STAT243 - Problem Set 4 P3

Cristobal Pais - cpaismz@berkeley.edu

October 11th, 2017

Problem 3

In this problem We have two main functions that can be improved: (1) *ll()* for calculating the log-likelihood value of a current iteration and (2) *oneUpdate()* function that performs all the computations needed for updating (in an iterative way) all the relevant variables needed for calculating the new log-likelihood function until a convergence criterion (a certain threshold) is reached. The general strategy for improving both functions consists of:

1. Compact the *ll()* function in only one command line (expression) without explicitly calculating the indexes needed for the summation of the Θ matrix, saving both memory and computation time.
2. Eliminate as much as possible all the *for* loops (nested) inside the *oneUpdate()* function using vectorized operations in such a way that the same multidimensional array q is obtained in a far more efficient way, and delete all the extra (and unnecessary) calculations/variables/steps performed in order to obtain exactly the same final result as the original function.

In order to compare our proposed functions with the original ones, a series of microbenchmark comparisons will be performed for particular chunks of the code and for the overall function usage. Plots are provided for visualization purposes alongside with memory profilings.

a) Improving a log-likelihood function: AS-IS state

Before describing the modifications performed to both functions, we can analyze the current state of *ll()* and *oneUpdate()* in terms of memory usage and running time. In order to do this, we can use the useful "profvis" package that allows us to check the memory usage and running time of each line of the code in order to easily detect the bottlenecks and potential improvements of the current functions/steps. Therefore, we proceed as follows:

1. Relevant libraries are loaded in order to be able to analyze the code and obtain the same results by fixing a random seed for our analysis. We set the working directory and random testing parameters (*theta.init*) are generated using the data loaded from the .Rda file provided with the problem statement.

```
# No warnings (visualization purposes)
options(warn=-1)

# Loading libraries
library(profvis)
library(microbenchmark)
library(ggplot2)
library(pryr)
require(stats)

# Setting random seed for reproducibility
set.seed(1)
```

```r
# Set working directory and load the relevant data for A, n, K
setwd("C:/Users/chile/Desktop/Stats243/HW/HW4/Code/")
load('ps4prob3.Rda')

## Testing the functions
# Initialize the parameters at random starting values (remove temp, saving memory)
temp <- matrix(runif(n*K), n, K)
theta.init <- temp/rowSums(temp)
rm(temp)
```

2. The original function is loaded, a testing *Theta.init* value is generated (equivalent to *Theta.old* in the original code), and a microbenchmark running time analysis is performed for 10,000 times in order to be able to analyze the mean running time of the function.

```r
## Log-likelihood function
# LL original function
ll <- function(Theta, A) {
  sum.ind <- which(A==1, arr.ind=T)
  logLik <- sum(log(Theta[sum.ind])) - sum(Theta)
  return(logLik)
}

# ll() function microbenchmark results
Theta.init = theta.init %*% t(theta.init)
mbmll <- microbenchmark(ll(Theta.init, A), times = 1e4)

# Show and plot (if required) results
print(mbmll)
```

```
## Unit: milliseconds
##                expr      min       lq     mean   median       uq      max
##  ll(Theta.init, A) 2.193432 2.520587 3.516119 2.708754 3.655578 47.56356
##  neval
##  10000
```

```r
#autoplot(mbmll)
```

Based on the previous results, we can see that the original *ll()* function mean running time is around 3.41 milliseconds and its range is represented by [2.18, 50.08] milliseconds approximately. Since the operations performed inside this function are very simple and direct, there is not so much margin to improve it, however, as we will see in the next section, we can still improve its performance.

3. Now, the same procedure is replicated for the *oneUpdate()* function: definition and microbenchmark running time analysis.

```r
## Iteration function
# OneUpdate original function
oneUpdate <- function(A, n, K, theta.old, thresh = 0.1) {
  theta.old1 <- theta.old
  Theta.old <- theta.old %*% t(theta.old)
  L.old <- ll(Theta.old, A)
```

```
  q <- array(0, dim = c(n, n, K))

  for (i in 1:n) {
    for (j in 1:n) {
      for (z in 1:K) {
        if (theta.old[i, z]*theta.old[j, z] == 0){
          q[i, j, z] <- 0
        } else {
          q[i, j, z] <- theta.old[i, z]*theta.old[j, z] /
            Theta.old[i, j]
        }
      }
    }
  }
  theta.new <- theta.old
  for (z in 1:K) {
    theta.new[,z] <- rowSums(A*q[,,z])/sqrt(sum(A*q[,,z]))
  }
  Theta.new <- theta.new %*% t(theta.new)
  L.new <- ll(Theta.new, A)
  converge.check <- abs(L.new - L.old) < thresh
  theta.new <- theta.new/rowSums(theta.new)
  return(list(theta = theta.new, loglik = L.new,
              converged = converge.check))
}

# oneUpdate() function microbenchmark results
mbmoneUp <- microbenchmark(oneUpdate(A, n, K, theta.init), times = 1)

# Show and plot (if required) results
print(mbmoneUp)
```

```
## Unit: seconds
##                            expr      min       lq     mean   median
##  oneUpdate(A, n, K, theta.init) 96.44214 96.44214 96.44214 96.44214
##        uq      max neval
##  96.44214 96.44214     1
```

```
#autoplot(mbmoneUp)
```

In this case we can notice that the mean running time per iteration is around one minute and 30 seconds: very slow and poor performance for a function that we would need to run maybe hundred of times before reaching the algorithm convergence. Therefore, a series of improvements can be implemented as we will see in the next section (b).

4. In order to finalize our AS-IS analysis, we perform a code profiling using the *profvis/profmem* package that will allow us to detect the current bottlenecks and memory usage of each line of code in the original functions.

```
# Loading libraries
library("profmem")
```

```r
# Original code profiling (using .init instead of .old for consistency)
porig <- profmem({

## Log-likelihood function
# LL original function
ll <- function(Theta, A) {
  sum.ind <- which(A==1, arr.ind=T)
  logLik <- sum(log(Theta[sum.ind])) - sum(Theta)
  return(logLik)
}

# OneUpdate original function
oneUpdate <- function(A, n, K, theta.old, thresh = 0.1) {
  theta.old1 <- theta.old
  Theta.old <- theta.old %*% t(theta.old)
  L.old <- ll(Theta.old, A)
  q <- array(0, dim = c(n, n, K))

  for (i in 1:n) {
    for (j in 1:n) {
      for (z in 1:K) {
        if (theta.old[i, z]*theta.old[j, z] == 0){
          q[i, j, z] <- 0
        } else {
          q[i, j, z] <- theta.old[i, z]*theta.old[j, z] /
            Theta.old[i, j]
        }
      }
    }
  }
  theta.new <- theta.old
  for (z in 1:K) {
    theta.new[,z] <- rowSums(A*q[,,z])/sqrt(sum(A*q[,,z]))
  }
  Theta.new <- theta.new %*% t(theta.new)
  L.new <- ll(Theta.new, A)
  converge.check <- abs(L.new - L.old) < thresh
  theta.new <- theta.new/rowSums(theta.new)
  return(list(theta = theta.new, loglik = L.new,
              converged = converge.check))
}

# Call the function
oneUpdate(A, n, K, theta.init)
})

# Display results
porig
```

As seen in the previous output, the code profiling analysis is called using the profvis/profmem function. Based on the profiling results, we can see that the largest amount of time of the function (and hence, its bottleneck) is spent in the three *for* nested loops for generating the multidimensional array q. Alongside with this, it is clearly the operation that demands more memory from the system.

Thus, we will focus our improvements in trying to avoid the nested loops implementation using vectorized and algebraic operations for computing the relevant values (q) and perform an efficient iteration.

b) Improving a log-likelihood function: Proposed solution

Based on the AS-IS state, we perform a series of modifications to the original functions in order to improve the performance of both of them. A great performance improvement is obtained when the *oneUpdate()* function is reformulated in a way that it takes advantage of the structure of the problem using a series of algebraic (and vectorized) operations instead of a series of nested *for* loops (very slow implementation). In addition, some steps are eliminated or modified in order to perform the minimum number of calculations as possible.

Although the original *ll()* function is pretty efficient due to the fact that its operations are straightforward and direct, we are able to improve it by performing all the calculations in just one line.

1. Instead of computing, extracting, and recording the relevant indexes for the log-likelihood function summation, we can simply contract everything in one command line where we include the values of the A matrix that are equal to one. It is very easy to check that both functions perform exactly the same calculations but in our version, no intermediate steps are needed. In order to compare them, we perform a microbenchmark comparison using the loaded data for testing, and a value checking step is added for visualization purposes:

```
# LL new function: All computations in one line (no need to save indexes)
llNew <- function(Theta, A) {
  logLik <- sum(log(Theta[A == 1])) - sum(Theta)
  return(logLik)
}

## Log-likelihood fuctions comparison
mbmllcomp <- microbenchmark(llorig <- ll(Theta.init, A),
                            llnew <- llNew(Theta.init, A), times = 1e4)

# Print results
print(mbmllcomp)
```

```
## Unit: milliseconds
##                           expr      min       lq     mean   median
##    llorig <- ll(Theta.init, A) 2.202413 2.363211 2.505571 2.413246
##  llnew <- llNew(Theta.init, A) 1.671695 1.791011 1.905097 1.831211
##        uq      max neval
##  2.517380 38.23302 10000
##  1.912464 45.17084 10000
```

```
# Plot results (if required)
#autoplot(mbmllcomp)

# Check values
print(paste("Log-likelihood values:", llorig, ",", llnew, sep = ""))
```

```
## [1] "Log-likelihood values:-55574.3696913836,-55574.3696913836"
```

2. In the case of the *oneUpdate()* function, our main modification (and more relevant in terms of performance impact) consists of contract the series of *for* loops into one simple expression that exploits vectorized operations while using a simple call of the *lapply*() function. The expression is based on the structure of the computation: every iteration we are multiplying a specific entry of the *theta.init* (old in the original notation) vector by another one, depending on the values of i, j, and z. Then, an unnecesary zero value assignment is performed since the q multidimensional array is initialized with all its entries equal to zero so we will be able to eliminate that step, and on the other hand, the operations performed when no null values are multiplied can be easily vectorized when we notice that a similar result can be obtained by simply using an outer multiplication between vectors and divide it by the relevant value of the *Theta.init* (.old in the original code) matrix.

   As an extra safety step, we include an if clause in the proposed expression for detecting null values inside the *Theta.init* matrix such that we avoid divisions by zero (inf numbers) or NaN values. Important is to note that q is no longer a multidimensional array in the new function, but a multidimensional list: better memory performance and easy to manipulate or unlist if we need it.

```r
# Original function: q calculation step
qOneUpdate <- function(){
  q <- array(0, dim = c(n, n, K))

  for (i in 1:n) {
    for (j in 1:n) {
      for (z in 1:K) {
        if (theta.init[i, z] * theta.init[j, z] == 0){
          q[i, j, z] <- 0
        } else {
          q[i, j, z] <- theta.init[i, z] * theta.init[j, z] / Theta.init[i, j]
        }
      }
    }
  }
  return(q)
}

# New function: q calculation step
qOneUpdateNew <- function(){

  # Replace of the nested for loops: one lapply call using 1:K vector as the
  # main input. New coordinate system q[[z]][i,j] instead of q[i, j, z]
  q <- lapply(1:K,
              function(x) {
                ifelse(Theta.init != 0,
                       (theta.init[, x] %o% theta.init[, x]) / Theta.init,
                       0)
                })
  # Return the q multidimensional list
  return(q)
}
```

3. In order to test our proposed function, we perform a set of specific comparisons for the code needed to obtain q in both the original and the new function versions as follows:

```r
# Microbenchmark analysis
mbmq <- microbenchmark(q <- qOneUpdate(), qNew <- qOneUpdateNew(), times = 1)

# Print results
print(mbmq)
```

```
## Unit: seconds
##                      expr        min         lq       mean     median
##        q <- qOneUpdate() 100.382178 100.382178 100.382178 100.382178
##  qNew <- qOneUpdateNew()   2.128431   2.128431   2.128431   2.128431
##          uq        max neval
##  100.382178 100.382178     1
##    2.128431   2.128431     1
```

```r
# Checking results: dimension and comparison
qNewplain <- unlist(qNew)

# Comparison with original q
print(paste("Number of different elements: ", length(q[!(q == qNewplain)]),
            sep = ""))
```

```
## [1] "Number of different elements: 0"
```

```r
print(paste("Lenght of original q: ", length(q), sep = ""))
```

```
## [1] "Lenght of original q: 25250000"
```

```r
print(paste("Lenght of new q: ", length(qNewplain), sep = ""))
```

```
## [1] "Lenght of new q: 25250000"
```

```r
# Checking random values qV2[[z]][i, j] = q[i, j, z] for completeness
q[1, 2, 3] == qNew[[3]][1, 2]
```

```
## [1] TRUE
```

```r
q[24, 15, 77] == qNew[[77]][24, 15]
```

```
## [1] TRUE
```

```r
q[444, 29, 61] == qNew[[61]][444, 29]
```

```
## [1] TRUE
```

```r
# Plot results (uncomment if required)
#autoplot(mbmq)
```

Based on the results we can see that our proposed algorithm for calculating q obtains a far better performance than the original algorithm, reaching a **63.5-fold speedup**, a very impressive result. This great performance is related to the fact that we are exploiting the efficient vectorized operations in R (in a very smart way) while avoiding the poor performance nested for loops. We can see that the mean running time for the original function is around 92.06 seconds (1.5 minutes) while our new function just needs 1.43 seconds in average for solving exactly the same problem.

3. Finally, we define our new function, including the following relevant changes:

   i) Deletion of unnecessary lines (commented in the code).

   ii) Replace of the nested for loops structure.

   iii) Non-explicit computation of $Theta.new$ for calculating the new log-likelihood.

```r
# OneUpdate new function
oneUpdateNew <- function(A, n, K, theta.old, thresh = 0.1) {
  # Delete the Theta.old1 reference (unused), keep the next two lines
  Theta.old <- theta.old %*% t(theta.old)
  L.old <- llNew(Theta.old, A)

  # Replace of the nested for loops: one lapply call using 1:K vector as the
  # main input. New coordinate system q[[z]][i,j] instead of q[i, j, z]
  q <- lapply(1:K,
              function(x) {
                ifelse(Theta.old != 0,
                       (theta.old[, x] %o% theta.old[, x]) / Theta.old,
                       0)
                })

  # For loop is performed using list q (better performance than a/la/sapply
  # functions), modification in place without creating an extra copy theta.new
  for (z in 1:K) {
    theta.old[, z] <- rowSums(A * q[[z]]) / sqrt(sum(A * q[[z]]))
  }

  # New reference after modification (for simplicity)
  theta.new <- theta.old

  # Theta.new is created inside the llnew function call since Theta.new is not
  # used in the rest of the function, saving memory.
  L.new <- llNew(theta.new %*% t(theta.new), A)

  # Check convergence value
  converge.check <- abs(L.new - L.old) < thresh

  # Update theta.new
  theta.new <- theta.new / rowSums(theta.new)

  # Return results
  return(list(theta = theta.new, loglik = L.new,
              converged = converge.check))
}
```

4. Therefore, we perform a global comparison of both the original and new functions as follows:

```r
## Full functions comparison
# Individual calls
out <- oneUpdate(A, n, K, theta.init)
out2 <- oneUpdateNew(A, n, K, theta.init)
print(paste("Number of different Theta values: ",
            length(out$theta[!(out$theta == out2$theta)]),
            sep = ""))
```

```
## [1] "Number of different Theta values: 0"
```

```r
print(paste("Value of LLs obtained: ", out$loglik, ",",
            out2$loglik, sep = ""))
```

```
## [1] "Value of LLs obtained: -41655.6856747477,-41655.6856747477"
```

```r
print(paste("Convergence status: ", out$converged, ",",
            out2$converged, sep = ""))
```

```
## [1] "Convergence status: FALSE,FALSE"
```

```r
# Perform a single update
system.time(out <- oneUpdate(A, n, K, theta.init))
```

```
##    user  system elapsed
##   95.36    0.18   95.56
```

```r
system.time(out2 <- oneUpdateNew(A, n, K, theta.init))
```

```
##    user  system elapsed
##    1.06    0.83    1.90
```

```r
# Benchmark for multiple calls
mbm <- microbenchmark(oneUpdate(A, n, K, theta.init),
                      oneUpdateNew(A, n, K, theta.init),
                      times = 1)

# Print results
print(mbm)
```

```
## Unit: seconds
##                               expr       min        lq      mean    median
##     oneUpdate(A, n, K, theta.init) 95.570848 95.570848 95.570848 95.570848
##  oneUpdateNew(A, n, K, theta.init)  1.998067  1.998067  1.998067  1.998067
##         uq       max neval
##  95.570848 95.570848     1
##   1.998067  1.998067     1
```

```
# Plots (uncomment if required)
#autoplot(mbm)
#boxplot(mbm, log = FALSE, unit = "s", xlab = "Functions")
```

Based on the results obtained, we can easily see that we were able to improve the performance of the original function in a very significant way, reaching speedups up to 63.5-folds thanks to the smart vectorized operations instead of a series of nested for loops inside the original function.