BIOS 545 Spring 2015 Homework 2

Pittard

Due by 11:59 PM on February 14, 2015

Instructions

Send responses in a plain text file named LastName_Firstname_HW2.txt or LastName_Firstname_HW2.R. You should use RStudio to create this file of course. The subject line of your email should reference "BIOS545 Homework 2". We will run your commands at the R console to verify the statements. Email to BOTH `dvandom@emory.edu` and `wsp@emory.edu`. You may **not** install any addon packages to complete this assignment. Late submissions will be penalized at 10 percent for each day late. Note that the first day begins immediately after the 11:59 p.m. deadline.

1 Newton's Method - 15 points

Write a function that implements Newton's method to find the square root of a given number. Here is a suggested shell for a function called "mynewton" that could be used to develop your code.

```
mynewton <- function(n,guess,toler=0.0001,guesses=T) {

  # Function to compute square root of a number n
  # INPUT:  n:       a positive number
  #         guess:   our initial guess
  #         toler:   a tolerance threshold
  #         guesses: a T/F value indicating whether to return a vector with
  #                  guesses (except first one) Default T

  # OUTPUT: a named list containing
  #         1) answer:     the computed answer
  #         2) iterations: the number of iterations required to get answer
  #         3) guesses:    a vector containing all the
  #                        computed guesses (except the first one)
  #

   # Your code goes here
```

```
}  # end function
```

Table 1: Arguments to mynewton function

Argument	Purpose
n	a positive number whose square root you wish to compute
guess	an initial guess for the square root of n
tolerance	how close the improved guess squared must be to n before you end the method
guesses	a TRUE/FALSE value to return a vector containing all guesses (except the first)

```
# And here is how a call to this function might look

# Example 1

mynewton(121,9)
$answer
[1] 11

$iterations
[1] 3

$guesses
[1] 11.22222 11.00220 11.00000

# Example 2

mynewton(121,14)
$answer
[1] 11

$iterations
[1] 3

$guesses
[1] 11.32143 11.00456 11.00000

# Example 3

mynewton(121,14,guesses=F)
$answer
[1] 11
```

```
$iterations
[1] 3

# Example 4

mynewton(121,14,toler=0.00001,guesses=F)
$answer
[1] 11

$iterations
[1] 4
```

The steps involved to compute the square root of a number n using Newton's method is as follows:

1. Get the target number n (user supplied - e.g. 121)
2. Get the first guess (user supplied - e.g. 9)
3. Get the tolerance value (user supplied - e.g. .0001). This specifies how close the guess squared needs to be to n before it is acceptable.
4. Compute the difference between the supplied guess squared and the given target number n. Is it less than the specified tolerance value ?
5. If it is then you are done. If not then we use Newton's formula to improve upon our guess.

   ```
   guess <- (n/guess + guess)/2
   ```

6. While the difference between our improved guesses and n is greater than or equal to the specified tolerance we repeat steps 4 and 5.

Use the given examples to help check your code. Here is some pseudo code you might find to be helpful. Note there are other ways to approach this so this is just a suggestion. Please use error checking on the arguments to insure that the user is providing valid input.

```
Setup a list to return the answers as specified
Setup an  accumulator variable for the number of iterations
Setup an empty vector to capture all computed guesses (except first one)
Compute the initial difference between n and initial guess squared

While the difference between the guess squared and the target number n is
greater than or equal to the tolerance
  improve the guess using newton's formula
  recompute the difference
  increment the number of iternations accumulator
  put the current guess into vector for holding computed guesses

Return the answer list with named elements
```

2 Data Frames - 25 points

Write a function called my.sampler that fulfills the following requirements:
It will take a data frame, such as mtcars,(although it will need to work for any data frame), and "split" it up based on a given grouping variable. Then it will process each group and sample a specified number of records, (without replacement), from each group. When it is finished processing all groups it will combine those results into a data frame and return it. You need only accommodate a single group/factor per function call. As an example:

```
my.sampler(mtcars,mtcars$cyl,3)
                   mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Toyota Corona     21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
Honda Civic       30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
Merc 230          22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
Merc 280          19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
Mazda RX4         21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Merc 280C         17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
AMC Javelin       15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
Cadillac Fleetwood 10.4  8 472.0 205 2.93 5.250 17.98  0  0    3    4

my.sampler(ChickWeight,ChickWeight$Diet,2)  # ChickWeight is built-in to R
    weight Time Chick Diet
20     138   14     2    1
57     197   16     5    1
253    145   16    23    2
270     49    2    25    2
361    221   16    32    3
389     41    0    35    3
513    135   12    45    4
554    322   21    48    4
```

Table 2: Arguments to my.sampler function

Argument	Purpose
my.df	a dataframe (e.g. mtcars, ChickWeight, etc)
my.group	a grouping variable from my.df
numtosample	the number of records to sample from each value of my.group

Your function will have to work with the other potential grouping variables from the given data set (in the case of mtcars - gear, transmission, carb, and vs). Once you have split the data frame on a factor you should be able to, as part of your looping or apply logic, determine how many rows are in each category

using functions you've learned about in class. We will also run your function on the ChickWeight data and the diamonds dataset, (in the ggplot2 package), to insure that there is nothing specific in it to mtcars

THINGS TO CHECK: Insure that the number of unique values that the grouping variable takes on is less than 10. So if you are given a factor to split on then check that it assumes at most 10 unique values - otherwise complain to the user with an appropriate error message.

You should also do error checking to make sure that the number of records you are attempting to sample is not larger than the available number of records in the group. In that case then your program will stop with an informative message. For example in looking at the mtcars data, one can see that there is only one car that has a value of 6 in the carb column, thus

```
my.sampler(mtcars,mtcars$carb,2)  would fail:

Error in my.sampler(mtcars, mtcars$carb, 2) :
  Number of requested samples 2 exceeds available records

# As would the following because MPG really isn't a grouping variable:

my.sampler(mtcars,mtcars$mpg,5)
Error in my.sampler(mtcars, mtcars$mpg, 5) :
  Grouping factor has 25 unique values. Must be less than 10
```

Lastly, here are "recipes" you might find to be helpful for combining list elements back into a data frame. The first uses the "rbind" function that combines two data frame together. Using mtcars as an example:

```
mys <- split(mtcars,mtcars$cyl)
df <- data.frame()
for (ii in 1:length(mys)) {
   df <- rbind(df,mys[[ii]])
}

# or

unsplit(mys,mtcars$cyl)  # must unsplit using the original factor by which you first split
```

3 transprot - 30 points

Write a function called transprot that, given any nucelotide dna sequence, `ACGTs`, will return the translated protein. *You may not use any add on packages such as Bioconductor to solve this problem.*

Table 3: Arguments to transprot function

Argument	Purpose
sequence	A DNA sequence string (e.g. "AGTGTGC", c("A","T","T","G","C"))
collapsed (default FALSE)	If TRUE then return a collapsed string

Here is some background information to help you understand the problem. First, from an R prompt copy and paste in the following code block. This will create a vector called **stdcode** that contains a series of "triplets" that represent amino acids/ proteins. Note, **you will also need to paste this block inside your function definition so your function will have access to the vector**.

```
stdcode <- structure(c("TTT", "TCT", "TAT", "TGT", "CTT", "CCT", "CAT",
"CGT", "ATT", "ACT", "AAT", "AGT", "GTT", "GCT", "GAT", "GGT",
"TTC", "TCC", "TAC", "TGC", "CTC", "CCC", "CAC", "CGC", "ATC",
"ACC", "AAC", "AGC", "GTC", "GCC", "GAC", "GGC", "TTA", "TCA",
"TAA", "TGA", "CTA", "CCA", "CAA", "CGA", "ATA", "ACA", "AAA",
"AGA", "GTA", "GCA", "GAA", "GGA", "TTG", "TCG", "TAG", "TGG",
"CTG", "CCG", "CAG", "CGG", "ATG", "ACG", "AAG", "AGG", "GTG",
"GCG", "GAG", "GGG"), .Names = c("F", "S", "Y", "C", "L", "P",
"H", "R", "I", "T", "N", "S", "V", "A", "D", "G", "F", "S", "Y",
"C", "L", "P", "H", "R", "I", "T", "N", "S", "V", "A", "D", "G",
"L", "S", "*", "*", "L", "P", "Q", "R", "I", "T", "K", "R", "V",
"A", "E", "G", "L", "S", "*", "W", "L", "P", "Q", "R", "M", "T",
"K", "R", "V", "A", "E", "G"))
```

Here is a link to website that breaks down the code. `http://www.cbs.dtu.dk/courses/27619/codon.html`. First print the contents of the vector stdcode. Notice that, for example, the triplet `TTT` corresponds to Phenylalanine ("F"). The triplet AGG corresponds to Argenine, ("R"). Three of the triplets in the vector correspond to stop codons: `TAA, TAG, TGA`. These "stop codons" are represented by an asterisk. There are two functions in particular that you learned about in the vectors lecture that can help you determine *which* protein corresponds to a given triplet (or what triplet corresponds to which protein).

So given a string of DNA nucleotides, (e.g. `ACGTs`), write a function that can translate it into protein by considering every triplet in the string of DNA from beginning to end. Your function MUST accept collapsed strings, like `ATGTGTCGTG`, or character vectors, like `c("A","G","G","T")`. As an example either of the following strings would be valid.

```
mydna <- "ATTCTTATTGATTAAGCTGA"
```

```
-OR-

mydna <- c("A","T","T","C","T","T","A","T","T","G","A","T","T","A","A","G","C","T","G","A")

The triplets and the corresponding protein, would be:

ATT - I
CTT - L
ATT - I
GAT - D
TAA - *
GCT - A
GA
```

Notice that we have two extra characters at the end of the sequence that we can ignore since they don't form a triplet. So your job is to find a way to march along the input dna string 3 at a time and find out what the corresponding protein would be. You would capture that protein character, (maybe in a vector), and at the end of processing the string, return the vector. A call to transprot could look like:

```
somedna <- "AGTGTGCGTGTGGCAAATCGAT"
transprot(somedna)
[1] "S" "V" "R" "V" "A" "N" "R"

somedna <- c("A","G","C","T","G","C","A","A","T","G")
transprot(somedna)
[1] "S" "C" "N"

transprot(somedna,T)
[1] "SCN"

transprot("ATTCTTATTGATTAAGCTGA")
[1] "I" "L" "I" "D" "*" "A"
```

Use the above examples to check your work. Here is some example pseudo code for this function. This is just a suggestion. There are other ways to approach this problem so feel free to pick whatever methods you feel might work.

```
Get an input vector from the user
If you have been given a collapsed vector then
 expand it
Setup an empty vector that will be used to return the proteins

For the length of the input vector (or some variation thereof)
 Determine a way to get groups of 3 consecutive elements from the input vector
 Figure out what protein is represented by the triplet
 Stash the protein name in the return protein vector

If the user has specifed TRUE for the collapsed argument then
```

```
    collapse protein vector
  Else
    return expanded vector
```

4 Quartile - 30 points

Write a function called "quartile.nist" which computes the requested percentiles of a given vector x. The calling sequence should look like this:

```
quartile.nist(x, probs=c(0.25,0.50,0.75))
```

Table 4: Arguments to quartile.nist function

Argument	Purpose
x	a numeric vector of any size
probs	a vector of desired percentile(s) where ($0 <$ probs < 1)

The default value for probs should be c(.25,.50,.75)). Check for values in probs that are less than or equal to zero or greater than or equal to 1. For hints on how to structure this function see the psuedocode given below in Figure 1. Here are some examples of how this function can be used:

```
x <- c(19,15.5,15.0,20.5,18.3,20.9,11.7,24.3,23.9)
quartile.nist(x)

 25%  50%  75%
15.3 19.0 22.4

quartile.nist(x,c(.40,.60))
  40%  60%
18.3 20.5

set.seed(145)
testx <- rnorm(1000)
quartile.nist(testx)
    25%     50%     75%
-0.5810  0.0756  0.7543
```

4.1 Background

Suppose that we have a data vector x_1, x_2, . . . , x_n. The pth percentile is conceptually meant to be a value Q_p such that p proportion of the observations fall below Q_p. There are different ways to compute the percentiles / quartiles. Here we will use the "n+1" method. Consider the data set:

```
Xn <- 19.0 15.5 15.0 20.5 18.3 20.9 11.7 24.3 23.9

x <- c(19,15.5,15.0,20.5,18.3,20.9,11.7,24.3,23.9)

Here n = 9. We will first need to order these values:

sortedx <- 11.7 15.0 15.5 18.3 19.0 20.5 20.9 23.9 24.3
```

To compute the 20th percentile of the data we calculate Q_{20} as follows:

```
(n + 1) * p = (9 + 1) *.20 = 2
```

So the value representing the 20th percentile is sortedx[2] which is 15. Thus for computed Q values that wind up being an integer then you simply use the Q value to index into the sorted vector and you are done. Now, let's compute the 33rd percentile. Here we must use interpolation which will require some adjustment to our approach.

```
Q = (n + 1)*p = (9 + 1)*0.33 = 3.3
```

Note that we have a number with a fractional part. HINT: It will be very useful for you to code up some logic to break up this number into its whole component (3) and its fractional component (0.3). You will need these values early on in your function to do comparisons. So here we take the third observation (the whole part of Q) of the SORTED vector and add to it 0.3 (the fractional part of Q) times the distance between it and the next highest (4th) observation. So:

```
Q33 = sortedx[3] + 0.3*(sortedx[4]-sortedx[3]) is 16.3

quartile.nist(x,.33)
   33%
16.34
```

So that is the approach you implement for computed Q values that have a fractional part. Well almost. There are "boundary conditions" that we must consider. For low or high values of prob (like less than .10 or greater than .90) you need to pay attention to the computed Q value. Consider the case where probs is 0.95. Q = (9 + 1)*.95 = 9.5. Since there are only 9 elements in the original vector you can't interpolate it since there are no other elements above element 9. So in this case you would subtract 1 from the whole part (giving 8) and then apply the given interpolation formula:

```
Q95 = sortedx[8] + 0.5*(sortedx[9] - sortedx[8]) = 24.1

quartile.nist(x,.95)
  95%
24.1
```

Also consider the case where probs is 0.05. Q = (9 + 1)*.05 = 0.5. Note that there is no *zero-th* element of the vector. So we add 1 to Q and apply the given interpolation formula. So Q is 0.5 but we add 1 to it to get 1.5 and then apply the formula:

```
Q05 = sortedx[1] + 0.5*(sortedx[2] - sortedx[1]) = 13.3

quartile.nist(x,.05)
    5%
13.35
```

Figure 1: Possible Pseudocode for nist.quartile function

```
determine the length, (n), of the input vector x
sort the vector x for later use
setup an empty vector that will be used to return the requested quartiles

for each value in the probabilities vector do
    compute Q using the (n+1)*p formula
    split up Q into its whole and fractional parts
    if Q has a non zero fractional part
        implement some logic to check for low and hi boundary conditions;
        apply interpolation formula;
    else
        Q is a whole number with a zero fractional part;
        Use whole part of Q to index into the sorted array;
    end
    append the computed Q to the retrun vector
end
name the return vector elements to reflect the requested percentiles (e.g. 25%, 50%, etc)
```