = 12` as the input to `MedianOfMean()` has no effect on the global assignment for `bucket_size`. However, replacing `bucket_size = 12` with `bucket_size <- 12` in the call to `MedianOfMean()` is entirely different in terms of its effect on `bucket_size`. Demonstrate this, and explain what you are seeing in terms of global assignment. (Hint: As in subquestion 1, first assign a value 10 to the variable `bucket_size` in the global environment. Then call the function `MedianOfMean()` as above, but replace `bucket_size = 12` with `bucket_size <- 12` in its input argument. Finally, print out the value of `bucket_size` in the global environment. See whether it is equal to 10 or 12.)

```
# Your code goes here
```

3. The story now gets even more subtle. It turns out that the assignment operator `<-` allows us to define new global variables even when we are specifying inputs to a function. Pick a variable name that has not been defined yet in your workspace, say `b` (or something else, if this has already been used in your R Markdown document). Call `MedianOfMeans(x = rand_vec2, b <- 15)`, then display the value of `b` — this variable should now exist in the global environment, and it should be equal to 15! Also, can you explain the output of `MedianOfMeans(x = rand_vec2, b <- 15)`? (Hint: Is the output from `MedianOfMeans` computed based on the `bucket_size = 15` or other values? Why?)

```
# Your code goes here
```

Part 4: Practice with `browser()` (3+4+2+3 pts)

1. Below is a function `add_inv_powers()` that computes $1^1 + 2^{1/2} + \ldots + (n-1)^{1/(n-1)} + n^{1/n}$, via a `for()` loop, for some value of n specified in the first argument. The second argument is `verbose`; if this is `TRUE` (the default is `FALSE`), then the function will print out the current summand to the console, as a roman numeral. A short demo is given below. You will use `add_inv_powers()` and `roman_cat()` to do a bit of exploration with `browser()` in the next several questions. But before this, for good vectorization practice, show that you can compute the same expression as done in `add_inv_powers()`, but without any explicit looping, i.e., just using vectorization and `sum()`. Check that you get the same answers for the demo inputs. (Hint: you can use `all.equal()` to check for "very near" equality, since you may not get exact equality in all digits.)

```r
add_inv_powers = function(n, verbose=FALSE) {
  x = 0
  for (i in 1:n) {
    x = x + i^(1/i)
    if (verbose) roman_cat(i)
  }
  if (verbose) cat("\n")
  return(x)
}

roman_cat = function(num) {
  roman.num = as.roman(num)
  roman.str = as.character(roman.num)
  cat(roman.str, "... ")
}

add_inv_powers(n=3, verb=FALSE)
```

```
## [1] 3.856463
```

```r
add_inv_powers(n=5, verb=FALSE)
```

```
## [1] 6.650406
```

```r
add_inv_powers(n=10, verb=FALSE)
```

```
## [1] 13.15116
```

```r
# Your code goes here
```

2. Running the code chunk below will enter you to the RStudio browser mode.

```r
add_inv_powers = function(n, verbose=FALSE) {
  x = 0
  browser()
  for (i in 1:n) {
    x = x + i^(1/i)
    if (verbose) roman_cat(i)
  }
  if (verbose) cat("\n")
  return(x)
}

add_inv_powers(n = 5, verb = TRUE)
```

First, just look around: you should see the "Console" panel (as always), the "Source Viewer" panel, the "Environment" panel, and the "Traceback" panel. (The console is arguably the most important but the others add nice graphical displays.) Hit the return key repeatedly (while your cursor is in the console) to step through the function line by line, until you get to the last line of the function. Once this last line is run, you'll immediately exit the browser mode. Try the whole process again a few times, each time looking at the various R Studio panels and familiarizing yourself with what they are displaying. Instead of hitting the return key, note that you can type "n" in the console to step to the next line. Note also that you can type in variable names in the console and hit enter, to see their current values (alternatively, the "Environment" panel will show you this too).

Answer the following questions, exploring what you can do in the browser mode.

- How do you display the value of the variable `n` defined in the `add_inv_powers()` function? (Recall that typing "n" just gives you the next line.)

Answer: YOU ANSWER GOES HERE.

- How do you exit the browser mode prematurely, before the last line is reached?

Answer: YOU ANSWER GOES HERE.

- Suppose that you are running the browser mode with a call like `new_num = 6` in the console. After you run the browser mode to completion, would the variable `new_num` be defined in your console?

Answer: YOU ANSWER GOES HERE.

- Can you redefine existing variables in the browser? What happens, for example, if you were to redefine `x` the moment you entered the browser mode?

Answer: YOU ANSWER GOES HERE.

3. Typing the "f" key in browser mode, as soon as you enter a `for()` loop, will skip to the end of the loop. Try this a few times. What happens if you type "f" after say a few iterations of the loop? What happens if you type "f" right before the loop?

Answer: YOU ANSWER GOES HERE.

Now, typing the "c" key in browser mode will exit browser mode and continue on with normal evaluation. Try this too. Lastly, typing the "s" key in browser mode will put you into an even more in-depth mode, call it "follow-the-rabit-hole" mode, where you step into each function being evaluated, and enter browser mode for that function. Try this, and describe what you find. Can you step into `roman_cat()`? Can you step into even those build-in functions?

Answer: YOU ANSWER GOES HERE.

4. Now, you've had good practice with the `browser()` function. Use it to find and fix bugs in the function `fibonacci()` below. This function is supposed to generate the n-th number in the Fibonacci sequence 1, 1, 2, 3, 5, 8, 13, 21, 34, . . . , which begins with 1, 1, and where every number after this is the sum of the previous two. Describe what bugs you found, how you found them, and what you did to fix them. Once this is done, your function should be producing outputs on the test cases below that match those described in the comments.

Answer: YOU SHORT DESCRIPTION GOES HERE.

```
Fibonacci = function(n) {
  my_fib = c(1, 1)
  for (i in 2:(n-1)) my_fib[i+1] = my_fib[i] + my_fib[i-1]
  return(my_fib[i])
}

Fibonacci(1) # Should be 1
```

```
## Error in my_fib[i + 1] <- my_fib[i] + my_fib[i - 1]: replacement has length zero
```

```
Fibonacci(2) # Should be 1
```

```
## Error in my_fib[i + 1] <- my_fib[i] + my_fib[i - 1]: replacement has length zero
```

```
Fibonacci(3) # Should be 2
```

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

```
Fibonacci(5) # Should be 5
```

```
## [1] 3
```

```
Fibonacci(9) # Should be 34
```

```
## [1] 21
```

```
# Your correct function code goes here
```

```
## Uncomment the following code lines once you have your correct function.
# fibonacci(1)
# fibonacci(2)
# fibonacci(3)
# fibonacci(5)
# fibonacci(9)
```