Stat243: Problem Set 4, Due Wed. October 11

October 2, 2017

This covers the latter part of Unit 4.
It's due **on paper** and submitted via Github at the start of class
Some general guidelines on how to present your problem set solutions:

1. Please use your Rtex/Rnw/Rmd solution from PS1, problem 4 as your template for how to format your solutions (only non-Statistics students are allowed to use R Markdown).
2. As usual, your solution should mix textual description of your solution, code, and example output. And your code should be commented.
3. Your paper submission should be the printout of the PDF produced from your Rtex/Rnw/Rmd file. Your Github submission should include the Rtex/Rnw/Rmd file, any R or bash code files containing chunks that you read into your Rtex/Rnw/Rmd file, and the final PDF.
4. Use functions as much as possible, in particular for any repeated tasks. We will grade in part based on the modularity of your code and your use of functions.
5. Please note my comments in the syllabus about when to ask for help and about working together.
6. Please give the names of any other students that you worked with on the problem set.

Problems

1. In class we discussed the idea of a closure as a function that has data associated with it. We saw that the following code embeds the value of 'x' inside the function.

```
x <- 1:10
f <- function(input){
        data <- input
        g <- function(param) return(param * data)
        return(g)
}
myFun <- f(x)
data <- 100
myFun(3)

##  [1]  3  6  9 12 15 18 21 24 27 30
```

```
x <- 100
myFun(3)

##  [1]  3  6  9 12 15 18 21 24 27 30
```

(a) What is the maximum number of copies that exist of the vector 1:10 during the first execution of *myFun()?* 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 vector of large enough vector that your answer concentrates on the number of bytes involved in the numeric vector not in the function and any overhead for storing R objects.

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

```
x <- 1:10
f <- function(data){
        g <- function(param) return(param * data)
        return(g)
}
myFun <- f(x)
rm(x)
data <- 100
myFun(3)

## Error in myFun(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?

2. This question explores memory use and copying with lists. In answering this question you can ignore what is happening with the list attributes, which are also reported by `.Internal(inspect())`.

(a) Consider a list of vectors. 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) Now make a list of lists. Copy the list. Add an element to the second list. Explain what is copied and what is not copied and what data is shared between the two lists.

(d) Run the following code in a new R session. The result of *.Internal(inspect())* and of *object.size()* conflict with each other. In reality only ~80 MB is being used, as can be seen with *gc()*. Explain why this is the case.

```
gc()
tmp <- list()
x <- rnorm(1e7)
tmp[[1]] <- x
tmp[[2]] <- x
.Internal(inspect(tmp))
object.size(tmp)
gc()
```

3. Challenge 5 of Section 7.3 of Unit 4. The following is real code for maximizing a likelihood function of a statistical model, written by a Statistics grad student. The goal is to improve the efficiency of this R code. 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 a 12-fold speedup. Also, the style of the code I've given you could stand some improvement (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). *unit4prob3.Rda* provides values for A, k, and n.

```
load('ps4prob3.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
temp <- matrix(runif(n*K), n, K)
theta.init <- temp/rowSums(temp)

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


# in the real code, oneUpdate was called repeatedly in a while loop
# as part of an iterative optimization to find a maximum likelihood estimator
```

4. This is a variation on Challenge 8 in Section 7.3 of Unit 4. The goal is to write a function to sample k values without replacement from a population of size n. The inputs to the function are a vector, x, of the values of the population, and k. The *sample()* function in R is quite fast – using *microbenchmark*, I get a time of about 20 microseconds (for 100 evaluations) for $n = 10000$ and $k = 500$ on my computer. Now consider the PIKK and FYKD algorithms as implemented (naively) here:

```
PIKK <- function(x, k) {
    x[sort(runif(length(x)), index.return = TRUE)$ix[1:k]]
}

FYKD <- function(x, k) {
    n <- length(x)
    for(i in 1:n) {
        j = sample(i:n, 1)
        tmp <- x[i]
        x[i] <- x[j]
        x[j] <- tmp
    }
    return(x[1:k])
}


#
```

   (a) Figure out a way to speed up one of the two algorithms (or come up with a different approach) and demonstrate that your solution is faster than the code I present. You should be able to

speed up FYKD by orders of magnitude or PIKK by at least a factor of two (these are based on $n = 10000$ and $k = 500$). Use values of n such as 10000 or 100000. You can focus on the case that k is much less than n, e.g., $k = 100$ or $k = 1000$, but please produce one or more plots that show how the efficiency of your solution compares to the code above as k and n vary. Note your solution should involve coding in R, not writing code in C or another language.
Hint 1: Consider the output you are getting back and think about whether there are unnecessary calculations that are not used in producing that output.
Hint 2: one approach involves rethinking how to generate the random numbers in Algorithm 2 so that they can be done in a vectorized fashion.
Comment: Make sure you notice the units (milliseconds vs. microseconds) reported by *microbenchmark()*.

(b) Extra credit: Find effective ways to speed up both algorithms (or one algorithm and a new approach) and/or get the time down so it takes no longer than 3-4 times the time of *sample()* (I was able to get it down to about 3-fold for $n = 10000$ and $k = 500$).