Liya Li

HW #3 Report
Data Representations and Clustering

Q1: Object Sizes

1. Compared to storing everything in one table, the advantages of using lookup tables to represent the agencies and words are that, firstly, it helps to make faster information searching because it leads to direct references of agencies and words for users; secondly, it provides easier access to agencies and words information because the look up tables are considered as "storing only needed information" which mapping to the main weights table, while storing everything in one table takes not only much more time to search and access information, but also much more memory to store the data. However, the disadvantages are that users can't view the details about agencies and words linking with according weights, which means that it gets harder to read agencies, key words, and word weights at the same time for comparison or similar purposes.

2. The algebraical sizes and theoretical sizes of objects are shown in this following table:

Sizes Comparison	Algebraical Size	Actual Size
The triple representation in memory (the data frame weights)	23,336,544 bytes	23,337,992 bytes
The sparse matrix X, with columns for each agency and rows for each word	17,503,384 bytes	17,504,800 bytes
The transpose of X, with columns for each word and rows for each agency	18,863,276 bytes	18,864,696 bytes
The dense matrix version of X	661,379,904 bytes	661,381,016 bytes

3. The algebraical sizes are slightly smaller around 1KB than the actual sizes for objects above, which is as mentioned during a lecture.

   The size of weights.csv file on disk in ASCII text is 44,330,812 bytes. Compared the above objects' actual sizes with it, we can clearly see that the triple representation in memory, the sparse matrix X, and the transpose of X have smaller actual sizes than the weights.csv file size on disk in ASCII text, which means that in R, it takes much more memory to store weights.csv file directly than using those representations. However, the dense matrix version of X takes way much more memory than the weights.csv file on disk.

   The sparsity of dense matrix version of X is 98.23577% which refers that the dense matrix contains lots of zero-valued elements. This also gives that the density is 1.764231%, which means that this dense matrix of X is not that dense.

   After comparing the sizes, we can conclude that the most efficient memory representation for this particular data is to use the sparse matrix with columns for each agency and rows for each word.

If we were getting a smaller number of words than the number of agencies, the transpose representation with columns for each word and rows for each agency may work better in memory storing. However, dense matrix could not work better than sparse matrix since the sparse matrix uses integer arrays to represent data, each element with 4 bytes only, while the dense matrix uses one double array to represent data, each element with 8 bytes.

Q2: Clustering

1. crossprod uses meanly 541.0998 milliseconds, while explicitly computing XTX uses meanly 550.5521 milliseconds. Thus, crossprod is faster. The reason is that, %*% for explicitly computing XTX needs an explicit matrix transposition for X, leading to transposition overhead, while crossprod avoids all explicit matrix transposition.

2. crossprod on the sparse version of X uses the CPU time with value 0.018, while on the dense version of X uses 0.111. Thus, crossprod on the sparse version of X is faster. The reason is that, crossprod on the sparse version of X accesses less actual element values than on the dense version of X, for from the beginning, the dense matrix contains the zero-valued elements which need calculation while the sparse matrix accordingly contains nothing in those spots. Less computations leads to faster computation.

3. The range of similarity scores that appear between different agencies is [1.970153e-06, 0.9644715]. The most similar two agencies are with the name “agencies$agency_name[abs(a1)]” and “Employment and Training Administration”. This means that the words they are using as keywords are mostly the same or very similar.

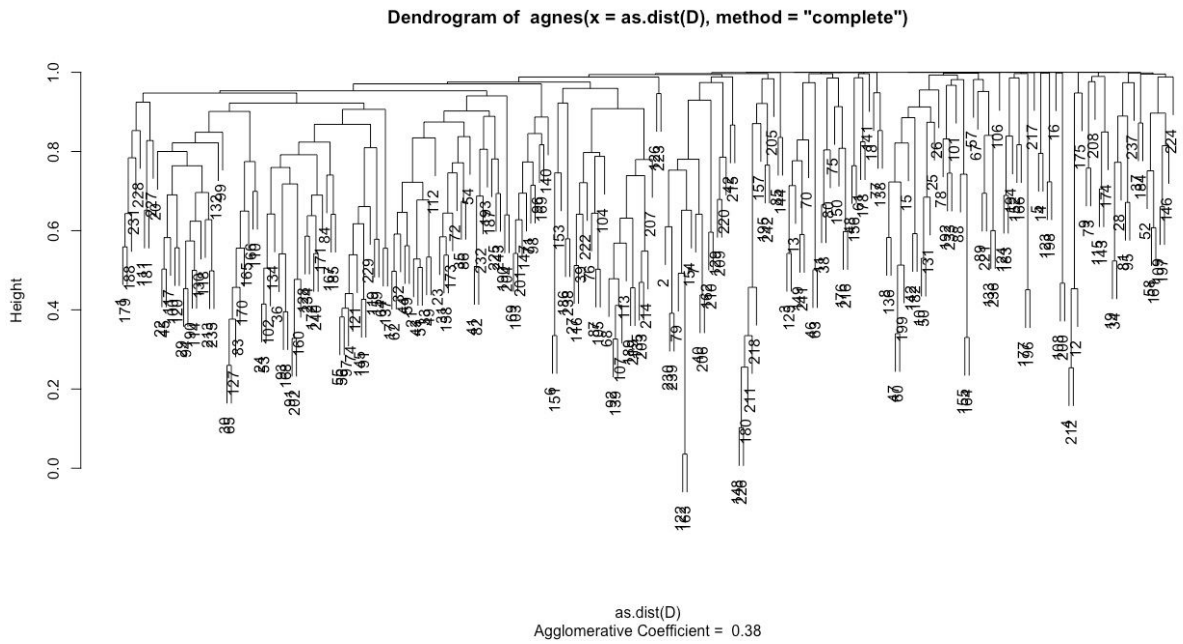
4. The agencies with name “Employment and Training Administration” and “agencies$agency_name[abs(a1)]” are grouped together first. This is exactly what I expected based on the previous question.

5. The first group of 3 agencies have names “Office of the Inspector General”, “Office of Inspector General” and “Office of Inspector General”.

   The first group of 4 agencies have names “Office of the Inspector General”, “Office of Inspector General” and “Office of Inspector General”, and “Assistant Secretary for Administration”.

   Agglomerative clustering seems not doing something reasonable because they appears repeated agency names in the merging.

6. The two clusterings methods we used are seem to be similar. For example, here is the clustering plot for model1:



7. I would agree that clustering is a quite subjective task because that since the steps taken in preparing the data, especially when the keywords are cleaned and extracted, a lot of subjective decisions were made to obtain the keywords. However, clustering does an important role in helping to understand relationships between similar agencies.

Appendix: R script

```
############## STA 141C HW3 - LIYA LI
############## Packages
library(data.table)
library(methods)        # use before Matrix library
library(Matrix)
library(cluster)
library(microbenchmark)  # for timing

##############
############## Load in the data
##############
# get file names from the zip file "award_words.zip" after unzipping it
zip_path = "~/Desktop/Winter Quarter 2019/STA 141C/hw3/award_words.zip"
files = unzip(zip_path, list = TRUE)
```

```
files$Name

##############
############## Q1: Object sizes
##############
# helper function to read in csv file
get_csv = function(fname){
  unzip(zip_path, fname)
  fread(fname)
}

weights = get_csv("weights.csv")
agencies = get_csv("agencies.csv")

unzip(zip_path, "words.csv")        # words.csv has no header: use header = FALSE in fread/read.csv, or use
read.table
words = fread("words.csv", header = FALSE)

### notations
d = dim(agencies)[1]              # number of distinct agencies, same as: d = max(unique(weights$agency_index))
w = dim(words)[1]                 # nubmer of distinct words
n = dim(weights)[1]               # number of observations in weights.csv
si = 4                          # size of an integer in R, bytes
sd = 8                           # size of a double precision floating point number in R, bytes

############## 1.2 Compute the sizes algebraically, and verify the sizes in R.
### the triple representation in memory(the dataframe weights = read.csv("weights.csv"))
str(weights)
# check closer what is inside "weights"
df_alge = n * 2 * si + n * 1 * sd   # weights has two int columns and one numeric(double) column
df_alge
df_r = object.size(weights)
df_r

### the sparse matrix X with columns for each agency and rows for each word (Use Matrix::sparseMatrix)
# sparse matrix: most of the elements are zero. If most of the elements are nonzero, then the matrix is considered
dense.
column = weights$agency_index
row = weights$word_index

X = sparseMatrix(i = row, j = column, x = weights$weight)
# X1 = as(X, "dgCMatrix")      # identical(X,X1) returns True
# if dgRMatrix: row oriented
str(X)      # i is the column index of each "value", p is the i array index for leading "value" from each row

X_alge = n * sd + n * si + 244 * si # size of array i + array p + array x
X_alge
```

```
X_r = object.size(X)
X_r

### the transpose of X, with columns for each word and rows for each agency
XT = t(X)
str(XT)
XT_alge = n * sd + n * si + 340217 * si
XT_alge
XT_r = object.size(XT)
XT_r

### the dense matrix version of X (this is to change the .'s to 0)
XD = as(X, "dgeMatrix")
str(XD)
XD_alge = n * sd + sum(XD==0) * sd # be careful here, all the 0's are numeric
XD_alge
XD_r = object.size(XD)
XD_r

############## 1.3
### compare to ASCII text
text_size = file.info(unzip(zip_path, "weights.csv") )
text_size

### sparsity = # of zeroes / # of elements
sparsity = sum(XD == 0)/ (ncol(XD) * nrow(XD))
sparsity
density = 1-sparsity
density

##############
############## Q2: Clustering
############## 2.1
microbenchmark(crossprod(X))  # use system.time(crossprod(X)) is fine too
microbenchmark(t(X) %*% X)

############## 2.2
system.time(crossprod(X))
system.time(crossprod(XD))

############## 2.3 similarity scores range, most similar agencies
sim_mat1 = crossprod(X)
str(sim_mat)
isSymmetric(sim_mat)   # TRUE

# compare only the elements except diagnal values
# options(scipen=999): take off scientific notation
```

```
min_similarity = min(sim_mat[ row(sim_mat) != col(sim_mat) ])
min_similarity
max_similarity = max(sim_mat[ row(sim_mat) != col(sim_mat) ])
max_similarity

# find max similarity and its position in the symmetric matrix
index = which(sim_mat == max_similarity)    # return indexes as in a vector
agency1 = agencies$agency_name[index[1]%%d]
agency1
agency2 = agencies$agency_name[index[2]%%d]
agency2

############## 2.4 fit agglomerative clustering model
D = 1 - sim_mat          # distance matrix between agencies
model1 = agnes(as.dist(D), method = "complete")

# which 2 agencies group together first
merging = model1$merge
merging
a1 = merging[1,1]
agency1_name = agencies$agency_name[abs(a1)]
a2 = merging[1,2]
agency2_name = agencies$agency_name[abs(a2)]
agency2_name

############## 2.5 first group of 3, first group of 4
group3_1 = abs(merging[2,1])
group3_2 = abs(merging[2,2])
group3_3 = abs(merging[4,2])
first3 = c(group3_1, group3_2, group3_3)
first3
first3_names = agencies$agency_name[first3]
first3_names

group4 = abs(merging[9,2])  # since the [9,2] one is linked to row 4, while row 4 links to row 2, so these four
agencies join tgh
first4 = c(first3, group4)
first4_names = agencies$agency_name[first4]
first4_names

############## 2.6 fit partitioning around medoids clustering model with k=2 clusters
model2 = pam(as.dist(D), k = 2L)
model2$clustering

# plot for model1
plot(model1, which.plots = 2L)
```

```
# plot for model2
points(model2$medoids, bg = "blue", pch = 21)
```