Stat243: Problem Set 4, Due Monday Oct. 12, 5 pm

October 6, 2020

This covers the second half of Unit 5.
It's due **as PDF submitted to Gradescope** and submitted via GitHub at 5 pm on Oct. 12.
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.**

Problems

1. If I want to compute the trace of a matrix, $A = XY$, where both X and Y are $n \times n$ and where the trace is $\sum_{i=1}^{n} A_{ii}$, a naive implementation is `sum(diag(X%*%Y))`.

   (a) What is the computational complexity of that naive implementation: $O(n)$, $O(n^2)$ or $O(n^3)$?

   (b) Why is that naive implementation inefficient?

   (c) How could you (much) more efficiently compute the trace in R using vectorized operations on the matrices (please provide R code and do not use *apply()*)? What computational complexity is your solution?

2. First, some background on random number generation, which we'll discuss in great detail in the simulation unit. Random numbers on a computer are actually pseudo-random and are generated from deterministic algorithms that produce a fixed sequence of 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. If we set the seed to be a given number, we can repeatedly generate the same "random" numbers, as seen by running the code here.

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

## [1] 0.2655087

set.seed(1)
runif(1)   ## same random number!
```

```
## [1] 0.2655087

runif(1)   ## different

## [1] 0.3721239

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

## [1] 0.2655087

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

## [1] 0.3721239
```

Why does running *myfun()* not generate the same random number as earlier? How can I change the code so that it works? You are only allowed to modify the contents of *myfun()*. Note: this is basically the comprehension problem from Section 6.6 of Unit 5.

3. The following example comes from a problem encountered by a Statistics graduate student for her thesis work. She 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, she 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 (as we'll discuss in Unit 6), 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. Note that if you try to profile the code, it may not be informative because the code may be byte-compiled.

4. This question explores memory use and copying with character vectors.

(a) Consider the following character vector. Modify one of the elements. Can R make the change in place? (Be careful, there are two aspects to this, so it's a bit more complicated than simply replacing an element in a numeric vector.) Also note that you may need to copy-paste output from R or RStudio as compiling the PDF may change the answer.

```
vec <- c("hello there", "better luck next time", "that's not clear")
```

(b) Now consider this vector: `vec <- c(rep(’hello friend’, 1e6))`. Given each character should take 1 byte, this would seemingly use 12 million bytes. How much memory is being used? Explain what is happening and account for all major uses of memory.

(c) Compare the size of the string ’hello’ with that of a single string of length 1 million characters (i.e., one where *nchar()* returns 1 million). Does each character take up 1 byte? What does this comparison suggest about short strings?