Jared Yu

1. Object Sizes

1. Using the lookup table allows us to compress the data into a simple format. It gives us the ability to save space in addition to organizing the data. It can however become troublesome to have incredibly large lookup tables when there are many different categories. When storing them all in one table, there may be an issue of understanding the organization of the data without having a table to understand how the data is distributed. The most noticeable advantage towards having one large data frame with multiple columns, and having variables repeated many times throughout the columns would be the space saved. Avoiding this repetition as much as possible by using lookup tables can help to save large amounts of space when datasets are considerably large.

2. The triple representation in memory of the weights data frame is calculated by accounting for the two types of data that exist in the matrix, integers and double. The first two columns are integer, and the last column is double. So, to calculate the expected size we have:
$$(n * 2 * s_i) + (n * 1 * s_d) = (1{,}458{,}534 * 2 * 4) + (1{,}458{,}534 * 1 * 8) = 23{,}336{,}544\ bytes$$

   Using the object.size() function to check the actual size of the weights data frame in memory, the size comes out to $23{,}337{,}544\ bytes$. This is actually $1{,}000\ bytes$, or $1KB$, larger, which is due to the overhead of storing the data in memory. This was also explained in the question for this problem.

   The sparse matrix $\boldsymbol{X}$ has an expected size of $(n \times s_d) + ((d+1) \times s_i) + (n \times s_i) = (1{,}458{,}534 \times 8) + \big((243+1) \times 4\big) + (1{,}458{,}534 \times 4) = 17{,}503{,}384$. This formula is due to the compressed sparse row format of the Matrix package. There are three arrays stored to represent the sparse matrix within the memory. The first array, $(n \times s_d)$ stores all the nonzero-valued elements. So, this is represented as n for all the elements multiplied by s_d, the format they are stored in. The second array $((d+1) \times s_i)$, stores the index within the first array of the first nonzero value in each column of the matrix. The reason for the $+1$ is that it adds the number of elements in the first array as well. These are all done in s_i format. The last array stores the index of the row of each element in the first array. The size of this last array can be algebraically represented as $(n \times s_i)$. The first array is known as the value array, the second as the row array, and the third as the column pointer array.

   Using the object.size() function to check the actual size of the sparse matrix $\boldsymbol{X}$, the size is $17{,}504{,}880\ bytes$, so the actual size is $1{,}496\ bytes$ larger than expected. It is interesting to note the $+1$ calculation in the second array which accounts for the number of nonzero elements. In this case, it seems to overcount by $4\ bytes$, however in the transpose of $\boldsymbol{X}$ it seems that there is no such issue. In the following calculation for the transpose matrix, the difference is an even $1{,}500\ bytes$. However, this apparent 'nice' number is not better by any known reason. This calculation seems to show that there is some unknown factor as to how the true size is calculated, and it may not be reasonable to say that the entirety of the extra bytes is

due to overhead. It is possible that there is a mistake in the assumptions behind the algebraic calculations.

The transpose of matrix $\boldsymbol{X}$, or $\boldsymbol{X}^T$ would have a larger expected size as the sparse matrix. The difference would be that since the rows and columns are transposed, there would be an increase due to the columns being of length w rather than d. The algebra for the calculation would be $(n \times s_d) + ((w+1) \times s_i) + (n \times s_i) = (1{,}458{,}534 \times 8) + \big((340{,}216+1) \times 4\big) + (1{,}458{,}534 \times 4) = 18{,}863{,}276$.

Using the object.size() function to check the actual size of the transpose of the sparse matrix $\boldsymbol{X}$, or matrix $\boldsymbol{X}^T$, the size is $18{,}864{,}776\ bytes$, so the actual size is $1{,}500\ bytes$ larger than expected. Here, in this case, where using the $+1$ to account for the number of nonzero elements in the second array, the resulting difference in calculation is a 'nice' even $1{,}500\ bytes$. Once again, however, it is worth noting that there is not currently a way to understand if this even number accounts purely for overhead, or there is some factor which is not being correctly estimated.

The dense matrix version of $\boldsymbol{X}$ would have an expected size of $w \times d \times s_d = 340{,}216 \times 243 \times 8 = 661{,}379{,}904$. This is because the matrix has the dimensions of $w \times d$, where all the elements are all of type s_d. The w rows represent the unique word indices, and d columns represent the unique agencies. It is interesting that the $0's$ are also of type double, like all the weight values.

Using the object.size() function to check the actual size of the dense matrix $\boldsymbol{X}$, the size is $661{,}381{,}080\ bytes$, so the actual size is $1{,}176\ bytes$ larger than expected. This makes sense due to the overhead involved. Also, the size of the apparent overhead is close to the $1{,}000\ bytes$ that was mentioned in the question for the assignment. The format of 'dgeMatrix' may have some unknown properties that make it difficult to consider exactly the overhead of the matrix within memory.

3. Below is a table of the sizes that have been previously calculated and their difference between the size of weights.csv as it exists on the disk. (all the values are in bytes):

Data	Actual Size	Size difference between actual and weights.csv
$weights.csv$	44,330,812	0
$weights\ data\ frame$	23,337,544	20,993,268
$Sparse\ matrix$	17,504,880	26,825,932
$Sparse\ matrix\ transpose$	18,864,776	25,466,036
$Dense\ matrix$	661,381,080	−617,050,268

The table is comparing the size of weights.csv which was found by using file.info() in RStudio. The size is roughly $44\ MB$, considerably large compared to the other pieces of data stored in memory, except for the dense matrix. Apparently, merely turning the weights.csv into a data frame takes up less space than storing the entire weights.csv on the disk. The sparse matrix also saves a considerable amount of space, more than half the size of the weights.csv is saved when the file is stored in such a format of memory. The dense matrix is surprisingly inefficient, taking

up much more space in memory than the weights.csv file itself requires. However, since the weights.csv file itself is rather sparse, it is difficult to know how it would compare with a dataset that had all nonzero values. In such a case, it is possible that the dense matrix would be comparatively less inefficient.

The sparsity of the matrix is calculated with the formula:

$$1 - \frac{n}{w \times d} = 1 - \frac{1{,}458{,}534}{340{,}216 \times 243} = 0.9823577$$

This calculates the number of zero-valued elements in the sparse matrix. It shows that the matrix is highly sparse, and it makes sense to store the data in such a format to save space in the memory of the computer. If the sparsity was much lower, like a matrix with no nonzero elements, then it wouldn't have made sense to attempt to store the matrix using a sparse matrix format. The additional overhead would have made such a decision counterproductive.

The most efficient of the methods in storing the data is the sparse matrix at $17{,}504{,}880\ bytes$. The reason is that out of all the formats, this one was the smallest, hence saving the most memory. This is true for this dataset specifically, with the operations that we have planned to use. It is possible, that with certain objectives and a different dataset, it would be ideal to use a dense matrix for the purpose of the given assignment. This will be further elaborated on in the following paragraph.

Having a sparse matrix would be useful if the number of rows and columns are extremely high such as in this case. Using a dense matrix to store all the data would require an unnecessary number of zeroes to also be stored in memory. It is also possible that calculations are faster in the sparse matrix, due to the fewer number of elements that need to be considered before the calculation is completed. However, the speed improvement may not exist if there is no implementation for a sparse matrix in the programming package. A dense matrix then may have the advantage that it is the required format for a matrix to be stored in to work properly. If the usage of the sparse matrix has implementation issues, then the space saved by storing it in such a format would be useless. In such cases, it would be wise to check the documentation of the programming package to be sure that it is practical to store the data in a specialized sparse matrix format.

2. Clustering

1. Below is a table of the times for crossprod() versus a manual computation on $\boldsymbol{X}$:

$Technique$	$Elapsed\ Time, system.time\ (seconds)$
$crossprod()$	13.50
$\% * \%$	16.82

The crossprod and %*% were used on the dense matrix. In this case, the crossprod function proved to be faster by more than 3 seconds. When using %*%, it is necessary to use the t() function to create the transpose beforehand. This extra step can create unnecessary calculation time, which can be avoided if directly using crossprod(). For this reason, the crossprod() function is faster than manually calculating %*% using a dense matrix.

2. Below is a table of the times for crossprod() versus a manual computation on the sparse matrix $\boldsymbol{X}$:

$Technique$	$Median\ Time, microbenchmark\ (milliseconds)$
$crossprod()$	293.3959
$\% * \%$	287.3542

Here it is evident that the crossprod() function is slower than using %*% to compute the product of the sparse matrices. It is possible that the benefit of using an optimized function is not present in a sparse matrix. The sparse matrix itself has few elements, and so using an optimized function may provide too much overhead to prove effective in its use for this case. The difference in calculation time is less than $10\ milliseconds$. Previously, the advantage of not having to calculate the transpose of the matrix provided an advantage to the crossprod() function. However, in this case, the matrix is in the sparse format, and such a calculation is possibly not as intensive to compute in RStudio. Since the times are so close, it is possible that if the matrix were larger, or that the data type was different, then the difference would be different.

3. The range of similarity scores in matrix $\boldsymbol{D}$ is $(0.03552853, 0.99999803)$. The agencies which are most similar are agencies 122 and 163, with a value of 0.03552853. The matrix $\boldsymbol{D}$ is a distance matrix, so the closest two distances are the most similar. This implies that the words that are used by the two agencies are used frequently and are also weighted highly within the agency in terms of their importance. The technique used was to set the diagonal elements of the matrix to -1, and then search for the smallest nonnegative element in the matrix. This helped to avoid searching for extremely small values which are virtually 0, due to them being placed along the diagonal.

4. The agnes() function has two an argument called method within the parameters. The 'simple' argument searches for the minimum distance between points, creating a snake-like connection between points before all the data is connected. The 'complete' argument uses the maximum distance between points, allowing for a different type of clustering to take place which is different from the snake-like result produced by 'simple.' Results from both arguments will be analyzed for problems 4 and 5. In problem 6 it will become clear why the 'complete' argument is chosen as the cluster to compare with the pam() cluster. In method = 'simple', the first two agencies grouped together are agencies 122 and 163 (by matrix index). This makes sense since they are the nearest agencies in matrix $\boldsymbol{D}$. It would be surprising if there were another two agencies chosen, as these were already determined to be the most similar. The same two agencies are grouped together in method = 'complete.'

5. In method = 'simple', the first group of three agencies are agencies: $148, 226$, and 180. The first group of four agencies are agencies: $148, 226, 180$, and 218 (by matrix index). In method = 'complete,' the first group of three agencies are the same as in method = 'simple.' However, in method = 'complete,' the first group of four agencies are agencies: $148, 226, 180$, and 211. The following is a table of matrix index, agency ID's and agency names:

$\boldsymbol{D}$ $Index$	$Agency\ ID$	$Agency\ Name$
148	770	$Office\ of\ the\ Inspector\ General$
226	459	$Office\ of\ Inspector\ General$
180	270	$Office\ of\ Inspector\ General$
218	213	$Office\ of\ the\ Inspector\ General$
211	886	$Assistant\ Secretary\ for\ Administration$

It becomes quickly clear why so many of the agencies were clustered together. The first four agencies under method = 'simple' and the first three agencies under method = 'complete' are essentially the same agency. They only slightly differ in the spelling of the name, with the word 'the' being omitted in two agencies ($226,180$). Doing a bit of research online, this agency may exist within various departments and different areas of the government. Their goal is to investigate fraud, misconduct, etc. by employees or contractors, and so it is not surprising that they may be using a similar spending throughout each of the spread-out agencies to perform the same task. The Assistant Secretary for Administration plays a role that is similar in nature to what the other agencies do, in that it provides oversights for departments to make sure that they are operating according to certain rules. In that regard, it does make sense for them to be clustered together.

6. Below is a table of agnes() and pam(),

$agnes()$	'$method\ =\ complete$'	$pam()$	'$k = 2$'
$Group\ 1$	$Group\ 2$	$Group\ 1$	$Group\ 2$
218	25	155	88

The agnes() function was used with argument method = 'complete'. In the default where method = 'simple', the second cluster only contains one element, so it is not a useful result to compare in this case. It is apparent from the two distributions, that the group sizes are quite different, with group 2 being around three times smaller after using pam(). In total, there are 24 out of 25 possible elements which overlap in group 2. So, despite there being a somewhat large difference in the sizes of the groups, it is interesting that the closeness is quite high. However, it is also difficult to comment on the difference in the size of the groups. It seems that it would be difficult to make an accurate analysis based on this dataset alone. It is possible that with a different sort of variable, the agnes() and pam() methods will make more or less accurate predictions based on clustering.

7. So far in preparing the data, there have been many different techniques that have been used by the students in the class. If we were using our own data for the current assignment, it is likely our results would all be extremely different. The different ways of filtering the descriptions and creating weighted character vectors was unique for everyone, so in this regard the preparation up until clustering was quite subjective. We all used a similar set of tools from the NLP toolkit to filter words down to their stems. However, even if we had a similar goal in mind, it is possible that we all did it with a different perspective on what parts of words are adequate in describing an expense. If clustering is done in such a manner, where the data requires significant preparation, then clustering itself must also be considered a subjective task. The clustering results in this assignment is only unique due to the data being the same for all students.

References:

https://cran.r-project.org/web/packages/Matrix/vignettes/Intro2Matrix.pdf
https://en.wikipedia.org/wiki/Sparse_matrix#Compressed_sparse_row_(CSR,_CRS_or_Yale_format)
http://www.john-ros.com/Rcourse/sparse.html
http://netlib.org/linalg/html_templates/node92.html
https://www.youtube.com/watch?v=Lhef_jxzqCg
https://www.rdocumentation.org/packages/Matrix/versions/1.2-15/topics/sparseMatrix
http://www.johnmyleswhite.com/notebook/2011/10/31/using-sparse-matrices-in-r/
http://web.mit.edu/r/current/lib/R/library/Matrix/html/matrix-products.html
http://astrostatistics.psu.edu/su07/R/html/base/html/diag.html
https://oig.justice.gov/
https://oig.hhs.gov/
https://www.fcc.gov/general/office-inspector-general-investigations
https://www.hhs.gov/about/agencies/asa/about-asa/index.html
https://piazza.com/class/jqmje0ujwrm2wx?cid=183