Dennis Simon, SJSUID: 007742215, Rank: 36, NMI: 0.0527

3. My approach:

The DBScan pseudocode is listed further below, I will first explain all my approaches/methodologies. First I will go over my approach for importing the input data/CSR matrix provided. My approach was very similar to activity data 3 which we completed earlier in the semester, there were some key differences though in my implementation. First off, in the activity we were reading in the actual document with words and not just data for word count of each document in the document-term sparse matrix format. So to adjust to this I had to change how the build matrix function worked. Nrows was equal to the length of docs because each line from the read in file was a doc and thus the docs array had 8580 lines and 8580 documents. Next I set the number of columns equal to the biggest feature id of a word from any of the documents in the file. I did this because the number of feature columns must be equal to the highest feature column value since the highest column/feature value should be the highest number of possible features. While doing this I skipped over the odd values in each document as they represented the value count for each word feature. After that I got my ind, val and ptr arrays for creating a CSR matrix using the creation method. Originally I retrieved every ind by setting it equal to the value of the feature in each document/line of the file(feature was always in the even placeholder) and I retrieved the value at the same time by setting equal to the number after the feature(d[w+1]) because the number of occurences always followed the feature in the odd placeholder. I ended up having to change this as the values of the ind/features were far too high b/c the original keys in document are very large(6-7 digits long). I did this by creating a dictionary to track unique values/feature ID's and incrementing the dictionary if the feature id wasn't present, if it was I used the feature ID earlier stored in the dictionary along with the value for the feature on the current document. Finally I created the sprase matrix using the ind, val, and ptr types as mentioned before.

After this I decided to perform dimensionality reduction which turned out to be a grueling and very long process. I first tried PCA on the csr_matrix but wasn't able to as PCA is not applicable on sparse data types, following this I tried TruncatedSVD which is a sklearn variant of PCA. I had many memory issues because as I mentioned earlier the original feature values that were stored in ind array for CSR matrix creation were extremely large and caused me to have kernel crashes no matter what I tried until implementing the dictionary solution in matrix creation. Next I tuned the parameters of TruncatedSVD on my CSR matrix by setting a required explained variance of 90%(.90) and calculating the total variance of the model as the number of components increased until the total captured variance was greater than the required explained variance, the number of components ended up being 1292. Finally I reapplied the TruncatedSVD on the CSR matrix using the tuned parameters.

Finally came implementing the DBScan. Since we were not allowed to use any libraries outside of the basic ones (scipy dot product, math, np.array, etc. based on Rathna's email) I first a created distance function so that I could calculate Euclidean distance because I could not use any predefined libraries to create a distance matrix. The formula I followed was based on Newton's method/Babylonian method which I found on Wikipedia. This method is also similar to sklearns implementation and is very efficient in calculations and is done by calculating the dot product of the first parameter by itself summed with the dot product of the second parameter by itself and finally subtracted by two times the dot product of both parameters. The nice thing about it is it scales based on the number of features provided to it in each parameter.

After this I created a function to check if two points in a n dimensional space were in epsilon (passed parameter) of each other. I did this by using the distance function created prior on the two input vector parameters and checking if it was less than or equal to the epsilon parameter passed to the function based on the provided guidelines from the lecture slides.

Following this I created a function, core_points, to find all the core points within the feature reduced CSR matrix(or any matrix passed to the function). This function took a matrix, epsilon and minpts and parameters because those are the only 3 required parameters for DBScan. In core points I created an array of type bool that I would use to store if a point was a core point or not(True = core, False = border/noise) and it would maintain order because it was based off the CSR matrix index. I used arrays instead of lists for the core/border/noise point list because it is more efficient. First of all I created for loop in the range of the length of the provided matrix for the point I was checking to be a core point then another for loop of the same range in order to check other elements of the matrix to see if they were in eps of each other. The second loop would run continuously for each point in the csr until the number on min pts(based on the parameter) were found within epsilon of the point being tracked by the first loop at which time the point was added to the index array as true(core point) based on its index to retain order. If the number of minpts was not achieved within epsilon then the point was set to false(not a core point). Finally I called another function, border_points, inside the core_points function using the matrix, calculated array of core/noncore points for each index, and epsilon.

The border_points function essentially added to the index array of core/noncore points by going through every non core point at its index and checking if it was in the epsilon neighborhood of any core point. If it was then it was said to 1(1 represented border point on the index) and if it wasn't it was set to 2 which represented a noise point. Obviously every core point from the previous function was retained in its index. The final array containing which points were core, border or noise based on its index equivalent to the reduced CSR matrix was returned.

After this I called a function to connect the core points based on if they were in epsilon distance (Euclidean scale) of each other by using the distance function I created prior. I passed as parameters the matrix, my created array of core/non-core points and epsilon. I created an empty dictionary that would I later use to store each cluster/connected component as an array of points(the index of the point on the matrix at least) for each component(key in dict), the dictionary keys were integers(0-x) representing the cluster/component. First I created a for loop to go through the range of the length of the matrix(8580 rows so 8580 iterations), inside of it I created a condition by using an if statement to check if the point at the current index of the for loop was a core point on my created core point array/list. If it was not(border/noise) then it would not be added to the connected components. Then I added the first core point found to a dictionary. Then I checked if matrix row at the current for loop was within epsilon distance (using my created function) of matrix row of the index point for the current component of the cluster, if it was I appended the index value of the point to the value of this component key in the dictionary. If it wasn't then I added a new component(new key) by incrementing up 1 for the keys. I did this for every core point.

Next I tried to connect the core points together, because core points within epsilon of each other are supposed to be connected to each other, but I was unable to do this properly as I had issues appending lists so I left this out. This was the reason my NMI score was so bad as the core points were not connected completely thus creating high amounts of clusters.

Finally I added the border and noise points to the components based on if they were in the neighborhood of a core point in any of the components in my dictionary. If a point on the earlier created array/list of core/border/noise points was a border point then I checked if it was in epsilon distance of any of the core points in a component, if it was I added it in to that as a component and if it wasn't then I checked the next component. Logically(and practically after testing) every border point was assigned to a component/cluster as they were all previously already calculated to be within epsilon distance of core points. I then assigned any noise points to the K+1 cluster. I did this by calculating the max value of my keys(which represented clusters from 1 to x) and adding 1 to it then using this to create a new dictionary pair where the key was the max value of keys + 1 and the values were all of the remaining noise points.

This was the implementation of my DBScan algorithm which returned a dictionary of clusters with the keys being incrementing integers and representing different clusters and the values being the points from the CSR matrix appended into a list for each component. Basically the dictionary value was a list of indexes of each point for each cluster. Following this I used basic pandas dataframe logic to export my clusters to a CSV, please note I did not use this in DBScan implementation, only to export the dictionary of clusters to a CSV.

Pseudocode:

```
Core_points:

For I as 0 to length of matrix
    For j as 0 to length of matrix
            Continue if i=j
            If mat[i] = mat[j] minpts amount of times add to core point list as core
            Else add to border/noise list
Return core_points list

Border_points:

If point is not true in corepoints list
    For I as 0 to length of matrix
          If mat[i] is within epsilon of another point and other point is core
            Set as border point on corepoint list
        Else set as noise point
    Return core point list
```

Connect_comp:

```
If point is core on core list
Loop for 0 to length of
    If core points in range of each other append to dict value for key as loop index
    Else add new cluster/key and store core point here
Add_border/noise_points
If point is border on border/core/noise index list
    For each list of each component in dictionary
        If point in eps of element in list
            Add border point to list
        Else check next component … forever
If point is noise point add to K+1 cluster/dictionary key
```

4. Determine radius eps for minpts varying from 3 to 21:

I did not do this as I ran out of time.

5. Implement internal evaluation metric:

I did not do this as I ran out of time.

6. Describe any feature reduction used:

I used TruncatedSVD to reduce the number of features of the originaly CSR matrix. I did this because the original matrix had a huge amount of features and computing distance caluclations(Euclidean) for each feature for every row would be inefficient and too memory taxing for my system. I used TruncatedSVD because it is an altered version of PCA for the sklearnlibrary that can be implemented on sparse data types which was perfect for me as I had to use a sparse data matrix (CSR) over an array in feature selection otherwise I had memory issues and kernel failure. I tuned the parameters of TruncatedSVD on my CSR matrix by setting a required explained variance of 90%(.90) and calculating the total variance of the model as the number of components increased until the total captured variance was greater than the required explained variance, the number of components ended up being 1292. Finally I reapplied the TruncatedSVD on the CSR matrix using the tuned parameters.