ISYE 6740 – Homework 2

Nicholas Chao – [nchao3@gatech.edu](mailto:nchao3@gatech.edu)

**Part 1.**

(Code for implementation is submitted in separate files, my\_sol\_kmeans.py and my\_sol\_kmedoids.py)

1. There were three primary choices I made in my implementation of the K-Medoids algorithm. Those were the selection of cluster representatives each cluster, the distance measures I tried and chose, and when iterations were stopped.

To determine the representative datapoint for each cluster, I selected the datapoint closest to the geometric mean, measured by the Euclidean distance. I selected this due to its speed and simplicity. Many descriptions of the K-Medoids suggest selecting the datapoint with the lowest average distance from all other points in the cluster (i.e. the PAM implementation). This is however computationally expensive, with computation time growing on an exponential order. For this reason, I elected to the less expensive alternative of choosing the point nearest to the geometric mean of the cluster.

I tried both Euclidean distance as well as the L1 distance. The L1 distance would be less sensitive to outliers as absolute distance in Euclidean is squared, making outliers more significant. The better choice is dependent on the use-case.

I had 2 stopping criterion for the algorithm, and the algorithm stops when either is reached. The first criterion is if 300 iterations are met. Because K-medoids will stabilize after a certain amount of iterations, this criterion is meant as a fail-safe if oscillation occurs. It is also an adjustable parameter in my function. The second criterion is when the Jaccard similarity between the label assignment vector of the current iteration and the previous iteration is equal to one (e.g. the algorithm converges and cluster assignment no longer changes). Alternatively, a convergence tolerance could easily be implemented (e.g. the Jaccard similarity surpasses some threshold close to 1).

1. I used the following image:



1. I found with very large values for K, running time for each iteration tended to be longer and more iterations were needed to reach convergence. Additionally, with a large value of K, it was often the case that some clusters would contain no pixels, and so K would have to be reduced. This reduction in K for empty clusters was automatically done in my implementation and the user is notified when it is the case.
2. I found that the initial centroids did not impact the final cluster assignment.
3. Similar to in K-Medoids, a high value of K led to longer iteration times, often more iterations needed for convergence, and empty clusters resulting in an automated reduction in K.

**Part 2. (Spectral Clustering)**

1. Consider connected components in an undirected graph with positive weights. I’ll note that the Laplacian matrix can be considered as a representation of the sum of differences between each vertex in the graph to all other vertices (when applied to a given vertex of the graph). This representation can be written as:

Now consider the eigenvectors of the graph that correspond to transformations from each connected component. Because the components are independent, these eigenvectors are clearly orthogonal, and because

We know that these eigenvectors correspond to eigenvalues of 0 for that connected component, and there are such eigenvectors. The other eigenvectors (which correspond to eigenvalues greater than 0) hold the weights towards other components and are there are only of these 0 eigenvalues.

1. I calculated a false classification rate of 49% when applying my spectral clustering implementation on the political blogs dataset. Because this dataset was essentially balanced in classes, this is a near-trivial classifier. This suggests that further modifications of the data would be needed or an alternative classification algorithm would be better suited for this task. (See attached spectral\_cluster.py script for my code).

**Part 3.**

Here are the first two principal components plotted (for both questions 1 and 2):



The primary observation is that there appears to be little to no correlation between the first two principal components; they are orthogonal. Additionally, we can see the range of the first principal component (the x-axis) is much greater than the second principal component (the y-axis), and so the first principal component holds a much greater amount of the variation in the observed data than the second. This is what PCA is designed to achieve. Each principal component is capturing as much variation of the data as possible, and each principal component is orthogonal to the next. (see PCA.py for my implementation).