Stat243 PS4

Meng Wang, SID: 21706745

October 10, 2015

1 Problem 1

1.1 (a)

At the first beginning, we set the seed is 0 by using "set.seed(0)" function. Afte we generate a random number "runif(1)", the seed will be reset. We could see this change by checking ".Random.seed[2]".

```
set.seed(0)
print(.Random.seed[2])

## [1] 624

runif(1)

## [1] 0.8966972

print(.Random.seed[2])

## [1] 1
```

Later when we save the ".Random.seed" to file "tmp.Rda" and read it back again, we are basically resetting the ".Random.seed" in the global environment. Here is how "Random.seed" changes with each line of the code.

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

## [1] 0.2655087

 print(.Random.seed[2])

## [1] 2

load('tmp.Rda')
 print(.Random.seed[2])

## [1] 1

runif(1)

## [1] 0.2655087

 print(.Random.seed[2])

## [1] 2
```

As we can see, the position in the periodic sequence is set to be "2" after another "runif(1)" and is reset "1" by loading data. All of these changes happen in the Global environment.
However, inside function "tmp", when we load in the data, the function will create a local variable ".Random.seed" to save the data "tmp.data", and it is only valid inside this function. When function "runif(1)" is called inside function "tmp", the actually variable ".Random.seed" used by "rubif(1)" is the global one instead of the local one. Thus, the position locally inside the function is "1", while globally the position is "2" before function "tmp" is called and is set to "3" after the function is called. This can be validated by print out the position inside the function and after the call of the function.

```
 # print out the global varaible before and after the function call
 print('Before tmp is called, the global variable .Random.seed is' )
```

```
## [1] "Before tmp is called, the global variable .Random.seed is"
```

```
 print(.Random.seed[2])
```

```
## [1] 2
```

```
 tmp <- function(){
  load('tmp.Rda')
  runif(1)
  ## print the local varaible .Random.seed
  print('In tmp function, the local variable .Random.seed is' )
  print(.Random.seed[2])
 }
 tmp()
```

```
## [1] "In tmp function, the local variable .Random.seed is"
## [1] 1
```

```
 print('After tmp is called, the global variable .Random.seed is' )
```

```
## [1] "After tmp is called, the global variable .Random.seed is"
```

```
 print(.Random.seed[2])
```

```
## [1] 3
```

1.2 (b)

This problem can be easily solved by assigning "tmp.Rda" to the global variable instead of the local variable. Function "load" has the "env" argument to specify the environment. Here is the change and result

```
############## Problem 1(b) ########################
rm(list = ls())
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

tmp <- function(){
  load('tmp.Rda',envir = parent.frame())
  runif(1)
}
 ## check the result
tmp()

## [1] 0.2655087
```

2 Problem 2

2.1 (a)

The reason why we need to evaluate the result in log scale is because some term in $f(k; n, p, \phi)$ can be very big, for example n^n with $n = 200$

```
 200^200

## [1] Inf
```

The code for the denominator is very straight forward. First write the density function $f(k; n, p, \phi)$ and use "sapply" to go over the number k from 0 to n. The result should be the sum of the sapply function.

```
# compute the demoninator
comp_denom1 <- function(num){
  f <- function(k, n = num, p = 0.3, phi = 0.5){
    if(k == 0){
      result = n*phi*log(1-p)
    }else if(k == n){
      result = n*phi*log(p)
    }else{
      result = lgamma(n+1) - lgamma(k+1) - lgamma(n-k+1)+
        k*(1-phi)*log(k) +(n-k)*(1-phi)*log(n-k) - n*(1-phi)*log(n) +
        k*phi*log(p) + (n-k)*phi*log(1-p)
    }
    return(result)
  }
  return(sum(sapply(0:num, f)))
}
comp_denom1(100)

## [1] -1664.196
```

2.2 (b)

For function f inside the $comp_denom1$, the only part that can not take argument k as list is the if-else statement. We shall get rid of the if-else statement and take care of the special case when $k = 0, n$ outside the function. The rest part can be vectorized directly. Here is the function rewrite,

```r
# compute the demoninator
comp_denom2 <- function(num, p = 0.3, phi = 0.5){
  f2 <- function(k, n = num, p = 0.3, phi = 0.5){
    result = lgamma(n+1) - lgamma(k+1) - lgamma(n-k+1)+
      k*(1-phi)*log(k) +(n-k)*(1-phi)*log(n-k) - n*(1-phi)*log(n) +
      k*phi*log(p) + (n-k)*phi*log(1-p)
    return(result)
  }
  return(num*phi*log((1-p)*p) + sum(f2(1:(num-1))))
}
comp_denom2(100)

## [1] -1664.196
```

I use the function *benchmark* in library "rbenchmark". The function *benchmark* will replicate the test function 100 times by default. Since when n is small, function *system.time*() will simply give us 0. Here are some test cases for the time consumed bt using *sapply* and vertorization

```r
 #test for time
library(rbenchmark)
benchmark(comp_denom1(10));benchmark(comp_denom2(10))

##              test replications elapsed relative user.self sys.self
## 1 comp_denom1(10)          100    0.01        1         0        0
##   user.child sys.child
## 1          0         0
##              test replications elapsed relative user.self sys.self
## 1 comp_denom2(10)          100   0.002        1         0        0
##   user.child sys.child
## 1          0         0

benchmark(comp_denom1(100));benchmark(comp_denom2(100))

##               test replications elapsed relative user.self sys.self
## 1 comp_denom1(100)          100   0.081        1         0        0
##   user.child sys.child
## 1          0         0
##               test replications elapsed relative user.self sys.self
## 1 comp_denom2(100)          100   0.004        1         0        0
##   user.child sys.child
## 1          0         0

benchmark(comp_denom1(1000));benchmark(comp_denom2(1000))

##                test replications elapsed relative user.self sys.self
## 1 comp_denom1(1000)          100   0.781        1     1.527        0
##   user.child sys.child
## 1          0         0
##                test replications elapsed relative user.self sys.self
## 1 comp_denom2(1000)          100   0.025        1         0        0
##   user.child sys.child
## 1          0         0

benchmark(comp_denom1(2000));benchmark(comp_denom2(2000))
```

```
##                test replications elapsed relative user.self sys.self
## 1 comp_denom1(2000)          100   1.398        1     1.563        0
##   user.child sys.child
## 1          0         0
##                test replications elapsed relative user.self sys.self
## 1 comp_denom2(2000)          100   0.052        1         0        0
##   user.child sys.child
## 1          0         0
```

As we could see, using function *sapply*() is approximately 5 times slower for $n = 10$, 18 times slower for $n = 100$ and 30 times slower when $n = 1000, 2000$ (the time may change on different machine).

3 Problem 3

3.1 (a)

This part is straight forward. We load the data first, use *sapply* to each element of list $wgtsA$ and $IDsA$. We need to use *unlist* to each of the element of the matrix.

```
# load data
load("mixedMember.Rda")
# pick up a i and a Case, we use Case A as example
comp_oneElementA <- function(i){
 return(sum(unlist(wgtsA[i])*muA[unlist(IDsA[i])]))
}
#sapply(1:length(IDsA), comp_oneElementA)
comp_oneElementB <- function(i){
 return(sum(unlist(wgtsB[i])*muA[unlist(IDsB[i])]))
}
#sapply(1:length(IDsB), comp_oneElementB)
```

3.2 (b)(c)

We vectorize the computation and use the matrix computation. It shall be very fast. The problem I encountered is that since the matrix for $w_{i,k}$ and $\mu_{ID_{i,k}}$ is very big ($10000/ \times 8$ each). A full matrix multiply requires huge memory.

```
# vectorize the code and use matrix computation
# Case A
num_obs_A <- length(wgtsA)
num_m_A <- nrow(do.call(cbind, wgtsA))
# create the matrix
matrix_w_A <- matrix(0, nrow = num_obs_A, ncol = num_m_A)
matrix_mu_A <- matrix(0, nrow = num_m_A, ncol = num_obs_A)
# get the value for matrix
for (i in 1:num_obs_A){
  mi = length(unlist(wgtsA[i]))
  matrix_w_A[i,1:mi] <- t(as.matrix(unlist(wgtsA[i])))
  matrix_mu_A[1:mi,i] <- as.matrix(muA[unlist(IDsA[i])])
}
# Case B
num_obs_B <- length(wgtsB)
num_m_B <- nrow(do.call(cbind, wgtsB))
```

```r
# create the matrix
matrix_w_B <- matrix(0, nrow = num_obs_B, ncol = num_m_B)
matrix_mu_B <- matrix(0, nrow = num_m_B, ncol = num_obs_B)
# get the value for matrix
for (i in 1:num_obs_B){
  mi = length(unlist(wgtsB[i]))
  matrix_w_B[i,1:mi] <- t(as.matrix(unlist(wgtsB[i])))
  matrix_mu_B[1:mi,i] <- as.matrix(muB[unlist(IDsB[i])])
}
```

3.3 (d)

Since the matrix is too big to compute the multiplication directly, we have to compute it one by one, which will slow down the process.

```r
#Case A
# use sapply
system.time(sapply(1:length(IDsA), comp_oneElementA))
# use matrix
matrix_mul_A <- function(i) return(matrix_w_A[i,]%*%matrix_mu_A[,i])
system.time(sapply(1:num_obs_A, matrix_mul_A))
# Case B
# use sapply
system.time(sapply(1:length(IDsB), comp_oneElementB))
# use matrix
matrix_mul_B <- function(i) return(matrix_w_B[i,]%*%matrix_mu_B[,i])
system.time(sapply(1:num_obs_B, matrix_mul_B))
```

Here is the result for comparison,

```
 #Case A
 > system.time(sapply(1:length(IDsA), comp_oneElementA))
   user  system elapsed
  0.000   0.000   0.493

> system.time(sapply(1:num_obs_A, matrix_mul_A))
   user  system elapsed
  0.000   0.000   0.281

 # Case B
> system.time(sapply(1:length(IDsB), comp_oneElementB))
   user  system elapsed
  0.000   0.000   0.514

> system.time(sapply(1:num_obs_B, matrix_mul_B))
   user  system elapsed
  0.000   0.000   0.294
```

4 Problem 4

4.1 (a)

First we check how many memory are currently using, and then generate the data and check the memory again

```
library(pryr)
mem_used()

## 35.6 MB

# generate the data
y <- rnorm(1e6); x1 <- rnorm(1e6)
x2 <- rnorm(1e6);x3 <- rnorm(1e6)
# check the total memroy use after data generation
mem_used()

## 67.6 MB
```

As we could see, there are around 32MB more memory used. Each of the data we generated is 8Mb, as indicated below.

```
 object.size(y)

## 8000040 bytes
```

Then we could fit the model and check the memory use again

```
# fit the model and check memory again
lsm <-lm(y ~ x1 + x2 + x3)
mem_used()

## 252 MB
```

Now the memory usage is 251 MB, so function $lm.fit$ used 184.1 MB

4.2 (b)

Use the function we defined on class $ls.sizes$, we could check the size of the variables in global environment.

```
> ls.sizes()
object         bytes
lsm        264017120
x1           8000040
x2           8000040
x3           8000040
y            8000040
ls.sizes       27768
```

The memory for lsm is around 264MB. Check the size of the members of lsm one by one, we have

```
> object.size(lsm$residuals)
64000192 bytes
> object.size(lsm$coefficients)
```

```
456 bytes
> object.size(lsm$effects)
16000440 bytes
> object.size(lsm$fitted.values)
64000192 bytes
> object.size(lsm$fitted.assign)
0 bytes
> object.size(lsm$fitted.qr)
0 bytes
> object.size(lsm$fitted.terms)
0 bytes
> object.size(lsm$fitted.model)
0 bytes
> object.size(lsm$fitted.call)
0 bytes
```

As we can see, *residuals* and *values* take 64MB each and $effects$ takes 16MB. Theses three objects take 144MB which is more than half of the memory use of lsm. The other parts does not seem to take much memory at all.

4.3 (c)

Since inside the function, the function will allocate the memory for input varaible $y, x1, x2, x3$ again. We could directly use the references of the input instead of reallocating the memory. This could definitely save us lots of memory usage. Or we could allocate memory for the inputs, but after the fitting process, reuse the memory for the inputs as the place to store the fitted values and residuals.
** R code saved in HW4.R; all codes have been run and tested*