Stat243: Problem Set 4

Sicun Huang

October 12, 2015

1.

(a) Insert browser() before and after loading tmp.Rda to examine effects of the load command. Before tmp.Rda is loaded, call environment(tmp) and notice that the function is defined on the global enviroment. Do ls(environment(tmp)) and see that the only object in the global environment is tmp the function. After the load command, call ls(environment(tmp)) again and realize that the tmp.Rda file was not loaded into the global environment but the local environmnet of the tmp function.

```
set.seed(0)
save(.Random.seed, file = 'tmp.Rda')

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

(b) Add the second argument "env = .GlobalEnv" when loading tmp.Rda to ensure that it gets loaded into the global environment.

```
tmp <- function() {
  load('tmp.Rda', env = .GlobalEnv)
  runif(1)
}
tmp()

## [1] 0.8966972

tmp()

## [1] 0.8966972
```

2.

(a) Write function to calculate the denominator. Note that we have to compute each term on log scale before exponentiating and summing because the range of numbers a computer can store is limited; calculating on log scale will swap multiplication of many small numbers for addition of such numbers, which helps avoid underflowing.

```
sumDenom <- function( n, p, phi ){
  logDenom <- function( k ){
    #special case k=0, returns NaN if computed using the generic formula
    if( k==0 ){
```

```
      exp( (n*phi)*log(1-p) )
    }
    #special case k=n, returns NaN if computed using the generic formula
    else if( k==n ){
      exp( (n*phi)*log(p) )
    }
    else{
      exp( lchoose(n,k)+k*log(k)+(n-k)*log(n-k)-n*log(n)+phi*(n*log(n)-k*log(k)-(n-k)*
          log(n-k))+(k*phi)*log(p)+((n-k)*phi)*log(1-p) )
    }
  }
  return( sum( sapply(0:n, logDenom) ) )
}

sumDenom(10,0.3,0.5)

## [1] 1.475851
```

(b) Vectorize the code in (a).

```
sumDenomVec <- function( n, p, phi ){
  k <- 0:n
  denom <- exp( lchoose(n,k)+k*log(k)+(n-k)*log(n-k)-n*log(n)+phi*(n*log(n)-k*log(k)-(n-k)*
              log(n-k))+(k*phi)*log(p)+((n-k)*phi)*log(1-p) )
  #substitute in results for special cases when k=0 and k=n
  denom[1] <- exp( (n*phi)*log(1-p) )
  denom[n+1] <- exp( (n*phi)*log(p) )
  return( sum(denom) )
}

sumDenomVec(10,0.3,0.5)

## [1] 1.475851

system.time( sumDenom(20,0.3,0.5) )

##    user  system elapsed
##       0       0       0

system.time( sumDenomVec(20,0.3,0.5) )

##    user  system elapsed
##       0       0       0

system.time( sumDenom(200,0.3,0.5) )

##    user  system elapsed
##   0.002   0.000   0.001

system.time( sumDenomVec(200,0.3,0.5) )

##    user  system elapsed
##       0       0       0

system.time( sumDenom(2000,0.3,0.5) )
```

```
##    user  system elapsed
##   0.012   0.001   0.012

system.time( sumDenomVec(2000,0.3,0.5) )

##    user  system elapsed
##   0.000   0.000   0.001
```

3.

(a) Calculate the weighted sum using sapply.

```
load("/Users/Sici/Documents/Cal/stat243/units/mixedMember.Rda")
wgtSumA <- sapply(1:100000, function(x){ muA[ IDsA[[x]] ]%*%wgtsA[[x]] })
head(wgtSumA)

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

wgtSumB <- sapply(1:100000, function(x){ wgtsB[[x]]%*%muB[ IDsB[[x]] ] })
head(wgtSumB)

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

(b) Set up matrices with columns muA[IDsA[[i]]] and wgtsA[[i]] respectively. Multiply together the corresponding entries and compute the sums of the columns to get a vector of the weighted sums for all observations.

```
#determine number of rows for matrices
maxLengthA <- max( sapply(IDsA, length) )
#set up mu matrix with columns being the appropriate mu's for each individual
muIdA <- matrix( 0, nrow=maxLengthA, ncol=100000 )
for( i in 1:ncol(muIdA) ){
  muIdA[,i] <- c( muA[ IDsA[[i]] ], rep( 0, maxLengthA-length(IDsA[[i]]) ) )
}
#construct matrix weightA from list wgtsA; each column is one element of the list
weightA <- matrix( 0, nrow=maxLengthA, ncol=100000 )
for( i in 1:ncol(weightA) ){
  weightA[,i] <- c( wgtsA[[i]], rep( 0, maxLengthA-length(wgtsA[[i]]) ) )
}

weightSumA <- colSums( weightA*muIdA )
head(weightSumA)

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

(c) Similar to (b), set up matrices with columns muB[IDsB[[i]]] and wgtsB[[i]] respectively. Multiply together the corresponding entries and compute the sums of the columns to get a vector of the weighted sums for all observations.

```
maxLengthB <- max( sapply(IDsB, length) )
muIdB <- matrix( 0, nrow=maxLengthB, ncol=100000 )
for( i in 1:ncol(muIdB) ){
  muIdB[,i] <- c( muB[ IDsB[[i]] ], rep( 0, maxLengthB-length(IDsB[[i]]) ) )
}
weightB <- matrix( 0, nrow=maxLengthB, ncol=100000 )
for( i in 1:ncol(weightB) ){
```

```
  weightB[,i] <- c( wgtsB[[i]], rep( 0, maxLengthB-length(wgtsB[[i]]) ) )
}

weightSumB <- colSums( weightB*muIdB )
head(weightSumB)

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

(d) Compare speed of code.

```
#case A sapply approach
system.time( sapply(1:100000, function(x){ muA[ IDsA[[x]] ]%*%wgtsA[[x]] }) )

##    user  system elapsed
##   0.176   0.003   0.180

#case A data object approach
system.time( colSums( weightA*muIdA ) )

##    user  system elapsed
##   0.003   0.000   0.003

#case B sapply approach
system.time(sapply(1:100000, function(x){ wgtsB[[x]]%*%muB[IDsB[[x]]] }))

##    user  system elapsed
##   0.180   0.002   0.183

#case B data object approach
system.time( colSums( weightB*muIdB ) )

##    user  system elapsed
##   0.004   0.001   0.004
```

4.

(a) Run the following code in plain r to avoid added bulk of RStudio. We can see that there is a total of 170MB of memory being used when lm.fit is called. That is 170-53.7=116.3MB of additional memory.

```
library(pryr)
mem_used()
# 21.6 MB

y <- rnorm(1000000)
x1 <- rnorm(1000000)
x2 <- rnorm(1000000)
x3 <- rnorm(1000000)
mem_used()
#53.7 MB

debug(lm)
lm( y ~ x1 + x2 + x3 )
#step through function; type the follwing command after lm.fit() is called
mem_used()
#170 MB

undebug(lm)
```

(b) Use debug to step through the lm function again, notice the following 3 lines resulted in the most memory usage in the function. Interestingly, being a subset of mf, y has a larger size; also, the call to model.response took more memory than object y alone did. Furthermore, x, the matrix created from mf also has a larger size than mf; although surprisingly, the call to model.matrix took less memory than the size of object x.

```
debug(lm)
lm( y ~ x1 + x2 + x3 )
#step through function; check memory usage after each line
mem_used()

#the following command resulted in 32MB of memory usage; evaluate call in parent environment,
#which is the global environment in this case, save result in object mf
mf <- eval(mf, parent.frame())
object_size(mf)
#32 MB

#the following command resulted in 80MB of memory usage; return the response variable
#data with type numeric from mf, save in object y
y <- model.response(mf, "numeric")
#note size of y is smaller than total memory used from the command above
object_size(y)
#64 MB

#the following command resulted in 40MB of memory usage; create a model matrix; mt is the
#terms attribute of mf
x <- model.matrix(mt, mf, contrasts)
#note size of x is larger than total memory used from the command above
object_size(x)
#88 MB

undebug(lm)
```

(c) Instead of saving x and y in their actual values, save them in the form of references to mf to reduce repetitive data will help save memory.