Stat243: Problem Set 4, Due Wednesday Oct. 13

October 8, 2021

This covers the second half of Unit 5.

It's due **as PDF submitted to Gradescope** and with all materials submitted via GitHub at 10 am on Oct. 13.

Comments:

1. The formatting requirements are the same as previous problem sets.
2. Please note my comments in the syllabus about when to ask for help and about working together. In particular, **please give the names of any other students that you worked with on the problem set and indicate in comments any ideas or code you borrowed from another student.**
3. Note that for problems 4 and 5, you will be inspecting the addresses of objects and determining the copying that takes place. The results when knitting and results in RStudio may differ from results when running R outside of RStudio. If you see inconsistencies, the correct answer is obtained when running R outside of RStudio. So if there are inconsistencies, you may need to include a screenshot or copy the output from the R console into your solution as comments in your code.

Problems

1. When I run the code *plot(x,y)*, the function *range()* is called when making the axis limits. Suppose I create a function called *range()* in my R session, as follows and then make my plot.

   ```r
   n <- 10
   x <- rnorm(n)
   y <- rnorm(n)

   range <- function(...) cat("Good luck with that.\n")
   range(rnorm(3))
   ## Good luck with that.
   plot(x, y)
   ```

   Explain in detail why can I still make a plot, including exactly where R looks for *range()* and where it finds it. As part of your answer, say what functions are on the call stack at the point that *range()* is called from within *plot()*.

2. Challenge 5 of Section 7.3 of Unit 5.
   The function *oneUpdate()* (and a helper function *ll()*) are code used in maximizing the likelihood function of a statistical model, written by a Statistics grad student. (In the real code, *oneUpdate()* was

called repeatedly in a while loop as part of an iterative optimization to find a maximum likelihood estimator.) The goal is to improve the efficiency of this R code (also available in *ps4.R*). There are a number of improvements that can be made; in particular the code should not need three nested for loops. Consider also whether there are any calculations that are done repeatedly that need only be done once. Report the time it takes before and after your improvements. Compared to this code, I was able to achieve more than a 10-fold speedup. Also, the style of the code can be improved (though you should probably keep the names of objects somewhat similar to what they currently are to assist comparing between the two versions of the code). *ps4prob2.Rda* provides values for A, k, and n.

```r
load('ps4prob2.Rda') # should have A, n, K

ll <- function(Theta, A) {
  sum.ind <- which(A==1, arr.ind=T)
  logLik <- sum(log(Theta[sum.ind])) - sum(Theta)
  return(logLik)
}

oneUpdate <- function(A, n, K, theta.old, thresh = 0.1) {
  theta.old1 <- theta.old
  Theta.old <- theta.old %*% t(theta.old)
  L.old <- ll(Theta.old, A)
  q <- array(0, dim = c(n, n, K))

  for (i in 1:n) {
    for (j in 1:n) {
      for (z in 1:K) {
        if (theta.old[i, z]*theta.old[j, z] == 0){
          q[i, j, z] <- 0
        } else {
          q[i, j, z] <- theta.old[i, z]*theta.old[j, z] /
            Theta.old[i, j]
        }
      }
    }
  }
  theta.new <- theta.old
  for (z in 1:K) {
    theta.new[,z] <- rowSums(A*q[,,z])/sqrt(sum(A*q[,,z]))
  }
  Theta.new <- theta.new %*% t(theta.new)
  L.new <- ll(Theta.new, A)
      converge.check <- abs(L.new - L.old) < thresh
  theta.new <- theta.new/rowSums(theta.new)
  return(list(theta = theta.new, loglik = L.new,
              converged = converge.check))
}

# initialize the parameters at random starting values
```

```r
temp <- matrix(runif(n*K), n, K)
theta.init <- temp/rowSums(temp)

# do single update
out <- oneUpdate(A, n, K, theta.init)
```

3. Challenge 2 of Section 7.3 of Unit 5.
   Suppose you have a mixed membership-style model, $y_i \sim \mathcal{N}(\sum_{j=1}^{m_i} w_{i,j}\mu_{\text{ID}_{i,j}}, \sigma^2)$, for a large number of observations, $i = 1, \ldots, n$, where $\mu = (\mu_1, \ldots, \mu_K)$. In this model, each observation belongs to multiple clusters, where each cluster has its own mean. Please write code to do the following using the objects in *mixedMember.Rda* (see https://www.stat.berkeley.edu/~paciorek/transfer/mixedMember.Rda). There are two test cases:

   - (A) K, the number of components, is large but there are a limited number of components per observation, i.e., $\max_i(m_i)$ is not large.
   - (B) K is small.

   For each case, *mixedMember.Rda* provides a vector containing the μ values, a list of weights, and a list of IDs that map how the weights correspond to the components of the mean vector. For example, for person 1, one might have weights: $w_{1,1} = 0.3$, $w_{1,2} = 0.5$, and $w_{1,3} = 0.2$ and IDs: $\text{ID}_{1,1} = 2$, $\text{ID}_{1,2} = 4$, $\text{ID}_{1,3} = 5$, indicating that we need to calculate $0.3\mu_2 + 0.5\mu_4 + 0.2\mu_5$.

   (a) Write a line of code using *sapply()* that will calculate $\sum_{j=1}^{m_i} w_{i,j}\mu_{\text{ID}_{i,j}}$ for all the observations.

   (b) Set up data objects (in whatever format you choose) and write efficient code that uses those objects to calculate $\sum_{j=1}^{m_i} w_{i,j}\mu_{\text{ID}_{i,j}} \forall\, i$ under case A.

   (c) Set up data objects (in whatever format you choose) and write efficient code that uses those objects to calculate $\sum_{j=1}^{m_i} w_{i,j}\mu_{\text{ID}_{i,j}} \forall\, i$ under case B.

   (d) Compare speed for the two test cases using the three different variants of the code using R's benchmarking tools. For Case A, your code in (b) should give roughly one order of magnitude speed-up compared to your code in (a). For Case B, your code in (c) should give between one and two orders of magnitude speed up relative to your code in (a).

   Note, your code for setting up the data objects in parts (b) and (c) does NOT count toward your computational efficiency. The idea is to get the data in the form you want it, implicitly assuming that this preparation would only be done once, but the calculation of the weighted sum would be done repeatedly. Also, your solution should be able to keep the *muA* and *muB* vectors as they are from *mixedMember.Rda*.
   Hints: Consider how you can fully vectorize the calculation and/or convert the problem to a simple matrix algebra calculation. For both case A and case B you should be able to set up matrices to hold the data and efficiently do the computation on matrices rather than on lists.

4. This question explores memory use and copying with lists.

   (a) Consider a list of numeric vectors, such as:

```
x <- list(rnorm(3), rnorm(3))
```

Modify an element of one of the vectors. Can R make the change in place, without creating a new list or a new vector?

(b) Next, make a copy of the list and determine if there any copy-on-change going on. When a change is made to one of the vectors in one of the lists, is a copy of the entire list made or just of the relevant vector?

(c) Suppose you grow a list in the inefficient way we discussed for vectors in the efficient R tutorial as seen below. Given what you've learned above about the structure of a list, if at the end you have n elements in the list, how many bytes are unnecessarily copied (relative to the situation of you creating an empty list of length n in advance) when running the following code?

```
n <- 10
myList <- list()
for(i in 1:n) {
    myList[[i]] <- rnorm(20)
}
```

5. Section 6.6 of Unit 5 talks about the idea of a *closure* as a function that has data associated with it, as we discussed in class. This builds on the ideas we talked about when discussing scoping. The following code embeds the value of 'x' inside the function as variable named 'data'.

```
x <- rnorm(10)
scaler_constructor <- function(input){
        data <- input
        g <- function(param) return(param * data)
        return(g)
}
scaler <- scaler_constructor(x)
data <- 100
scaler(3)

##  [1]  0.90657024  0.02372059  0.03630277  2.87331032
##  [5] -0.43799319  2.18915054 -1.41444791 -5.41235135
##  [9] -0.43737839 -2.58520131

x <- 100
scaler(3)

##  [1]  0.90657024  0.02372059  0.03630277  2.87331032
##  [5] -0.43799319  2.18915054 -1.41444791 -5.41235135
##  [9] -0.43737839 -2.58520131
```

(a) What is the maximum number of copies of the values represented by x that exist in memory during the execution of *scaler_constructor()?* Why?

(b) Use *serialize()* to generate a sequence of bytes that store the information in the closure. Is the size of the serialized object the size you would expect given your answer to (a)? If not, can you explain what is happening? For this part of the problem, make x a large enough vector that your answer concentrates on the number of bytes involved in the numeric vector rather than in the function itself or any overhead for storing R objects.

(c) It seems unnecessary to have the “data <- input” line, so let’s try the following.

```
x <- rnorm(10)
scaler_constructor <- function(data){
        g <- function(param) return(param * data)
        return(g)
}
scaler <- scaler_constructor(x)
rm(x)
data <- 100
scaler(3)

## Error in scaler(3):  object 'x' not found
```

Explain what is happening and why this doesn’t work to embed a constant data value into the function. Recall our discussion of when function arguments are evaluated.

(d) Can you figure out a way to make the code in part (c) work without explicitly creating a copy of the vector as in the original code? If you do that, how big is the resulting serialized closure?