Problem Set 4

Alexander Brandt
SID: 24092167

October 14 2015

1

To discover the source of error, I used the debug() function to step through the tmp() function. The . I realized that the loaded seed was not being applied to the global environment, so I exported the loaded tmp.Rda data into the global enivronment with the envir variable and the globalenv() function.

```
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() {
  # Added globalenv() to solve the issue
  load('tmp.Rda',envir = globalenv())
  runif(1)
}
tmp()

## [1] 0.2655087
```

2

Here, to perform the calcuation in the log scale, I use a the standard log additive identity, with a special method for corner cases. By allowing k to equal an int, or a range, so it can return a vector of values. This is summed in both methods, with a timer, to show that the non-vector and vector answers are the same, though the vectorized expression is much, much faster.

```
denominator <- function(k,n,p,phi) {
  a <- lchoose(n,k);
  b <- k*log(k) + (n-k)*log(n-k) - n*log(n)
  c <- phi * (n*log(n) - k*log(k) - ((n-k)*log(n-k)));
  d <- (k*phi)*log(p);
  e <- (n-k)*phi*log(1-p);
  return(exp(a + b + c + d + e))
}

# The "corner cases" where k=0 or k=n.
denominator_cc <- function(k,n,p,phi) {
  a <- lchoose(n,k);
  b <- log(1)
  c <- phi * log(1);
  d <- (k*phi)*log(p);
  e <- (n-k)*phi*log(1-p);
  return(exp(a + b + c + d + e))
}

N <- 2000
a <- sequence(N-1)

# 2a)
ptm <- proc.time()
v1 <- unlist(lapply(a,denominator,n=N,p=0.3,phi=0.5))
sum(v1) + denominator_cc(k=0,n=N,p=0.3,phi=0.5) +
  denominator_cc(k=N,n=N,p=0.3,phi=0.5)
```

```
## [1] 1.414436
```

```
proc.time() - ptm
```

```
##    user  system elapsed
##   0.019   0.002   0.021
```

```
# 2b) FULLY VECTORIZED!
ptm <- proc.time()
v2 <- denominator(a,n=N,p=0.3,phi=0.5)
sum(v2) + denominator_cc(k=0,n=N,p=0.3,phi=0.5) +
  denominator_cc(k=N,n=N,p=0.3,phi=0.5)
```

```
## [1] 1.414436

proc.time() - ptm

##    user  system elapsed
##   0.003   0.000   0.003
```

3

Here we construct a one line solution, and an "optimized" solution for the problem of weighted linear addition of the observations. The comments are useful in comparing the two approaches, but one is a more "brute force" method of calcuating the sums vector, and the other attempts to use a spare matrix with matrix multiplication to construct the sums.

```
# Question 3

# Import the matrix library that allows for a sparse matrix.
# Otherwise, it would overwhelm even powerful RAM systems.

library(Matrix)
mmrda <- "~/Dropbox/solutions_github/stat243/ps4/mixedMember.Rda"
load(mmrda)

# 3a)

# In the "one line" sapply function, the weights are applied
# to the mu's given by a set of id's in the IDsX object.

my_sums_A <- sapply(1:length(IDsA),
                    function(i) sum(wgtsA[[i]] * muA[IDsA[[i]]]))
my_sums_B <- sapply(1:length(IDsB),
                    function(i) sum(wgtsB[[i]] * muB[IDsB[[i]]]))

# 3b)

# Here, a "selection" matrix is created, where each weight is
# built into the jth index for the ith row, and then the two
# matricies are multiplied together to create a linear algebra
# solution for the problem.  This method gives about a two
# order of magnitude speed up for both 3a and 3b.

ptm <- proc.time()
my_sums_A <- sapply(1:length(IDsA),
```

```
                    function(i) sum(wgtsA[[i]] * muA[IDsA[[i]]]))
proc.time() - ptm
```

```
##    user  system elapsed
##   0.191   0.005   0.197
```

```
selection_matrix_A <- Matrix(0,
                             nrow=length(IDsA),
                             ncol=length(muA),sparse=TRUE)

for (i in (1:length(IDsA)))
{
  selection_matrix_A[i,IDsA[[i]]] <- wgtsA[[i]]
}
ptm <- proc.time()
my_sums_v2_A <- selection_matrix_A %*% muA
proc.time() - ptm
```

```
##    user  system elapsed
##   0.004   0.000   0.005
```

```
head(my_sums_A)
```

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

```
head(unlist(my_sums_v2_A[,1]))
```

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

```
# 3c)

ptm <- proc.time()
my_sums_B <- sapply(1:length(IDsB),
                    function(i) sum(wgtsB[[i]] * muB[IDsB[[i]]]))
proc.time() - ptm
```

```
##    user  system elapsed
##   0.179   0.004   0.184
```

```
selection_matrix_B <- Matrix(0,
                             nrow=length(IDsA),
                             ncol=length(muB),sparse=TRUE)

for (i in (1:length(IDsA)))
{
  selection_matrix_B[i,IDsB[[i]]] <- wgtsB[[i]]
}
ptm <- proc.time()
my_sums_v2_B <- selection_matrix_B %*% muB
proc.time() - ptm
```

```
##    user  system elapsed
##   0.003   0.001   0.004
```

```
head(my_sums_B)
```

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

```
head(unlist(my_sums_v2_B[,1]))
```

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

4

Here we use the debug feature along with the mem_used and object_size methods from pryr to examine the memory requirements to construct our data, as well as the memory usage withing the lm method. We find that there is some bloat given various attributes from the objects using the attributes function, and then suggest some attributes that might be pared away to minimize the memory usage (such as the names attribute, which is effectively redundant).

```
# Question 4

library(pryr)

# A great option is mem_change and object_size from pryr in the following segments
# the command sort(sapply(ls(),function(x){object_size(get(x))})) ain't a bad way
# of sorting it out

# Size of N_pt4 is negligible
N_pt4 <- 1000000
mem_change(x1 <- rnorm(N_pt4))
```

```
## 8.01 MB
```

```
mem_change(x2 <- rnorm(N_pt4))
```

```
## 8 MB
```

```
mem_change(x3 <- rnorm(N_pt4))
```

```
## 8 MB
```

```
mem_change(b <- c(1,5,9))
```

```
## 1.43 kB
```

```
mem_change(y <- x1*b[1] + x2*b[2] + x3*b[3] + rnorm(N_pt4))
```

```
## 8 MB

# sort(sapply(ls(),function(x){object_size(get(x))}))
# debug(lm)

lm(y ~ x1 + x2 + x3)

##
## Call:
## lm(formula = y ~ x1 + x2 + x3)
##
## Coefficients:
## (Intercept)           x1           x2           x3
##   0.0002181    0.9994089    5.0004205    8.9977552

# 4b)

# The objects that are bigger than 10% of the original are:

# Browse[2]> object_size(get("mf"))
# 32 MB

# Browse[2]> object_size(get("x"))
# 88 MB

# Browse[2]> object_size(get("y"))
# 64 MB

# Browse[2]> object_size(get("z"))
# 168 MB

# The reasons that the vectors can be more than just 8 bytes x
# the number of elements is because R vectors/lists can feature
# several attributes/pieces of metadata, like the "names" of
# columns or lists for example, that require variable space. In
# our case, the "names" are just the indices (tautological).
#

# 4c)

# To reduce the memory usage before lm.fit() I would remove
# unnecessary attributes like £names from the data frame.
```