STAT243 PS4

Zhaochen Gao

10/4/2018

Problem 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?

Ans: (1) make_container takes one input variable n and creates a double-precision vector (called x) of the specified length (which is n) with each element equal to 0. It also has a nested function which check if the input value is null. If so it will return the transformed n. Otherwise, it will let i-th position of transformed n to be the input value and add 1 to the counter i. When we run make_container, the nested function will be returned. (2) The enclosing environment of bootmeans, n, i, x is make_container(). The enclosing environment of make_container() is the global environment. (3) When we executes bootmeans(), it will return x. If there's input argument within make_container, it will draw 100 numbers with replacement from the faithful data for 100 times, take the average every time, and place the results in x in the make_container() environment. (4) Actually bootmeans() will not make a copy of data, if we use gc() to look at the memory currently used and max used, we can see after calling bootmeans, the memory used doesn't change. So it just pass in the data, draw numbers and take the average. (5) If n=1000000, it will use 28 megabytes.

Problem 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,]))
```

```
##    user  system elapsed
##   0.451   0.038   0.492
```

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.

```
set.seed(1)
#calculate the cumulative sum for each row of probs matrix
mat_csum = t(apply(probs, 1, cumsum))
#generate n random numbers followed by Uniform Distribution with min 0 and max 1
rdm_num = runif(n, min = 0, max = 1)

#first method: vectorized operations
method1 = function(){
  check_mat = apply(mat_csum, 2, function(x){x > rdm_num})
  smp1 = apply(check_mat, 1, function(x){which(x==1)[1]})
}

system.time(method1())
```

```
##    user  system elapsed
##   0.190   0.006   0.197
```

```
#second method:
method2 = function(){
  #create an empty list for the categorical sample
  smp2 = rep(0,n)
  for (i in 1:p){
    #loop through each element at each row
    #find the first position that the cumulative sum is greater than the random number
    smp2[mat_csum[,i]>rdm_num & smp2==0] = i
  }
}

system.time(method2())
```

```
##    user  system elapsed
##   0.010   0.002   0.013
```

Problem 3

(a) First, write code to evaluate the denominator using apply()/lapply()/sapply(). Make sure to calculate all the terms in f(k; n; p; theta) 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.

```
denominator <- function(n){
  p = 0.3
  theta = 0.5
  #temporary variables made to increase the speed of codes
```

```
  tmp_1 = n*log(n)
  tmp_2 = theta*log(p)
  tmp_3 = theta*log(1-p)
  #function to calculate separate pdf for each k
  sep_pdf <- function(k){
    log_f_k = lchoose(n,k)+(1-theta)*((n-k)*log(n-k)-tmp_1+k*log(k))+k*tmp_2+(n-k)*tmp_3
  }
  #when k = 0
  result_0 = (1-p)^(n*theta)
  #when k = n
  result_n = p^(n*theta)
  result<-sum(exp(unlist(lapply(seq(1,n-1,1),sep_pdf))),result_0,result_n)
  return(result)
}
```

(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.

```
vectorized_de <- function(n){
  k = seq(0,n,1)
  p = 0.3
  theta = 0.5
  #temporary variables made to increase the speed of codes
  tmp_1 = n*log(n)
  tmp_2 = theta*log(p)
  tmp_3 = theta*log(1-p)
  result = exp(lchoose(n,k)+(1-theta)*((n-k)*log(n-k)-tmp_1+k*log(k))+k*tmp_2+(n-k)*tmp_3)
  #when k = 0
  result[1]= (1-p)^(n*theta)
  #when k = n
  result[n+1]= p^(n*theta)
  return(sum(result))
}
```

```
library(rbenchmark)
time_test <- function(y){
  rbind(benchmark(denominator(y), vectorized_de(y),
replications = 10, columns=c('test', 'elapsed', 'replications')))
}
rbind(time_test(20),time_test(100),time_test(500),time_test(1000),time_test(2000))
```

```
##                test elapsed replications
## 1    denominator(y)   0.000           10
## 2  vectorized_de(y)   0.000           10
## 3    denominator(y)   0.002           10
## 4  vectorized_de(y)   0.000           10
## 5    denominator(y)   0.007           10
## 6  vectorized_de(y)   0.001           10
## 7    denominator(y)   0.013           10
## 8  vectorized_de(y)   0.001           10
## 9    denominator(y)   0.023           10
```

```
## 10 vectorized_de(y)   0.003           10
```

Ans: According to the results, vectorized way to calculate the denominator is much faster.

(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.

```
library(proftools)
library(fields)
Rprof(filename = 'tmp.out', interval=0.00005, line.profiling=TRUE)
denominator(2000)
Rprof(NULL)
summaryRprof('tmp.out')
```

Ans: When I use Rprof() to assess the steps in my codes, lappy is the function that takes the most time for the function evaluating the denominator using apply()/lapply()/sapply(). So I simplify the calculation of probability density function of f by combining log terms. For vectorized way, I also simplify the calculation of probability density function. I also extract the parts like n*log(n) which remain the same over all of the k and calculate them at first.

Problem 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?

```
m = list(1,'abc',5,6)
.Internal(inspect(m))
m[1]<-2
.Internal(inspect(m))
```

Ans: R cannot make the change in place without creating a new list or a new vector. As we can see, the memory location of the list changes after we modify the first vector. The memory location which stores first vector also changes.

(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?

```
n = m
.Internal(inspect(n))
n[1] <- 3
.Internal(inspect(n))
```

Ans: When we make a copy without modification, the memory location of the copy n remains the same as original list m. There's no copy made. When a change is made to one of the vectors in one of the lists, a copy of the entire list is made.

(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.

```
l = list(list(12,'abc'),list('def',46))
.Internal(inspect(l))
x = l
.Internal(inspect(x))
x = c(x, 2)
.Internal(inspect(x))
```

Ans: When we copy the list, we actually do not make a copy. Both x and l are pointing to the same memory address. When we add an element to the second list which is x, we make a copy of x, add one new element and then change the label of new memory address to be x. When R store the l and x as a list, it actually create a list of memory address of each element. Since the first two elements in l and x are the same, so they shared the same memory location data for the first two elements.

(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))
```

```
## @7fccc6c655c8 19 VECSXP g0c2 [NAM(1)] (len=2, tl=0)
##   @1139a6000 14 REALSXP g0c7 [NAM(3)] (len=10000000, tl=0) 0.525891,-0.487544,1.13825,1.21513,-0.4248
##   @1139a6000 14 REALSXP g0c7 [NAM(3)] (len=10000000, tl=0) 0.525891,-0.487544,1.13825,1.21513,-0.4248
```

```
object.size(tmp)
```

```
## 160000160 bytes
```

```
library(pryr)
object_size(tmp)
```

```
## 80 MB
```

```
.Internal(inspect(x))
```

```
## @1139a6000 14 REALSXP g0c7 [NAM(3)] (len=10000000, tl=0) 0.525891,-0.487544,1.13825,1.21513,-0.42483
```

```
object_size(x)
```

```
## 80 MB
```

Ans: When we use object_size to check the size of tmp, we can see it's 80 MB. And the output of .Internal(inspect(tmp)) shows that, the memory address of first element and second element are the same, so they are pointing to the same location, which is the location of x. Therefore, the tmp has the same size of x, which is 80 MB.

Problem 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

```
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()
```

Ans: Setting a seed starts you at a particular known starting point in a long list of pseudorandom numbers. And set.seed works in global environment. In order to generate same number, we have to execute set.seed(1) every time before we run rnorm(1). When we try to load tmp.Rda, the enclosing environment is function tmp(). We actually try to load the .Random.Seed into a local environment which hasn't set the seed to be 1. In order to let tmp() generate same result, you have to add code set.seed(1) within the function tmp() before line print(rnorm(1)).