Emma Peatfield

009192534

Rank: 11 (at time of writing report)

NMI: 0.4746

Program 3 – DBSCAN Text Clustering

Clustering is a very useful approach to placing similar objects within the same groups for easy classification. DBSCAN is a clustering technique based on the density of a set of points in a space. It stands for density-based spatial clustering of applications with noise. Given a set of points, DBSCAN will enable one to place these points in clusters based on the density of certain regions. This density can be found by defining a distance value, epsilon, and a neighborhood range, defined by the minpoints. Epsilon will define the radius of the neighborhood for a point. If other points lie within this radius, they are considered to be neighbors of this point, and if there are more neighbors than the minpoints value, then the original point is considered a core point. Each cluster must have at least one core point, and the rest can be considered as border points. Within the dataset, there might also be noise points, which do not belong to any cluster, and are just outliers.

There are quite a few ways to implement DBSCAN and, of course, there are packages available. However, we were not allowed to use any packages to implement this technique, only packages for preprocessing the data. The data that we were given was in sparse matrix form, so we had to create a CSR matrix from it. First, the file was read and parsed, since each line was a long string; each separate value was then changed into an integer. Then, looking at the data, I traversed through the lists and for every line I added to three different arrays. Values, Indices, and Indptr are three arrays that I created from the indices of the words (i.e. every even index on each line), the values of that word (i.e. the odd index values), and the index in values/indices where each document, or line, started and where each one stopped. Once obtaining those three arrays, I was able to combine them into a CSR matrix.

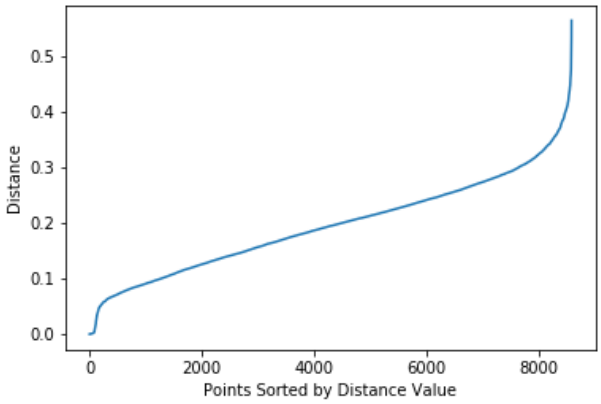
From our in-class Activity Data 3, I used the l2 normalization code so that I could normalize this CSR matrix. This would make it much more reliable in my future algorithm. I also looked into some feature selection/dimensionality reduction, but I will discuss that later on. Then, to finish off processing my data, I computed a similarity matrix of the cosine similarity between each of the documents within the dataset. From there, I was able to start my DBSCAN algorithm. I started by writing the steps of what needed to be done in order to cluster these points. Here is an example of some pseudo code that I planned out in the process of writing this algorithm:

```
def dbscan(matrix, eps, minpts):
        #initialize cluster count, labels array, and type array
        for entry in matrix: #search each documents similiarity comparison
                for ind, value in entry: #search similarity b/w documents and index
                        if value is within eps: add to neighboring pts array
                if no neighbors, label as type: Noise and assign to its own cluster
                else:
```

check for border points, core points, and more noise
check if the label (i.e. cluster value) is nan for this point
    if so,
        check neighbors if also all nan then create cluster
        if some nan, some not, look at the most named cluster and use that label
    else, set unlabeled neighbors to that cluster

As you can see, I had a broad sense of where I wanted to go with this algorithm. I have two extra arrays, one for labels and one for types (i.e. Core, Border, Noise) and I used the minpts value to help decipher those labels. My function takes 3 inputs, a similarity matrix which is computed before the function starts, an eps value for the radius distance, and a minpts value to help define core points, noise points, and border points. I use the eps value to help scope out the neighbors of each document to discover which are the closest and how many there are. Basically, this function just iterates through the entire training set data that is within a similarity matrix by this point to search for each documents' neighbors, and group them into clusters.

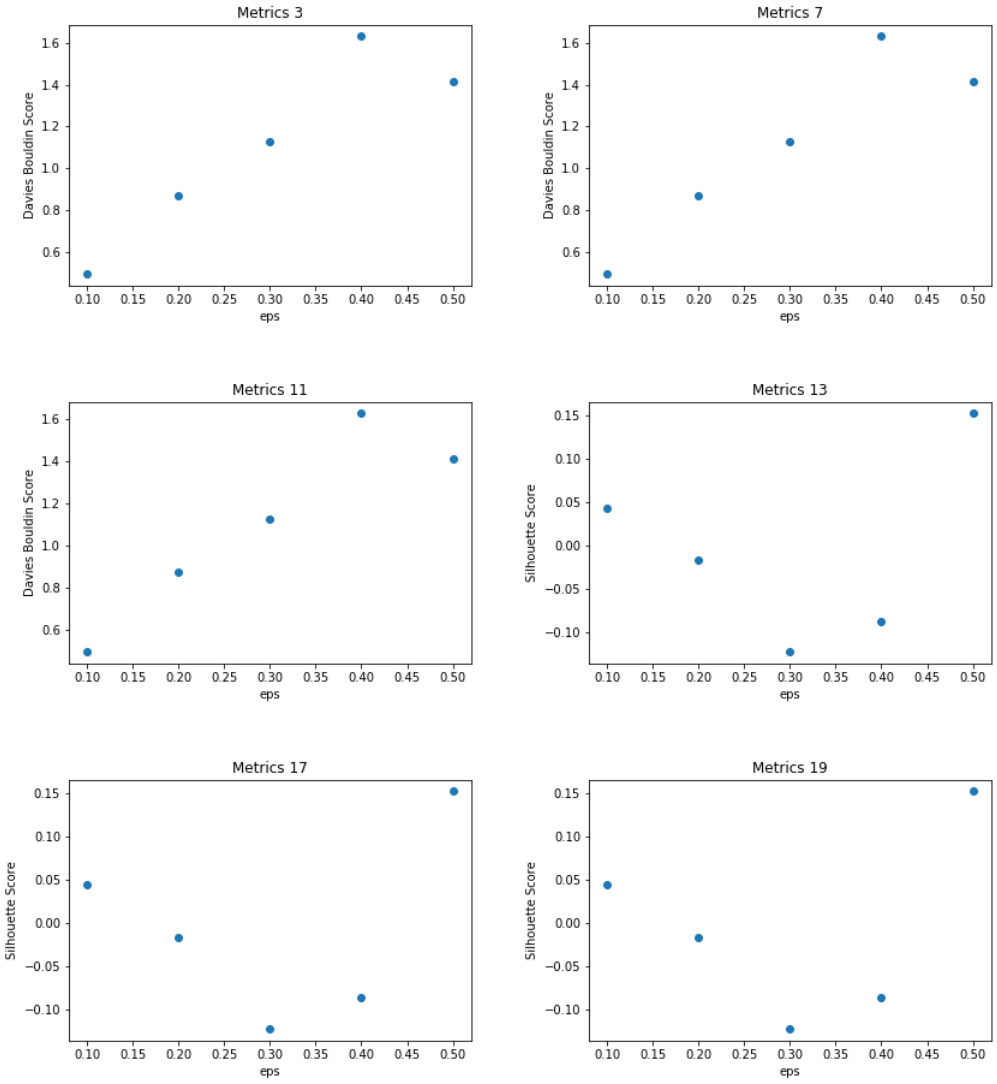
Creating this function took a lot of time and thought. When I first read the assignment, I wasn't sure where to start. I took it slow at first by just starting with processing the data. Then I wrote many drafts for a pseudo code for the algorithm. Once I had a clearer direction for my code, I was able to start implementing it. It didn't work at first, so I had to make a few changes, but as soon as I was able to get labels for my clusters, I turned it in to run on CLP and was quite pleased with my result. I think taking the time to hand-write all of my code and run through it over and over really helped me to create an efficient algorithm that produced good results.



The plot above was created after I had turned in my code to CLP, but it is a plot of the distance to the kth point for each point. The kth point means that if minpoints is 3, it would be the 3rd closest point for each point in the cluster. In looking at this plot, you can see an upward curve that hits an elbow. That elbow represents roughly where the noise points begin since the noise points will have the farthest distances to their kth point.

One more thing that I had yet to play around with was how the clustering results differentiated when using different eps and minpts values. For this, I started testing different

metrics on my code. I looked at the Davies Bouldin Score and the Silhouette Score the most. For each minpts value from 3-21 in steps of 2, I plotted the values in various plots using pyplot. Davies score was supposed to have the lowest score mean that that was a better eps value, and for Silhouette, the value could be between -1 and 1, accurately. Here are a few of those plots.



In the figures above, you can see what a few of my metric plots look like while using the Silhouette score for different values of eps and what a few look like when using the Davies Bouldin Score for different values of eps. Using these metrics, results didn't seem to vary much with the different minpoints values that I used, which might be due to a coding error, but I'm not entirely sure. It's also interesting to notice that for the Silhouette Score, a high eps value proved to be the best choice, while for the Davies Bouldin Score, a lower eps value is shown to give better results. Maybe it was variant for my code, but it's an interesting differentiation that I didn't think would be the case.

With this assignment, we not only had to consider the value of eps and minpts that we used, but we also had to consider using dimensionality reduction to avoid the curse of dimensionality. For this, I used Truncated Singular Value Decomposition to decrease the number of features that were being used. This seemed to drastically improve my code. I chose truncated singular value decomposition because it works well with sparse matrices, which is where my data started. It works efficiently with sparse matrices because it does not center the data beforehand. However, it does return a dense array, so I did have to transform the output back to a csr matrix.

This assignment was challenging in that we had to write our own clustering algorithm, but I'm glad that I got to do it. Most code these days can easily be written using packages, but it's always a good challenge to know how to code things yourself. You never know when you might need the knowledge and skill to write your own algorithm one day.