Stat243: Problem Set 4, Due Monday 10/12

September 30, 2015

Comments:

- This covers Unit 4 (Sections 6-8) and Unit 5.
- It's due at the start of class on 10/12, on paper and via Github.
- As usual, your solution should mix textual description of your solution, code, and example output.
- Please note my comments in the syllabus about when to ask for help and about working together.
- Please give the names of any other students that you worked with on the problem set.

Problems

1. As we'll see in the simulation unit, random numbers on a computer are not truly random. They are numbers generated deterministically in a way that they appear random. In fact, the generator is periodic - at some point the sequence of random numbers starts to repeat itself. The *set.seed* function chooses a position in that periodic sequence by setting the value of the *.Random.seed* variable. If you call *set.seed()* with the same input argument you will generate the same sequence of random numbers. When using R's default Mersenne twister random number generator, `.Random.seed[2]` will give information about the position within the periodic sequence. Every time you generate a new random uniform number, that position will increment by one.

   (a) The following code was an attempt by me to save the position of the generator to a file and then to read it back in within a function and be able to start generating random numbers where I left off when I saved the position to the file. This code does not work because of a scoping issue.

   ```
   set.seed(0)
   runif(1)

   ## [1] 0.8966972

   save(.Random.seed, file = 'tmp.Rda')
   runif(1)

   ## [1] 0.2655087

   load('tmp.Rda')
   runif(1)

   ## [1] 0.2655087
   ```

```r
tmp <- function() {
  load('tmp.Rda')
  runif(1)
}
tmp()

## [1] 0.3721239
```

Using R's debugging tools, determine what the problem is. Your written solution should describe how you used the debugging tools and should show a printout of the *tmp* function that includes any lines of code that you inserted for debugging.

(b) In light of your understanding of R's scoping rules, revise the code so that it works. I.e., when `runif(1)` is called in *tmp()*, it should give the same result as when `runif(1)` was called just after the *save()*.

2. 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.

3. Suppose you have a model, $y_i \sim \mathcal{N}(\sum_{k=1}^{m_i} w_{i,k}\mu_{ID_{i,k}}, \sigma^2)$, for a large number of observations, $i = 1,\ldots,n$. Please write code to do the following using the objects in *mixedMember.Rda* (see the *units* directory of the class repository). 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. E.g. 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 $ID_{1,1} = 2$, $ID_{1,2} = 22$, $ID_{1,3} = 31$, indicating that we need to calculate $w_{1,1}\mu_2 + w_{1,2}\mu_{22} + w_{1,3}\mu_{31}$.

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

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

(c) Set up data objects (in whatever format you choose) and write efficient code that uses those objects to calculate $\sum_{k=1}^{m_i} w_{i,k}\mu_{ID_{i,k}}$ 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 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.
Hints: (1) we saw a trick for setting up data structures for this kind of scenario in class. (2) Consider how you can fully vectorize the calculation and/or convert the problem to a simple matrix algebra calculation.

4. Consider a linear regression with 1 million observations and 3 covariates/predictors (4 counting the intercept). You can generate a test dataset using *rnorm()*, creating 4 individual vectors, one for y and three for the covariates and fit the model with `lm(y ~ x1 + x2 + x3)`. In your explorations, please use *mem_used()* rather than *gc()* as this seems to give more interpretable results here.

(a) Use the tools we've seen to figure out how much total memory is in use in *lm()* at the point at which *lm.fit()* is called. How much additional memory use is this compared to the memory used in the global environment to store the observations and covariates?

(b) Figure out where the major uses of memory are in *lm()* leading up to the call to *lm.fit()*. You can ignore any objects that are no bigger than 10% of the size of the vector of observations. Why are some of the vectors and matrices larger than 8 bytes per number? I'm not expecting a perfect answer here – R carries out some fairly involved manipulations to reduce memory – but you should be able to get the big picture of where additional memory gets used. If there seem to be contradictions in what you are seeing, you can report those contradictions without figuring out exactly what is going on.

(c) If you were rewriting the *lm()* function to not use as much memory, what could you do to reduce memory use before calling *lm.fit()*? Motivational note: obviously with 1 million observations and three covariates, memory use is not a problem on a modern machine, but consider if you had hundreds of millions of observations.

(d) Extra credit may be given for particularly detailed explanations of how R is using memory leading up to the call to *lm.fit()*. Extensive use of *.Internal(inspect())* is likely to be needed here.