Stat243: Problem Set 4, Due Wednesday October 10

September 30, 2018

This covers the second part of Unit 4.
It's due **as PDF submitted to bCourses** and submitted via Github at 2 pm on Oct. 10.
Some general guidelines on how to present your problem set solutions:

1. Please use Rmd/Rtex as in previous problem sets.
2. Your solution should not just be code - you should have text describing how you approached the problem and what the various steps were.
3. Your PDF submission should be the PDF produced from your Rmd/Rtex. Your Github submission should include the Rtex/Rmd file, any R code files containing chunks that you read into your Rtex/Rmd file, and the final PDF, all named according to the guidelines in *howtos/submitting-electronically.txt*.
4. Your code should have comments indicating what each function or block of code does, and for any lines of code or code constructs that may be hard to understand, a comment indicating what that code does. You do not need to show exhaustive output but in general you should show short examples of what your code does to demonstrate its functionality.
5. 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.**

Problems

1. Consider the function closure example in on page 65 of Section 6.10 of Unit 4. Explain what is going on in *make_container()* and *bootmeans()*. In particular, when one runs *make_container()* what is returned? What are the various enclosing environments? What happens when one executes *bootmeans()*? In what sense is this a function that "contains" data? How much memory does *bootmeans* use if $n = 1000000$?
2. Suppose we have a matrix in which each row is a vector of probabilities that add to one, and we want to generate a categorical sample based on each row. E.g., the first row might be (0.9, 0.05, 0.05) and the second row might be (0.1, 0.85, .0.5). When we generate the first sample, it is very likely to be a 1 and the second sample is very likely to be a 2. We could do this using a for loop over the rows of the matrix, and the *sample()* function:

```
n <- 100000
p <- 5  ## number of categories
```

```
## efficient vectorized way to generate a random matrix of
## row-normalized probabilities:
tmp <- exp(matrix(rnorm(n*p), nrow = n, ncol = p))
probs <- tmp / rowSums(tmp)

smp <- rep(0, n)

## slow approach: loop by row and use sample()
set.seed(1)
system.time(
    for(i in seq_len(n))
        smp[i] <- sample(p, 1, prob = probs[i, ])
)
```

However that is a lot slower than some other ways we might do it. How can we do it faster? Hint: think about how you could transform the matrix of probabilities such that you can do vectorized operations without using sample() at all. Hint #2: there is a very fast solution that uses a loop (I know, that seems crazy, right?). You can use the test matrix in the code above to compare your solution(s) to my slow approach (and make sure you show the result of benchmarking your solution versus my slow approach on your own computer). My solution has a speedup of 60 times the original loop above - your solution should aim for that much.

3. The following example comes from a problem encountered by a Statistics graduate student for his thesis work. He needed to compute the likelihood for an overdispersed binomial random variable with the following probability mass function (pmf):

$$\begin{aligned} P(Y &= y) = \frac{f(y;n,p,\phi)}{\sum_{k=0}^{n} f(k;n,p,\phi)} \\ f(k;n,p,\phi) &= \binom{n}{k} \frac{k^k(n-k)^{n-k}}{n^n} \left(\frac{n^n}{k^k(n-k)^{n-k}}\right)^{\phi} p^{k\phi}(1-p)^{(n-k)\phi}, \end{aligned}$$

where the denominator serves as a normalizing constant to ensure this is a valid probability mass function. Your job is to write code to evaluate the denominator of $P(Y = y)$. In the graduate student's work, he needs to evaluate $P(Y = y)$ many many times, so efficient calculation of the denominator is important. For our purposes here you can take $p = 0.3$ and $\phi = 0.5$ when you need to actually run your function.

   (a) First, write code to evaluate the denominator using *apply()/lapply()/sapply()*. Make sure to calculate all the terms in $f(k;n,p,\phi)$ on the log scale to avoid numerical issues, before exponentiating and summing. Describe briefly what happens if you don't do the calculation on the log scale. Hint: `?Special` in R will tell you about a number of useful functions. Also, recall that $0^0 = 1$.

   (b) Now write code to do the calculation in a fully vectorized fashion with no loops or *apply()* functions. Using the benchmarking tools discussed in the tutorial on efficient R code, compare the relative timing of (a) and (b) for some different values of n ranging in magnitude from around 10 to around 2000.

   (c) (Extra credit) Extra credit may be given based on whether your code is as fast as my solution. I'd suggest using *Rprof()* to assess the steps in your code for (b) that are using the most time and focusing your efforts on increasing the speed of those parts of the code.

4. 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. Show that only ~80 MB is used and explain why this is the case.

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

5. (Extra credit) Answer the comprehension problem from Section 6.8 of Unit 4, which I repeat here. As background (and as we'll discuss in the simulation unit), random numbers on a computer are actually pseudo-random and are generated from deterministic algorithms that produce numbers that behave as if they are random. *set.seed()* initializes the object *.Random.seed*, and that object determines where in the deterministic sequence we start generating random numbers. Consider this code.

```
set.seed(1)
save(.Random.seed, file = 'tmp.Rda')
rnorm(1)

## [1] -0.6264538

load('tmp.Rda')
rnorm(1)   ## same random number!

## [1] -0.6264538

tmp <- function() {
  load('tmp.Rda')
  print(rnorm(1))
}
tmp()    ## why is this not the same number?

## [1] 0.1836433
```

Why does running *tmp()* not generate the same random number as earlier?