Stat243: Problem Set 4, Due Wednesday 10/15

October 2, 2014

Comments:

- The file *ps4.R* has the code from this PDF as it will probably be helpful to have the code in a plain text file and also because some of the line breaks in the PDF are not ideal.
- This covers Unit 6.
- It's due at the start of class on 10/15.
- As usual, simply providing the raw code is not enough; make sure to describe how you approached the problem, the steps you took, and output illustrating what your code produces.
- Please note my comments in the syllabus about when to ask for help and about working together.
- As discussed in the syllabus, please turn in (1) a copy on paper, as this makes it easier for us to handle AND (2) an electronic copy through Git following Jarrod's instructions.

Problems

1. Consider this function:

```
f1 <- function(x = {
    y <- 1
    2
}, y = 0) {

    x + y

}

f1()

## [1] 3
```

   Explain fully why the result is 3, describing how the values for *x* and *y* are determined.

2. Explore whether it is more efficient in R to subset a vector based on a vector of indices or to subset based on vector of logical values (booleans). What are the important factor or factors in addition to the length of the vector being subset? In presenting your answer, make a plot or two illustrating how

timing of different methods scales with size. The goal is that your plot(s) provide as complete and easily-understandable an answer to the question as possible. Also, provide some reasoning for why the result you got makes sense.

3. For this question, the answer will differ between R version 3.0.x and R version 3.1.x. Please use R 3.1.x if at all possible. For those of you with SCF accounts, radagast has R 3.1.1; alternatively it's a good idea to upgrade your R on your own machine anyway. The current BCE VM (0.1-4) unfortunately has an older R.

   (a) Consider a dataframe (e.g., you can use *read.csv()* to read in the *cpds.csv* file in the data directory of the class repository). When you change an element of one of the columns in the data frame, what copies are made? Is the entire data frame copied, just that column, or is the element simply replaced without the entire column being copied? You'll want to use `.Internal(inspect(object))` to delve into the structure of the data frame and where in memory different parts of the data frame are stored. Look carefully at the memory addresses of the components of the data frame.

   (b) Compare your answer to what happens with a list containing vectors if you change an element of one of the vectors in the list.

4. Object-oriented programming practice. Consider a discrete random walk in two dimensions. At each step there is a 0.25 probability of moving left, right, up and down. An example of a random walk of three steps would be the first step is to move one unit to the right, the second step is to move one unit up, and the third step is to move one unit back down to the position attained after the first step.

   (a) Write a function that generates such a random walk. The input argument should be the number of steps to be taken. There should be an optional second argument (with a default) specifying whether the user wants the full path of the walk returned or just the final position. If the former, the result should be given in a reasonable format. As practice with defensive programming, your code should check that the input is a valid integer (it should handle zero and negative numbers and non-integers gracefully).

   (b) Now embed your function in object-oriented code that nicely packages up the functionality. You can choose S3, S4, or Reference Classes, but if you already have a fair amount of experience with S3, you should try one of the other two. You should create a class, called *rw*, with a constructor, a *print* method (for which the result should focus on where the final position is and some useful summary measures of the walk), a *plot* method, and a '*[*' operator than gives the position for the ith step. Also, create a replacement method called *start* that translates the origin of the random walk, e.g., `start(myWalk) = c(5, 7)` should move the origin and the entire walk so that it starts at the position *x=5, y=7*.

   (c) Extra credit: Have your function in (a) be vectorized to avoid looping.

5. The following is the code (borrowed from a recently-graduated Statistics grad student) mentioned in Section 3 of Unit 6. The inputs to the algorithm need to be non-negative integers. Basically the code counts up the number of times different values occur in *xvar* and *yvar*. In the real code from the student the *dummyFun()* C function actually did something, but here, I am just creating it as a placeholder so that you can run the code. The point is to examine the R code.

```
library(inline)
# this code is simply a placeholder to demonstrate that I can modify the
# input arguments as desired in C; in reality 'src' would contain
# substantive computations
src <- "\n        tablex[0] = 7;\n"

dummyFun <- cfunction(signature(tablex = "integer", tabley = "integer", xvar =
    yvar = "integer", useline = "integer", n = "integer"), src, convention = "


fastcount <- function(xvar, yvar) {
    nalineX <- is.na(xvar)
    nalineY <- is.na(yvar)
    xvar[nalineX | nalineY] <- 0
    yvar[nalineX | nalineY] <- 0
    useline <- !(nalineX | nalineY)
    tablex <- numeric(max(xvar) + 1)
    tabley <- numeric(max(yvar) + 1)
    stopifnot(length(xvar) == length(yvar))
    res <- dummyFun(tablex = as.integer(tablex), tabley = as.integer(tabley),
        as.integer(xvar), as.integer(yvar), as.integer(useline), as.integer(le
    xuse <- which(res$tablex > 0)
    xnames <- xuse - 1
    resb <- rbind(res$tablex[xuse], res$tabley[xuse])
    colnames(resb) <- xnames
    return(resb)
}
```

(a) Assume for the sake of concreteness that the user passes in two vectors, each of equal size (80 Mb), into the function and that example input vectors can be created with: `sample(c(seq(1, 20, by = 1), NA), n, replace = TRUE)` where `n <- 1e7`. Your job is to report when any new large objects are created in memory (even temporarily) and the amount of memory (in Mb) used at the point in the code that memory use reaches its maximum. You can ignore objects of negligible size relative to the larger objects (e.g., *tablex* and *tabley*). Note that numeric values (i.e, double precision floating point values) each take 8 bytes per value and integers and booleans 4 bytes per value. You can assume that negligible memory is used within the C code.

(b) (Extra credit) Rewrite the function to minimize memory use. You can also note whether there are ways to reduce memory use based on preprocessing before the values are passed into the function. I was able to keep memory use to about 350 Mb at the point that *dummyFun()* is called.

Note that on even an old computer, the memory use in this example with the value of *n* specified would not be something to be concerned about. But if one made each of those vectors 800 Mb or 8 Gb, then these considerations would be quite salient.

6. The following is also real code for maximizing a likelihood function of a statistical model, written by a Statistics grad student. The goal is to improve the effiency of this code. There are a number of improvements that can be made; in particular the code should not need three nested for loops. Report the time it takes before and after your improvements. On the VM the code as given to you takes 148 seconds, while my rewriting of the code takes 1 second. 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).

```
load("ps4prob6.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
```