STAT243 Problem Set4

Name: Chih Hui Wang SID: 26955255

October 4, 2015

1. (a) The problem is shown below.

```
#Origin Code
set.seed(0)
runif(1)

## [1] 0.8966972

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

## [1] 0.2655087
```

```
#The same as previous
load('tmp.Rda')
runif(1)

## [1] 0.2655087
```

```
#Should be the same as previous
tmp <- function() {
  load('tmp.Rda')
  runif(1)
}
tmp()

## [1] 0.3721239
```

It is the problem of scoping, the problem can only happen in two function, **load** and **runif**. First, we have to make sure where the function **runif** takes seed. It can only take the random seed from two environment, global and function. Let us take a look the random seed in R when we load the **tmp.Rda.**

```
load('tmp.Rda')
#Random seed after load tmp.Rda
head(.Random.seed)

## [1]         403           1 -2124957944  1654269195 -1877109783  -961256264

runif(1)

## [1] 0.2655087

head(.Random.seed)

## [1]         403           2 -2124957944  1654269195 -1877109783  -961256264
```

Then, let us put the **.Random.seed** inside the function to see what is going on when the function load the **tmp.Rda** and run the function **runif**.

```r
#Test
tmp_test <- function() {
  print(head(.Random.seed))
  load('tmp.Rda')
  print(head(.Random.seed))
  runif(1)
  print(head(.Random.seed))
}
tmp_test()

## [1]         403           2 -2124957944  1654269195 -1877109783  -961256264
## [1]         403           1 -2124957944  1654269195 -1877109783  -961256264
## [1]         403           1 -2124957944  1654269195 -1877109783  -961256264
```

By observing the previous outcomes, we can found that after the function run **load**, the seeds change indeed. However, after the **runif** finish, the random seed do not change at all. While it probably load the seed into the function environment, we still are not sure where exactly the place is the **load** put seed into. Now let us add more commands to print out the seed of the global enviroment(by parent.frame() in the function, the parent environment of the function is global environment), which can prove our assumption.

```r
tmp_test2 <- function(){
  #Before loading
  cat("function: ", head(.Random.seed), "\n")
  cat("global: ", head(parent.frame()$.Random.seed), "\n")

  load('tmp.Rda')
  #After loading
  cat("function: ", head(.Random.seed), "\n")
  cat("global: ", head(parent.frame()$.Random.seed), "\n")

  runif(1)
  #After generating random number
  cat("function: ", head(.Random.seed), "\n")
  cat("global: ", head(parent.frame()$.Random.seed), "\n")
}

tmp_test2()

## function:  403 3 -2124957944 1654269195 -1877109783 -961256264
## global:  403 3 -2124957944 1654269195 -1877109783 -961256264
## function:  403 1 -2124957944 1654269195 -1877109783 -961256264
## global:  403 3 -2124957944 1654269195 -1877109783 -961256264
## function:  403 1 -2124957944 1654269195 -1877109783 -961256264
## global:  403 4 -2124957944 1654269195 -1877109783 -961256264
```

The outcome indicate that the **runif** inside the **tmp** function takes the seed from global environment, not the tmp function environment. However, the **load** function actually put the seed into the function envrionment.

Now, the problem turn out to be that the function **load** do not load the seed to the correct environment, so let's see its argument to check whether there is any parameter we can adjust.

```
args(load)

## function (file, envir = parent.frame(), verbose = FALSE)
## NULL
```

As you can see, there is a argument called **parent.frame()**, which is the parent environment. In this case, the parent environment of the **load** function is **tmp** function environment.

(b) Therefore, the problem is that the **load** function put the seed into the place we do not want it to set. To solve the problem, we can change the enviroment to global environment for loading the seed by following code. Then, the problem is solved.

```
#Correct
tmp_right <- function() {
  load('tmp.Rda', envir=parent.env(environment()))
  runif(1)
}

#Demo1
tmp_right()

## [1] 0.2655087
```

```
#Demo2
tmp_right()

## [1] 0.2655087
```

2. (a) If we do not calculate on the log scale, when n is a large number, let's say 2000, the term n^n will become ∞, which make other calculation fail. Therefore, we should calculate on the log scale and after finishing computation, we take exponential to outcome to transform back to the original scale.

In my code, I seperate the equation into 4 part, calculate them respectively and then take exponential as well as sum the result. When using log scale, we encounter antoher problem. The range of the k is from 0 to n. The term $klogk$ may have some problem because the value inside log may be 0 which again will give us ∞. However, the main problem is the term $(n-k)log(n-k)$. When $n = k$, it will become $0 \times \infty$, which R will output **NaN**. Hence, after I evaulate b, I add a command to set those NaN to 0.

```
#Divided into 4 terms
f <- function(n, p=0.3, phi=0.5){
  compute <- function(n=n, k){
    a <- lchoose(n, k)
    b <- k*log(k) + (n - k)*log(n - k) - n*log(n)
    b[is.na(b)] <- 0
    c <- phi*(-b)
    d <- k*phi*log(p) + (n - k)*phi*log(1 - p)
    result <- exp(a + b + c + d)
  }
  k <- 0:n
  sum(sapply(k, compute, n=n))
}
```

```
#Demo
f(10)

## [1] 1.475851
```

(b) Set the k equal 0 to n, then we can compute the sum in a vectorized way.

```
f_v <- function(n, p=0.3, phi=0.5){
  k <- 0:n
  a <- lchoose(n, k)
  b <- k*log(k) + (n - k)*log(n - k) - n*log(n)
  #Deal with the NaN
  b[is.na(b)] <- 0
  c <- phi*(-b)
  d <- k*phi*log(p) + (n - k)*phi*log(1 - p
)
  sum(exp(a + b + c + d))
}

#Demo, same as previous
f_v(10)

## [1] 1.475851
```

Both of ways perform well in two cases.

```
library(rbenchmark)

## Warning:  package 'rbenchmark' was built under R version 3.2.2

#n=10
benchmark(
  f(10),
  f_v(10),
  replications=10
)

##      test replications elapsed relative user.self sys.self user.child
## 1   f(10)           10    0.02       NA      0.01        0         NA
## 2 f_v(10)           10    0.00       NA      0.00        0         NA
##   sys.child
## 1        NA
## 2        NA

#n=2000
benchmark(
  f(2000),
  f_v(2000),
  replications=10
)

##        test replications elapsed relative user.self sys.self user.child
## 1   f(2000)           10    0.19      9.5      0.19        0         NA
## 2 f_v(2000)           10    0.02      1.0      0.01        0         NA
##   sys.child
## 1        NA
## 2        NA
```

(c) As shown below, large proportion of time spends on computing the **lchoose**, so first we improve the performance of **lchoose**.

```
Rprof("f_v.prof")
invisible(sapply(1:2000, f_v))
Rprof(NULL)
summaryRprof("f_v.prof")$by.self

##             self.time self.pct total.time total.pct
## "lchoose"       0.46    54.76       0.46     54.76
## "FUN"           0.18    21.43       0.84    100.00
## "exp"           0.08     9.52       0.08      9.52
## "-"             0.06     7.14       0.06      7.14
## "*"             0.04     4.76       0.04      4.76
## "is.na"         0.02     2.38       0.02      2.38
```

I tried to replace the **lchoose** by **lfactorial** and find out that it will speed up the calculation.

```
f_v_c1 <- function(n, p=0.3, phi=0.5){
  k <- 0:n
  a <- lfactorial(n) - (lfactorial(k) + lfactorial(n - k))
  b <- k*log(k) + (n - k)*log(n - k) - n*log(n)
  #Deal with the NaN
  b[is.na(b)] <- 0
  c <- phi*(-b)
  d <- k*phi*log(p) + (n - k)*phi*log(1 - p
)
  sum(exp(a + b + c + d))
}
```

```
Rprof("f_v_c1.prof")
invisible(f_v_c1(2000000))
Rprof(NULL)
summaryRprof("f_v_c1.prof")$by.self

##                self.time self.pct total.time total.pct
## "lfactorial"       0.32    43.24       0.34     45.95
## "f_v_c1"           0.20    27.03       0.74    100.00
## "-"                0.08    10.81       0.08     10.81
## "*"                0.04     5.41       0.04      5.41
## ":"                0.04     5.41       0.04      5.41
## "+"                0.04     5.41       0.04      5.41
## "exp"              0.02     2.70       0.02      2.70
```

The most time-consuming part is still **lfactorial**, so I go inside the function line by line. I found out that the a, b and c are actullay all symmetric. Therefore, if we can calculate half of them and use index to produce the other half, then we probably can speed up the speed by around 50% (but we add more commands to distinguish the case for odd and even number so it may not be as well as we expected).

```
f_v_c2 <- function(n, p=0.3, phi=0.5){
  k <- 0:floor((n + 1)/2)
  if(n %% 2){
    j <- c(1:(n %/% 2 + 1), (n %/% 2 + 1):1)
  }else{
    j <- c(1:(n %/% 2 + 1), (n %/% 2):1)
  }
  a <- lfactorial(n) - (lfactorial(k) + lfactorial(n - k))
  b <- k*log(k) + (n - k)*log(n - k) - n*log(n)
  b[is.na(b)] <- 0
  c <- phi*(-b)
  d <-  (0:n)*phi*log(p) + (n - (0:n))*phi*log(1 - p)
  sum(exp(a[j] + b[j] + c[j] + d))
}
```

It turns out that the speed indeed accelerates.

```
Rprof("f_v_c2.prof")
invisible(f_v_c2(2000000))
Rprof(NULL)
summaryRprof("f_v_c2.prof")$by.self

##               self.time self.pct total.time total.pct
## "f_v_c2"           0.20    45.45       0.44    100.00
## "lfactorial"       0.12    27.27       0.14     31.82
## "-"                0.06    13.64       0.06     13.64
## "exp"              0.04     9.09       0.04      9.09
## ":"                0.02     4.55       0.02      4.55
```

3. (a) I use sapply to evaulate all the weighted mean by index across all the observations for A. It can also compute the observations for B in the same way.

```
#Load the data
load("mixedMember.Rda")

#One line code to compute the weighted mean
head(sapply(1:length(wgtsA), function(x) sum(wgtsA[[x]]*muA[IDsA[[x]]])))

## [1] -0.53997057 -0.68233057 -0.40414341 -0.24803496  0.44062079  0.03546354
```

(b) My first strategy is unlist all the weight and ID to a vector. Using ID as a index vector for getting the correct mu, and then I multiple two vectors together to get the all weighted value. Then, I use the **cumsum** function, I can get the sum of all values, which is quite close to the answer while need to more process a little bit more. To get the correct answer, I create a variable **location** which is the position of each sum located. By subsetting by location and use **diff**, we can get the weighted mean for each case.

```
#Strategy 1
wA <- unlist(wgtsA)
idA <- unlist(IDsA)

#length
lA <- sapply(wgtsA, length)
#Get the location of each sum
locationA <- cumsum(lA)

head(diff(c(0, cumsum(wA*muA[idA])[locationA])))

## [1] -0.53997057 -0.68233057 -0.40414341 -0.24803496  0.44062079  0.03546354
```

```
#Case B
wB <- unlist(wgtsB)
idB <- unlist(IDsB)

#length
lB <- sapply(wgtsB, length)
#Get the location of each sum
locationB <- cumsum(lB)

head(diff(c(0, cumsum(wB*muB[idB])[locationB])))

## [1] -0.4496267 -0.3697111 -0.2104093 -0.3426966 -0.3874494  0.6585238
```

(c) Second strategy is that we create a matrix with dimension $100,000 \times lengthmu$. The $lengthmu$ is the number of mean in each case.(i.e. case A is 1000, case B is 10) For each column, we directly put the weights into it and then use matrix product with vector muA or muB to get the results(Dimension: 100,000x10 $\times$ 10x1). This approach will be more appropriate for B because the mean vector for B is much smaller than A.

```
#Strategy 2
#Function to construct weight matrix
strategy2_weight <- function(weight, ID, mu){
  m <- length(weight)
  n <- length(mu)
  #Weight
  weight_matrix <- matrix(0, nrow=m, ncol=n)
  for(i in 1:10000){
    weight_matrix[i, ][ID[[i]]] <- weight[[i]]
  }
  weight_matrix
}

#Case A
S_2_Aw <- strategy2_weight(wgtsA, IDsA, muA)

head(S_2_Aw %*% muA)

##              [,1]
## [1,] -0.53997057
## [2,] -0.68233057
## [3,] -0.40414341
## [4,] -0.24803496
## [5,]  0.44062079
## [6,]  0.03546354
```

```
#Case B
S_2_Bw <- strategy2_weight(wgtsB, IDsB, muB)

head(S_2_Bw %*% muB)

##             [,1]
## [1,] -0.4496267
## [2,] -0.3697111
## [3,] -0.2104093
## [4,] -0.3426966
## [5,] -0.3874494
## [6,]  0.6585238
```

(d) The following is the comparison for two case with three approaches. As the result shown, the first strategy performa better on case A, and the second strategy works faster for case B. Both strategies are faster than sapply method.

```
#CaseA
benchmark(
  sapply=sapply(1:length(wgtsA), function(x) sum(wgtsA[[x]]*muA[IDsA[[x]]])),
  strategy1=diff(c(0, cumsum(wA*muA[idA])[locationA])),
  strategy2=S_2_Aw %*% muA,
  replications=10
)

##        test replications elapsed relative user.self sys.self user.child
## 1    sapply           10    2.62     26.2      2.63        0         NA
## 2 strategy1           10    0.10      1.0      0.09        0         NA
## 3 strategy2           10    2.25     22.5      2.25        0         NA
##   sys.child
## 1        NA
## 2        NA
## 3        NA
```

```
#CaseB
benchmark(
  sapply=sapply(1:length(wgtsB), function(x) sum(wgtsB[[x]]*muB[IDsB[[x]]])),
  strategy1=diff(c(0, cumsum(wB*muB[idB])[locationB])),
  strategy2=S_2_Bw %*% muB,
  replications=10
)

##        test replications elapsed relative user.self sys.self user.child
## 1    sapply           10    2.62   87.333      2.63        0         NA
## 2 strategy1           10    0.10    3.333      0.09        0         NA
## 3 strategy2           10    0.03    1.000      0.03        0         NA
##   sys.child
## 1        NA
## 2        NA
## 3        NA
```

```
rm(list=ls())
```

4. (a) I remove all the variables and functions before addressing the problem. There is total 57.8MB in R which includes 4 vector, y, x1, x2, x3(each are 8MB) and some other things.

```
library(pryr)

## Warning:  package 'pryr' was built under R version 3.2.2

n <- 1000000
y <- rnorm(n); x1 <- rnorm(n); x2 <- rnorm(n); x3 <- rnorm(n)
mem_used()

## 57.4 MB
```

Then, I source the lm functionm **mylm** I wirte (In Github). I add the command **mem_used** before the function call **lm.fit**.

```
source("mylm1.r")
```

The difference between this and the initial memory is around 156MB.

```
fit <- mylm(y ~ x1 + x2 + x3)

## 213 MB

fit

##
## Call:
## mylm(formula = y ~ x1 + x2 + x3)
##
## Coefficients:
## (Intercept)           x1           x2           x3
##  -3.785e-05    2.595e-04   -2.804e-04   -5.901e-04
```

(b) To figure out which objects occupy lots fo memory, I add the **mem_used** function into each line and find out that memory increases when the **lm** function run the following command.

mf <- eval(mf, parent.frame()): The object is a data.frame which contains 4 fields, y, x1, x2, x3

y <- model.response(mf, "numeric"): This is a vector which contains y.

x <- model.matrix(mt, mf, contrasts): This is a matrix which contains intercept, x1 x2 x3

For **mylm2** function, I add **mem_used** before and after the lines of above command. It will output the before and after memory and the difference will be the memory using by creating the object.

```
source("mylm2.r")
```

The first and second difference 32MB is probably the memory of **mf**. It is reasonable because we have 4 vector and each of them are double with length 1000000. Hence the memory of **mf** is around 32MB.

The third and fourth difference is probably the memory of **y**, which is around 16MB. It may have some problem with it because it should only consume 8MB to stroe the vector **y**. After I output it to the global environment, I find out that it is actually a vector with names and actually its memory is 64MB(In **mylm3**), so the rest of the memory may be the memory of character (name).

The fifth and sixth difference is the memory of **x**. It should have the same memory as **mf** because they have same dimensions. However, it turns out that it increases 40MB after create the object. I added a command to output the value to global environment and found out that actually the matrix contain several items. It includes the values(intercept, x1, x2, x3), colnames("intercept", "x1", "x2", "x3"), and the rownames("1", "2", ...,"1000000"). That is why the element of the matrix are stored bigger than 8 bytes per elements.

```
fit2 <- mylm2(y ~ x1 + x2 + x3)

## 246 MB
## 278 MB
## 278 MB
## 294 MB
## 294 MB
## 334 MB
```

```r
source("mylm3.r")
```

I rewrite the function to output those objects into global environment. The following are the actual memories usage for those objects.

```r
fit3 <- mylm3(y ~ x1 + x2 + x3)

## 366 MB
## 398 MB
## 398 MB
## 414 MB
## 414 MB
## 454 MB

#Memory of mf
object_size((mfnew))

## 32 MB

#Memory of y(vector with name)
object_size((ynew))

## 64 MB

#Memory of x(model.matrix)
object_size((xnew))

## 88 MB
```

When I see the above result, I am quite confused. The memory for **mf** is the same as expected. However, when the **lm** function creates the variable y, its memory should increase 64 MB instead of 16MB we saw previous. The same situation happens when it creat the model matrix, **x**. It should add another 88 MB to the memory while the memory only increases by 40MB. It may be that R automatically did some operations to reduce the memory usage.

(c) One way to reduce the memory usage is to change the x(the one created model.matrix), originally 40 MB. If we don't store the name, the size will reduce to 32MB as we calculated.

```r
object_size(xnew)

## 88 MB

head(xnew)

##   (Intercept)         x1          x2         x3
## 1           1 -1.3120350  1.51762833 -1.6734425
## 2           1  2.0852002 -0.04439491  1.1289378
## 3           1  0.2929012  0.79577121  2.1111618
## 4           1  0.4976107  0.73851520 -1.4236423
## 5           1  0.1921341  0.23886450 -0.1855364
## 6           1  1.2927997 -0.18024008 -0.1012110
```

```r
row.names(xnew) <- NULL
head(xnew)

##      (Intercept)         x1          x2         x3
## [1,]           1 -1.3120350  1.51762833 -1.6734425
## [2,]           1  2.0852002 -0.04439491  1.1289378
## [3,]           1  0.2929012  0.79577121  2.1111618
## [4,]           1  0.4976107  0.73851520 -1.4236423
## [5,]           1  0.1921341  0.23886450 -0.1855364
## [6,]           1  1.2927997 -0.18024008 -0.1012110

object_size(xnew)

## 32 MB
```

Another way is to reduce the size of y. We can create a same vector without name by using the command y <- mf[, 1].

```
y <- mfnew[, 1]
head(y)

## [1] -0.3262334  1.3297993  1.2724293  0.4146414 -1.5399500 -0.9285670

head(ynew)

##          1          2          3          4          5          6
## -0.3262334  1.3297993  1.2724293  0.4146414 -1.5399500 -0.9285670
```

```
source("mylm4.r")
```

mylm4 function is that I replace the line $x <- model.matrix(mt, mf, contrasts)$ in **lm** function by $x <- cbind(1, x1, x2, x3)$ and $y <- model.response(mf, \text{“}numeric\text{”})$ by $y <- mf[, 1]$. For **y**, the memory does not increase in this situation(R may do something else to reduce the memory). For **x**, the memory indeed increases only 32 MB when it run through the code. The total memory spending 64MB before calling the **lm.fit** function. The difference is between the original 156MB an 64MB is 92MB.

```
fit4 <- mylm4(y ~ x1 + x2 + x3)

## 518 MB
## 550 MB
## 550 MB
## 550 MB
## 550 MB
## 582 MB

fit4

##
## Call:
## mylm4(formula = y ~ x1 + x2 + x3)
##
## Coefficients:
##                      x1          x2          x3
## -3.785e-05   2.595e-04  -2.804e-04  -5.901e-04
```