Lab 8: Functions

Statistical Computing, 36-350

Week of Tuesday October 19, 2021

Name:
Andrew ID:
Collaborated with:

This lab is to be done in class (completed outside of class time if need be). You can collaborate with your classmates, but you must identify their names above, and you must submit **your own** lab as an knitted PDF file on Gradescope, by Friday 9pm, this week.

This week's agenda: creating and updating functions; understanding argument and return structures; revisiting Shakespeare's plays; code refactoring.

Huber loss function

The Huber loss function (or just Huber function, for short) is defined as:

$$\psi(x) = \begin{cases} x^2 & \text{if } |x| \leq 1 \\ 2|x| - 1 & \text{if } |x| > 1 \end{cases}$$

This function is quadratic on the interval [-1,1], and linear outside of this interval. It transitions from quadratic to linear "smoothly", and looks like this. It is often used in place of the usual squared error loss for robust estimation. For example, the sample average, $\bar{X}$—which given a sample $X_1, \ldots, X_n$ minimizes the squared error loss $\sum_{i=1}^n (X_i - m)^2$ over all choices of m—can be inaccurate as an estimate of $\mathbb{E}(X)$ if the distribution of X is heavy-tailed. In such cases, minimizing Huber loss can give a better estimate.

Q1. Some simple function tasks

- **1a.** Write a function `huber()` that takes as an input a number x, and returns the Huber value $\psi(x)$, as defined above. Hint: the body of a function is just a block of R code, e.g., in this code you can use `if()` and `else()` statements. Check that `huber(1)` returns 1, and `huber(4)` returns 7.

```
# YOUR CODE GOES HERE
```

- **1b.** The Huber function can be modified so that the transition from quadratic to linear happens at an arbitrary cutoff value a, as in:

$$\psi_a(x) = \begin{cases} x^2 & \text{if } |x| \leq a \\ 2a|x| - a^2 & \text{if } |x| > a \end{cases}$$

  Starting with your solution code to the last question, update your `huber()` function so that it takes two arguments: x, a number at which to evaluate the loss, and a a number representing the cutoff value. It should now return $\psi_a(x)$, as defined above. Check that `huber(3, 2)` returns 8, and `huber(3, 4)` returns 9.

```
# YOUR CODE GOES HERE
```

- **1c.** Update your `huber()` function so that the default value of the cutoff a is 1. Check that `huber(3)` returns 5.

```
# YOUR CODE GOES HERE
```

- **1d.** Check that `huber(a=1, x=3)` returns 5. Check that `huber(1, 3)` returns 1. Explain why these are different.

```
# YOUR CODE GOES HERE
```

- **1e.** Vectorize your `huber()` function, so that the first input can actually be a vector of numbers, and what is returned is a vector whose elements give the Huber evaluated at each of these numbers. Hint: you might try using `ifelse()`, if you haven't already, to vectorize nicely. Check that `huber(x=1:6, a=3)` returns the vector of numbers (1, 4, 9, 15, 21, 27).

```
# YOUR CODE GOES HERE
```

- **Challenge.** Your instructor computed the Huber function values $\psi_a(x)$ over a bunch of different x values, stored in `huber.vals` and `x.vals`, respectively. However, the cutoff a was, let's say, lost. Using `huber.vals`, `x.vals`, and the definition of the Huber function, you should be able to figure out the cutoff value a, at least roughly. Estimate a and explain how you got there. Hint: one way to estimate a is to do so visually, using plotting tools; there are other ways too.

```
# YOUR CODE GOES HERE
```

Q2. Plotting practice, side effects

- **2a.** Professor Tibs created in plot of the Huber function displayed here. Reproduce this plot with your own plotting code, and the `huber()` function you wrote above. The axes and title should be just the same, so should the Huber curve (in black), so should be the red dotted lines at the values -1 and 1, and so should the text "Linear", "Quadratic", "Linear".

```
# YOUR CODE GOES HERE
```

- **2b.** Modify the `huber()` function so that, as a side effect, it prints the string "Invented by the great Swiss statistician Peter Huber!" to the console. Hint: use `cat()`. Call your function on an input of your choosing, to demonstrate this side effect.

```
# YOUR CODE GOES HERE
```

- **Challenge.** Further modify your `huber()` function so that, as another side effect, it produces a plot of Switzerland's national flag. Hint: look up this flag up on Google; it's pretty simple; and you should be able to recreate it with a few calls to `rect()`. Call your function on an input of your choosing, to demonstrate its side effects.

```
# YOUR CODE GOES HERE
```

Q3. Exploring function environments

- **3a.** A modified version of the Huber function is given below. You can see that we've defined the variable `x.squared` in the body of the function to be the square of the input argument `x`. In a separate line of code (outside of the function definition), define the variable `x.squared` to be equal to 999. Then call `huber(x=3)`, and display the value of `x.squared`. What is its value? Is this affected by the function call `huber(x=3)`? It shouldn't be! Reiterate this point with several more lines of code, in which you repeatedly define `x.squared` to be something different (even something nonnumeric, like a string), and then call `huber(x=3)`, and demonstrate afterwards that the value of `x.squared` hasn't changed.

```
huber = function(x, a=1) {
  x.squared = x^2
```

```
  ifelse(abs(x) <= a, x.squared, 2*a*abs(x)-a^2)
}
```

```
# YOUR CODE GOES HERE
```

- **3b.** Similar to the last question, define the variable `a` to be equal to -59.6, then call `huber(x=3, a=2)`, and show that the value of `a` after this function call is unchanged. And repeat a few times with different assignments for the variable `a`, to reiterate this point.

```
# YOUR CODE GOES HERE
```

- **3c.** The previous two questions showed you that a function's body has its own environment in which locally defined variables, like those defined in the body itself, or those defined through inputs to the function, take priority over those defined outside of the function. However, when a variable referred to the body of a function is *not defined in the local environment*, the default is to look for it in the global environment (outside of the function).

  Below is a "sloppy" implementation of the Huber function called `huber.sloppy()`, in which the cutoff `a` is not passed as an argument to the function. In a separate line of code (outside of the function definition), define `a` to be equal to 1.5 and then call `huber.sloppy(x=3)`. What is the output? Explain. Repeat this a few times, by defining `a` and then calling `huber.sloppy(x=3)`, to show that the value of `a` does indeed affect the function's ouptut as expected. **Challenge**: try setting `a` equal to a string and calling `huber.sloppy(x=3)`; can you explain what is happening?

```
huber.sloppy = function(x) {
  ifelse(abs(x) <= a, x^2, 2*a*abs(x)-a^2)
}
```

```
# YOUR CODE GOES HERE
```

- **3d.** At last, a difference between `=` and `<-`, explained! Some of you have been asking about this. The equal sign `=` and assignment operator `<-` are often used interchangeably in R, and some people will often say that a choice between the two is mostly a matter of stylistic taste. 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, `a=5` as the input to `huber()` has no effect on the global assignment for `a`. However, replacing `a=5` with `a<-5` in the call to `huber()` is entirely different in terms of its effect on `a`. Demonstrate this, and explain what you are seeing in terms of global assignment.

```
# YOUR CODE GOES HERE
```

- **3e.** 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 `huber(x=3, b<-20)`, then display the value of `b`—this variable should now exist in the global enviroment, and it should be equal to 20! Also, can you explain the output of `huber(x=3, b<-20)`?

```
# YOUR CODE GOES HERE
```

- **Challenge.** The property of the assignment operator `<-` demonstrated in the last question, although tricky, can also be pretty useful. Leverage this property to plot the function $y = 0.05x^2 - \sin(x)\cos(x) + 0.1\exp(1 + \log(x))$ over 50 x values between 0 and 2, using only one line of code and one call to the function `seq()`.

```
# YOUR CODE GOES HERE
```

- **Challenge.** Give an example to show that the property of the assignment operator `<-` demonstrated in the last two questions does not hold in the body of a function. That is, give an example in which `<-` is used in the body of a function to define a variable, but this doesn't translate into global assignment.

```
# YOUR CODE GOES HERE
```

Shakespeare's complete works

Recall, as we saw in Week 4, that the complete works of William Shakespeare are available freely from Project Gutenberg. We've put this text file up at http://www.stat.cmu.edu/~ryantibs/statcomp/data/shakespeare.txt.

Q4. Getting lines of text play-by-play

- **4a.** Below is the `get.wordtab.from.url()` from lecture. Modify this function so that the string vectors `lines` and `words` are both included as named components in the returned list. For good practice, update the documentation in comments to reflect your changes. Then call this function on the URL for the Shakespeare's complete works: http://www.stat.cmu.edu/~ryantibs/statcomp/data/shakespeare.txt (with the rest of the arguments at their default values) and save the result as `shakespeare.wordobj`. Using `head()`, display the first several elements of (definitely not all of!) the `lines`, `words`, and `wordtab` components of `shakespeare.wordobj`, just to check that the output makes sense to you.

```
# get.wordtab.from.url: get a word table from text on the web
# Inputs:
# - str.url: string, specifying URL of a web page
# - split: string, specifying what to split on. Default is the regex pattern
#   "[[:space:]]|[[:punct:]]"
# - tolower: Boolean, TRUE if words should be converted to lower case before
#   the word table is computed. Default is TRUE
# - keep.nums: Boolean, TRUE if words containing numbers should be kept in the
#   word table. Default is FALSE
# Output: list, containing lines, words, word table, and some basic summaries

get.wordtab.from.url = function(str.url, split="[[:space:]]|[[:punct:]]",
                                tolower=TRUE, keep.nums=FALSE) {
  lines = readLines(str.url)
  text = paste(lines, collapse=" ")
  words = strsplit(text, split=split)[[1]]
  words = words[words != ""]

  # Convert to lower case, if we're asked to
  if (tolower) words = tolower(words)

  # Get rid of words with numbers, if we're asked to
  if (!keep.nums)
    words = grep("[0-9]", words, inv=TRUE, val=TRUE)

  # Compute the word table
  wordtab = table(words)

  return(list(wordtab=wordtab,
              number.unique.words=length(wordtab),
              number.total.words=sum(wordtab),
              longest.word=words[which.max(nchar(words))]))
}
```

- **4b.** Go back and look Q5 of Lab 6, where you located Shakespeare's plays in the lines of text for

Shakespeare's complete works. Set `shakespeare.lines = shakespeare.wordobj$lines`, and then rerun your solution code (or the rerun the official solution code, if you'd like) for Q5 of Lab 6 on the lines of text stored in `shakespeare.lines`. (Note: you don't actually need to rerun the code for Q5d or Q5e, since the code for Q5f will accomplish the same task only without encountering `NA`s). You should end up with two vectors `titles.start` and `titles.end`, containing the start and end positions of each of Shakespeare's plays in `shakespeare.lines`. Print out `titles.start` and `titles.end` to the console.

```
# YOUR CODE GOES HERE
```

- **4c.** Create a list `shakespeare.lines.by.play` of length equal to the number of Shakespeare's plays (a number you should have already computed in the solution to the last question). Using a `for()` loop, and relying on `titles.start` and `titles.end`, extract the appropriate subvector of `shakespeare.lines` for each of Shakespeare's plays, and store it as a component of `shakespeare.lines.by.play`. That is, `shakespeare.lines.by.play[[1]]` should contain the lines for Shakespeare's first play, `shakespeare.lines.by.play[[2]]` should contain the lines for Shakespeare's second play, and so on. Name the components of `shakespeare.lines.by.play` according to the titles of the plays.

```
# YOUR CODE GOES HERE
```

- **4d.** Using one of the apply functions, along with `head()`, print the first 4 lines of each of Shakespeare's plays to the console (sorry graders . . .). This should only require one line of code.

```
# YOUR CODE GOES HERE
```

Q5. Getting word tables play-by-play

- **5a.** Define a function `get.wordtab.from.lines()` to have the same argument structure as `get.wordtab.from.url()`, which recall you last updated in Q2a, except that the first argument of `get.wordtab.from.lines()` should be `lines`, a string vector passed by the user that contains lines of text to be processed. The body of `get.wordtab.from.lines()` should be the same as `get.wordtab.from.url()`, except that `lines` is passed and does not need to be computed using `readlines()`. The output of `get.wordtab.from.lines()` should be the same as `get.wordtab.from.url()`, except that `lines` does not need to be returned as a component. For good practice, incude documentation for your `get.wordtab.from.lines()` function in comments.

```
# YOUR CODE GOES HERE
```

- **5b.** Using a `for()` loop or one of the apply functions (your choice here), run the `get.wordtab.from.lines()` function on each of the components of `shakespeare.lines.by.play`, (with the rest of the arguments at their default values). Save the result in a list called `shakespeare.wordobj.by.play`. That is, `shakespeare.wordobj.by.play[[1]]` should contain the result of calling this function on the lines for the first play, `shakespeare.wordobj.by.play[[2]]` should contain the result of calling this function on the lines for the second play, and so on.

```
# YOUR CODE GOES HERE
```

- **5c.** Using one of the apply functions, compute numeric vectors `shakespeare.total.words.by.play` and `shakespeare.unique.words.by.play`, that contain the number of total words and number of unique words, respectively, for each of Shakespeare's plays. Each vector should only require one line of code to compute. Hint: recall `` `[[`() `` is actually a function that allows you to do extract a named component of a list; e.g., try `` `[[`(shakespeare.wordobj, "number.total.words") ``, and you'll see this is the same as `shakespeare.wordobj[["number.total.words"]]`; you should take advantage of this functionality in your apply call. What are the 5 longest plays, in terms of total word count? The 5 shortest plays?

```
# YOUR CODE GOES HERE
```

- **5d.** Plot the number of unique words versus number of total words, across Shakeapeare's plays. Set

the title and label the axes appropriately. Is there a consistent trend you notice?

```
# YOUR CODE GOES HERE
```

Q6. Refactoring the word table functions

- **6.** Look back at `get.wordtab.from.lines()` and `get.wordtab.from.url()`. Note that they overlap heavily, i.e., their bodies contain a lot of the same code. Redefine `get.wordtab.from.url()` so that it just calls `get.wordtab.from.lines()` in its body. Your new `get.wordtab.from.url()` function should have the same inputs as before, and produce the same output as before. So externally, nothing will have changed; we are just changing the internal structure of `get.wordtab.from.url()` to clean up our code base (specifically, to avoid code duplication in our case). This is an example of **code refactoring**.

  Call your new `get.wordtab.from.url()` function on the URL for Shakespeare's complete works, saving the result as `shakespeare.wordobj2`. Compare some of the components of `shakespeare.wordobj2` to those of `shakespeare.wordobj` (which was computed using the old function definition) to check that your new implementation works as it should.

```
# YOUR CODE GOES HERE
```

- **Challenge.** Check using `all.equal()` whether `shakespeare.wordobj` and `shakespeare.wordobj2` are the same. Likely, this will not return `TRUE`. (If it does, then you've already solved this challenge question!) Modify your `get.wordtab.from.url()` function from the last question, so that it still calls `get.wordtab.from.lines()` to do the hard work, but produces an output exactly the same as the original `shakespeare.wordobj` object. Demonstrate your success by calling `all.equal()` once again.

```
# YOUR CODE GOES HERE
```