Middle East Technical University Department of Computer Engineering

CENG 499

Intro to ML

Fall 2024-2025

Homework 2

Name SURNAME: Aly Asad GILANI
Student ID: 2547875

Part 1

IMPORTANT NOTE: I could not import Distance.py when it was outside the "Part1" folder, so I put the python file inside the folder. My code assumes that Distance.py, Knn.py, and Knnexperiment.py are all inside the Part1 folder.

For this part, I started by writing the code for Distance.py, which was easy as the formulas were given in the PDF. Then I worked on Knn.py. For this, I compared the input instance with every datapoint in the test dataset and kept a list of its K closest points. In the end, I took the majority labels (mode) from that list and returned it.

Lastly, I worked on Knnexperiment.py. Since I did not know how the scikit library worked, I first watched the recitation and tried out different codes on the scikit website. For my code, I iterated over my hyperparameters, printing each configuration. To setup the scikit's "RepeatedStratifiedKFold", I set "n_splits = 10" for 10 folds and "n_repeats = 5" to test each configuration 5 times with randomized input data/folds. Since the scikit's function was already randomizing the data and folds, I did not need to randomize it further using numpy.
Then, for the cross validation, I found 5 accuracies for each configuration, each one corresponding to a randomized dataset. To find the accuracies, I used scikit's provided folds with my previously defined KNN class/functions. Then I calculated the mean accuracy and confidence interval for the current hyperparameter configuration, and printed it. In the end, I printed the best mean accuracy from the 12 configurations as well. Note: I could have made "p" in Minkowski distance a hyperparameter as well, but since there were already so many configurations I decided not to.
In the beginning, my hyperparameters were:
K = [1, 3, 5, 7]
Distance functions = [Cosine, Minkowski, Mahalanobis]

However, from my tests, I was getting K = 7 as the best K, which meant my K range was too small. So I swapped K = 3 with K = 9 for configuration. After that, I was getting K = 7 as the best K as well, which meant that the best K is between 5 and 9. My final hyperparameters were:
K = [1, 5, 7, 9]
Distance functions = [Cosine, Minkowski, Mahalanobis]

The hyperparameter configurations and their confidence intervals is attached:

```
carnifex@Carnifex:~/Desktop/499/Part1$ python3 Knnexperiment.py
1) Testing hyperparameters: K = 1, Distance function: Cosine
Confidence interval: 91.73 ± 0.793

2) Testing hyperparameters: K = 1, Distance function: Minkowski
Confidence interval: 89.07 ± 0.467

3) Testing hyperparameters: K = 1, Distance function: Mahalanobis
Confidence interval: 86.67 ± 0.370

4) Testing hyperparameters: K = 5, Distance function: Cosine
Confidence interval: 95.07 ± 0.467

5) Testing hyperparameters: K = 5, Distance function: Minkowski
Confidence interval: 90.13 ± 0.573

6) Testing hyperparameters: K = 5, Distance function: Mahalanobis
Confidence interval: 89.33 ± 0.978

7) Testing hyperparameters: K = 7, Distance function: Cosine
Confidence interval: 95.60 ± 0.286

8) Testing hyperparameters: K = 7, Distance function: Minkowski
Confidence interval: 89.20 ± 1.005

9) Testing hyperparameters: K = 7, Distance function: Mahalanobis
Confidence interval: 88.80 ± 0.935

10) Testing hyperparameters: K = 9, Distance function: Cosine
Confidence interval: 95.47 ± 0.681

11) Testing hyperparameters: K = 9, Distance function: Minkowski
Confidence interval: 89.33 ± 0.370

12) Testing hyperparameters: K = 9, Distance function: Mahalanobis
Confidence interval: 87.73 ± 0.949


Best configuration: K = 7, Distance function: Cosine
Confidence interval: 95.60 ± 0.286
carnifex@Carnifex:~/Desktop/499/Part1$
```

Figure 1: Hyperparameter configurations and the confidence interval of their accuracies

Part 2

For this part, I started with K-means first since it wasn't related to the dimensionality reduction section. I started my code in Kmeansexperiment.py. For each dataset, I looped from K = 2 to K = 10, and inside the loop I looped over 10 times to find the average and confidence interval of each run. To run the K-means algorithm, I used sklearn.cluster's KMeans function, in which I set "init = k-means++" for better initializations and so the program converges faster, I also set "n_init = 10" so the function can run K-means 10 times and only give me the minimum inertia score and corresponding labels. Then, I found the mean and confidence intervals for both the loss and silhouette scores for each K value and printed them, then plotted them. The confidence intervals represent the vertical errors for each point in the graph. For the first dataset, I have attached the K-means loss and silhouette graphs:

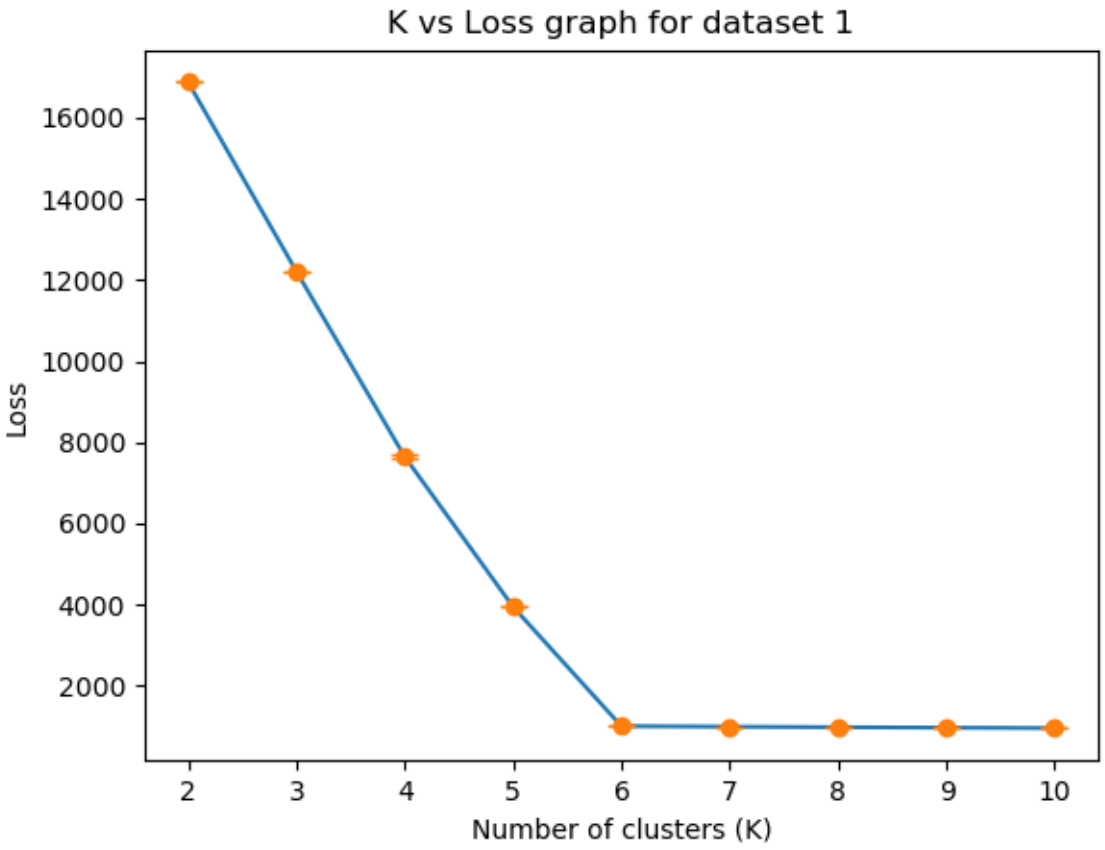

Figure 2: K-means K vs loss, dataset 1

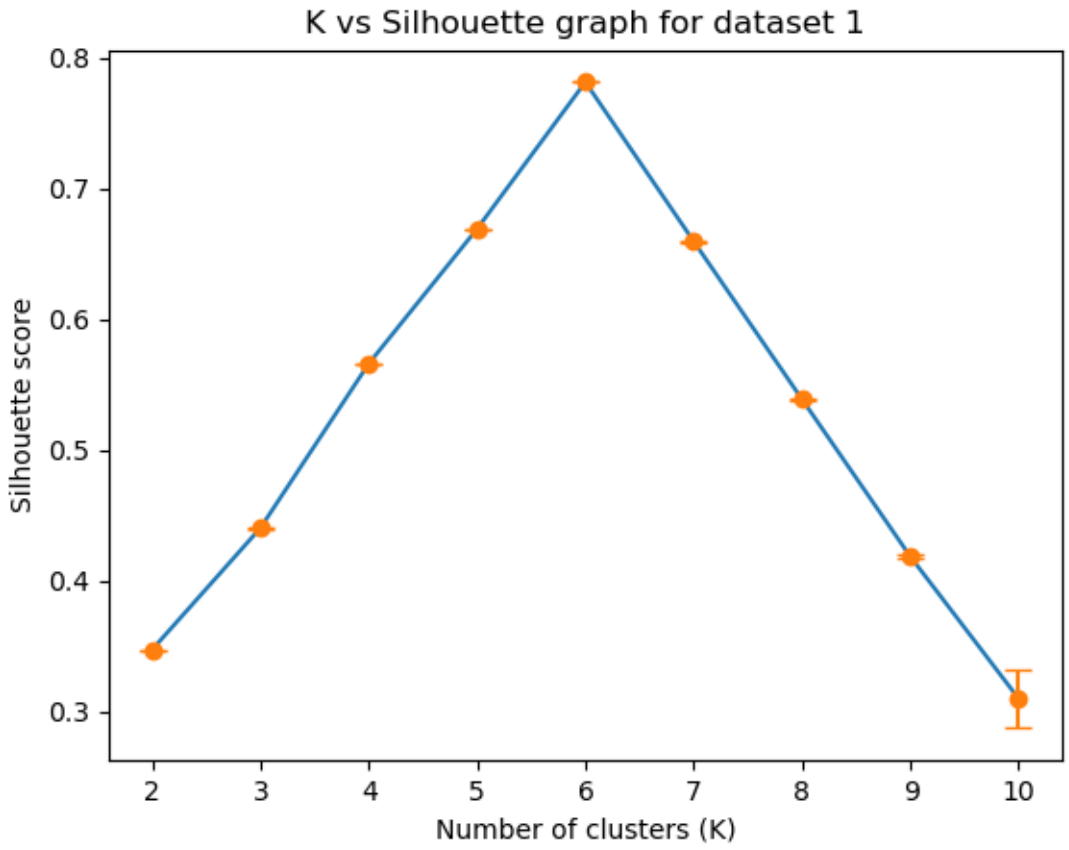

Figure 3: K-means K vs silhouette, dataset 1

We can use the elbow method on the loss graph and see quite clearly that the optimal number of clusters for dataset 1 is 6. This is reinforced by the silhouette graph, as the max silhouette value is also at cluster number 6, which means that for K-means and the first dataset, 6 is most definitely the optimal number of clusters.

For K-means with the second dataset, the graphs are:

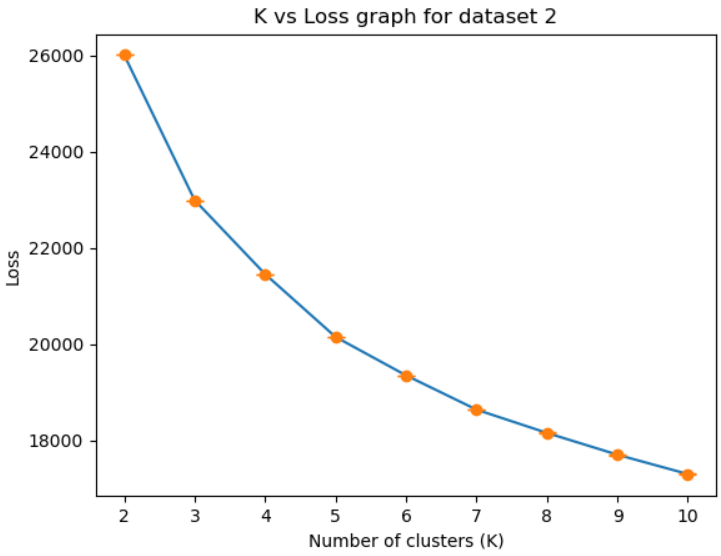

Figure 4: K-means K vs loss, dataset 2

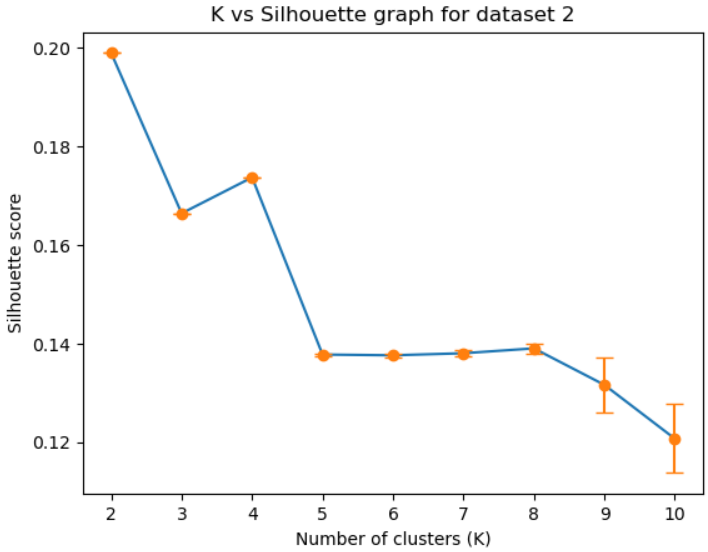

Figure 5: K-means K vs silhouette, dataset 2

We can see that because of the extremely high number of dimensions/features of the second dataset (784 features!), the K-means clustering algorithm just cannot make meaningful clusters because of the curse of dimensionality. Also, it could be the case that dataset 2 just does not have cluster-able data.

For the K-medoids algorithm, most of the code was the same, the only things to change was the function call to sklearn_extra.cluster's KMedoids function, and I had to add another loop to find the minimum inertia from 10 runs since the function call did not support the "n_init" input. For the first dataset, I have attached the K-medoids loss and silhouette graphs:

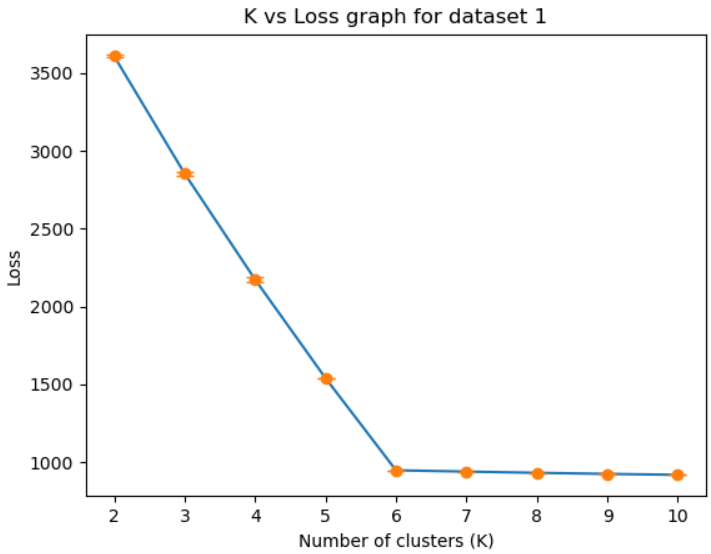

Figure 6: K-medoids K vs loss, dataset 1

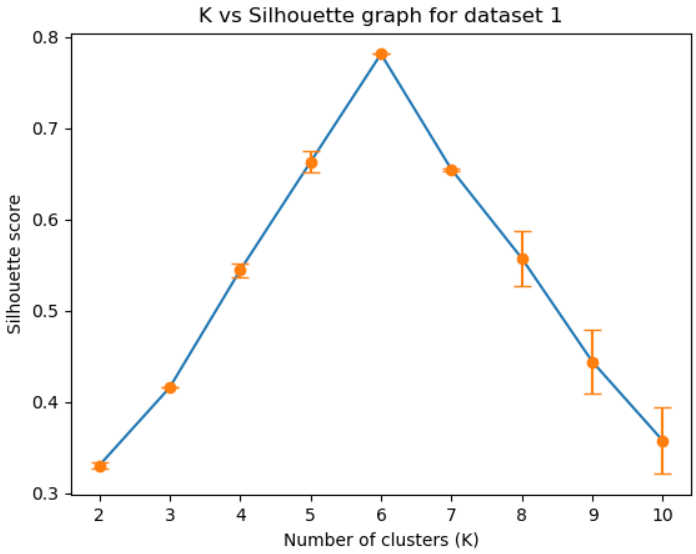

Figure 7: K-medoids K vs silhouette, dataset 1

We can use the same methods as for K-means and see that using K-medoids, the optimal number of clusters for the first dataset tend towards 6, which confirms our results from K-means as well.

For K-medoids with the second dataset, the graphs are:

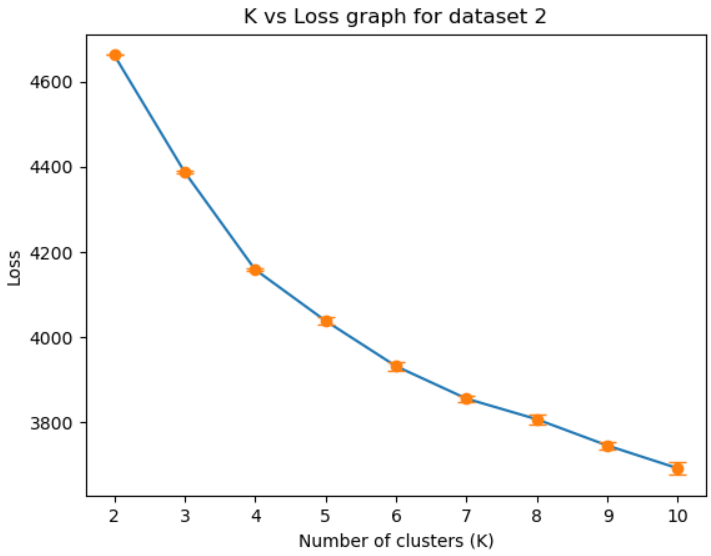

Figure 8: K-medoids K vs loss, dataset 2

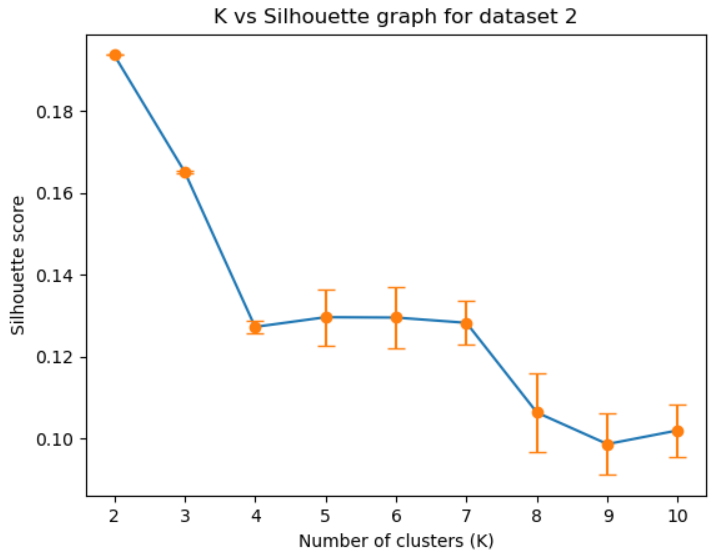

Figure 9: K-medoids K vs silhouette, dataset 2

The results for the second dataset of K-medoids suffer the same consequence of the extremely high number of features as that from K-means, in that it cannot create any meaningful clusters for the second dataset.

In the end, the K-means and K-medoids algorithm gave the same results, but from a cursory glance at the graphs, the K-medoids algorithm gave less overall loss for each K value.

Moving onto the dimensionality reduction. First, I worked on the pca.py file, the concept was simple as I have studied projection matrices in linear algebra and computer graphics and it was easy to implement with basic numpy functions. After that, I moved onto autoencoder.py, which was also familiar from the first assignment. I only used 1 hidden layer with 2 nodes for the AutoEncoder, as it was sufficient.

Lastly, for the part2_dimensionalityreduction.py file, I called just my PCA and AutoEncoder functions as well as the TSNE and UMAP functions from their libraries and plotted their outputs. For the PCA method, there were no hyperparameters. For the AutoEncoder, initially I set the learning rate to 0.1 and number of epochs to 1000, but it wasn't converging properly and not giving a good graph so I experimented with both hyperparameters until I settled on learning rate of 0.001 and 10000 epochs. For the other two methods, I mostly just used the default hyperparameters. The graphs for dataset 1 are:

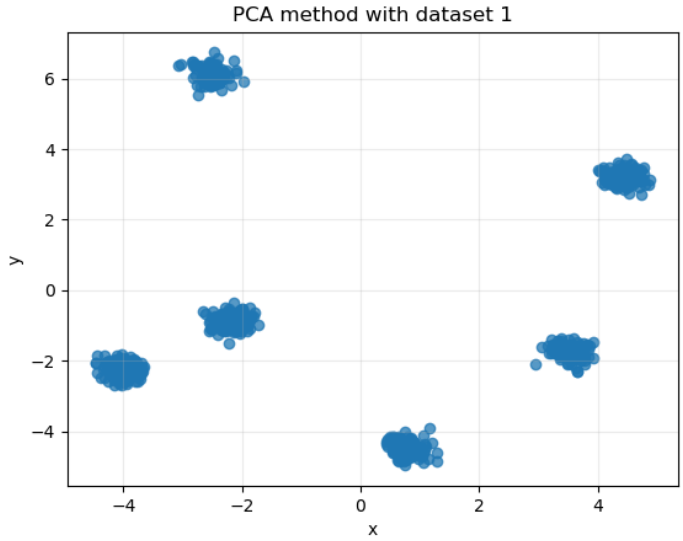

Figure 10: PCA dimensionality reduction on dataset 1

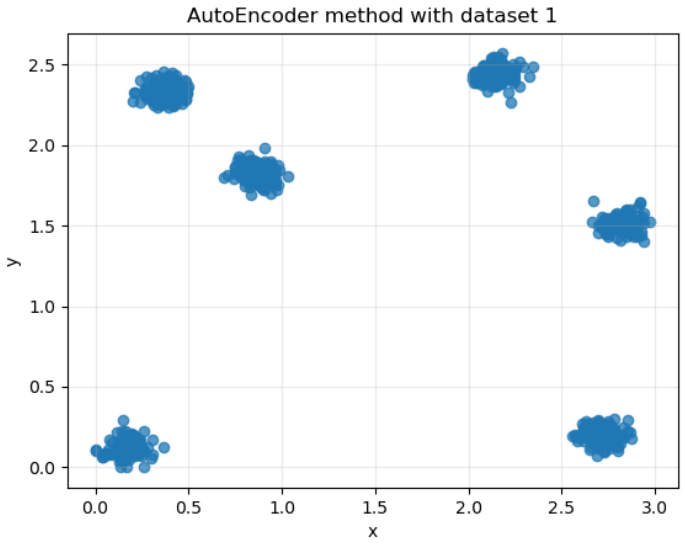

Figure 11: AutoEncoder dimensionality reduction on dataset 1

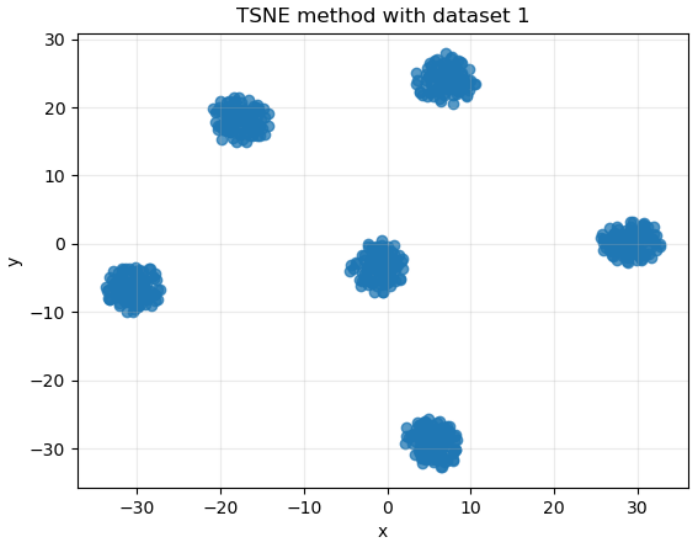

Figure 12: TSNE dimensionality reduction on dataset 1

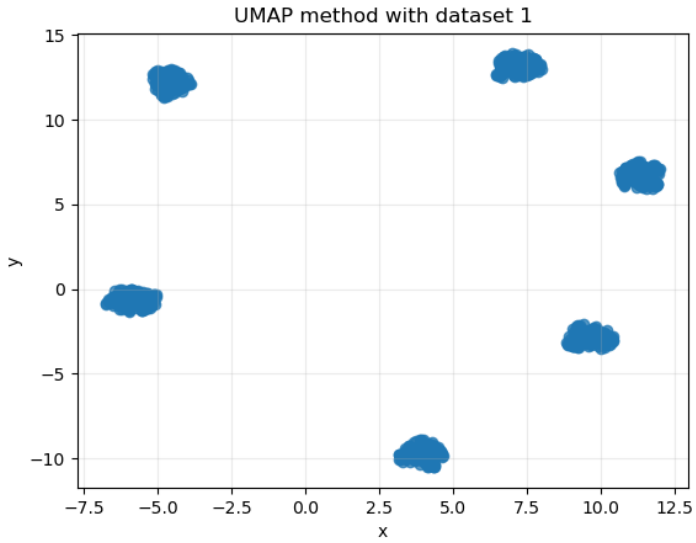

Figure 13: UMAP dimensionality reduction on dataset 1

As we can see, all four dimensionality reduction methods reduced the number of features to 2, and all made six different clusters on the 2D plane. This confirms our results from the K-means and K-medoids experiments that dataset 1 has 6 optimal clusters. As for the performance of the dimensionality reduction methods, TSNE performed the fastest, giving its result in less than a second, and AutoEncoder performed the slowest, giving its result in around two minutes. The other two gave their results in between 5 to 10 seconds. As for the quality of the clusters, UMAP gave the tightest clusters compared to their size, while TSNE's clusters were the most evenly spaced out. The other two methods created neither tight clusters nor even space between them.

Moving onto the second dataset, here the are the graphs:

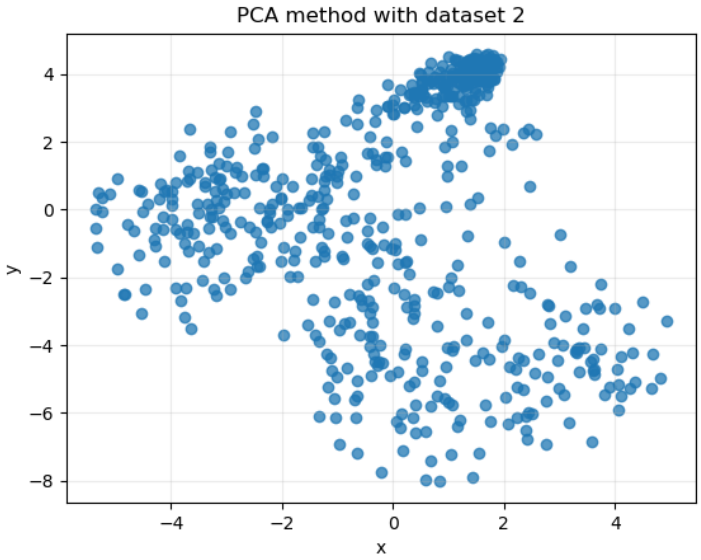

Figure 14: PCA dimensionality reduction on dataset 2

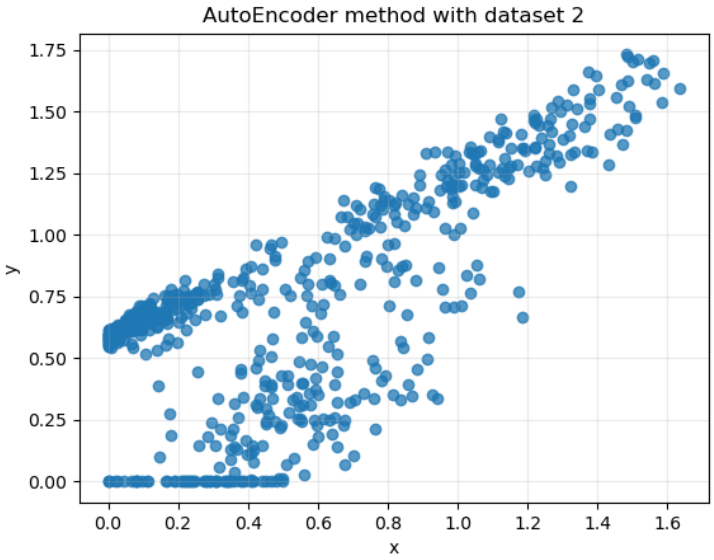

Figure 15: AutoEncoder dimensionality reduction on dataset 2

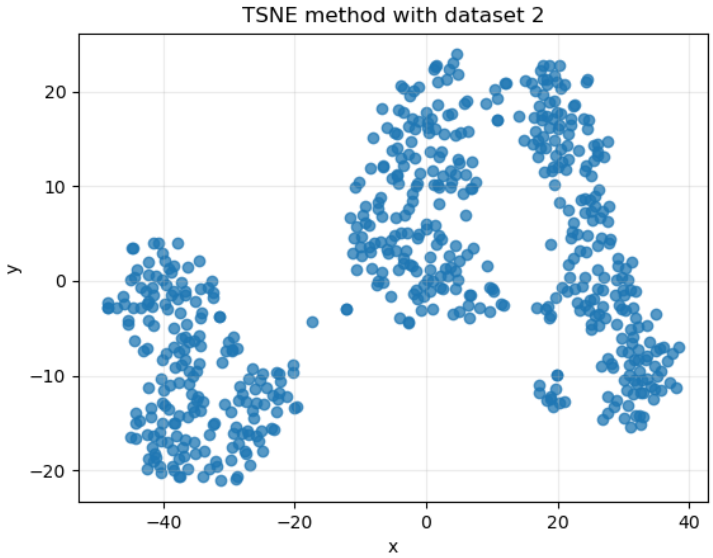

Figure 16: TSNE dimensionality reduction on dataset 2

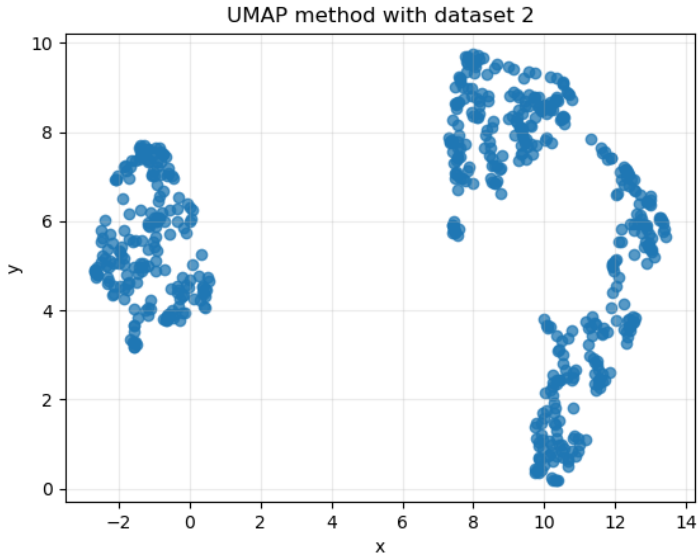

Figure 17: UMAP dimensionality reduction on dataset 2

As we can clearly see, no dimensionality reduction method could create and show any meaningful clusters for the second dataset. This result points to either of two things: either the second dataset just doesn't have cluster-able data, or 2 dimensions is not enough to represent all the 784 features of dataset 2. This result also correlates with our results from the K-means and K-medoids experiments, since they could not identify clusters for the second dataset as well.

Worst case analysis of K-means: For K-means, we perform two different tasks for each iteration: assigning each datapoint to a centroid, and then updating the locations of the centroid. For the first task, we compare each datapoint's (N) distance to each centroid (K) over the data's dimensions (d). So, the first task has worst case $O(N * K * d)$ time complexity. For the second task, we compute the new centeroids for each cluster. This requires iterating over all the datapoints. So the time complexity is $O(N * d)$. In total, for one iteration, the time complexity is the addition of the complexities of both tasks, which is $O((N * K * d) + (N * d)) = O(N * K * d)$. Now we multiply this result with the max number of iterations, and we get the worst-case running time complexity of K-means: $\boldsymbol{O(N * K * d * I)}$.

Worst case analysis of K-medoids: For K-medoids, we also have two tasks for each iteration: assigning each datapoint to a centroid, and then updating the locations of the centroids. The assignment task is the same as in K-means, so the time complexity is the same as well: $O(N * K * d)$. The second task is more complicated, for each cluster (K), we assume a datapoint as the centroid of that cluster (N/K assumptions on average), then find the distance of that datapoint to all other datapoints in the cluster $N/K * d$). So, the overall time complexity for this update is $O(K * (N/K)^2 * d) = O(N^2/K * d)$. For the worst case, we can assume that all data points are in one cluster, and the others are empty. So the worst case time complexity for this update would be $O(N^2 * d)$. Finally, we can find the overall worst case time complexity for the K-medoids algorithm to be the sum of the complexities of these two tasks, multiplied with the max number of iterations: $\boldsymbol{O(I * d * (N * K + N^2))}$.

Part 3

Moving onto the last part of the assignment, I started by writing everything related to HAC in "part3.py". First, I set the four hyperparameter configurations and looped over them. For each configuration, I ran the HAC algorithm using sklearn's "AgglomerativeClustering" class/function. Then, I plotted the dendrogram that resulted from that configuration (using the sample dendrogram code provided in recitation files). Then, for each K value from 2-5 for each configuration, I found the average silhouette values and made a list. Then, I plotted the average silhouette vs K graph for each configuration.

The results for each configuration, with best found K value and the corresponding silhouette value, is given below:

```
Running HAC:
HAC with hyperparameters: [single, euclidean]
Best K: 2 with value: 0.258

HAC with hyperparameters: [single, cosine]
Best K: 2 with value: 0.256

HAC with hyperparameters: [complete, euclidean]
Best K: 2 with value: 0.138

HAC with hyperparameters: [complete, cosine]
Best K: 2 with value: 0.083

#####
Best configuration:
[single, euclidean, 2], Silhouette value: 0.2582
```

Figure 18: Results of HAC

As we can see, $K = 2$ is the best K value for each hyperparameter configuration. The best configuration is Single Linkage, with Euclidean distance metric, with a Silhouette score of 0.2582.

The resulting dendrograms for each configuration is:

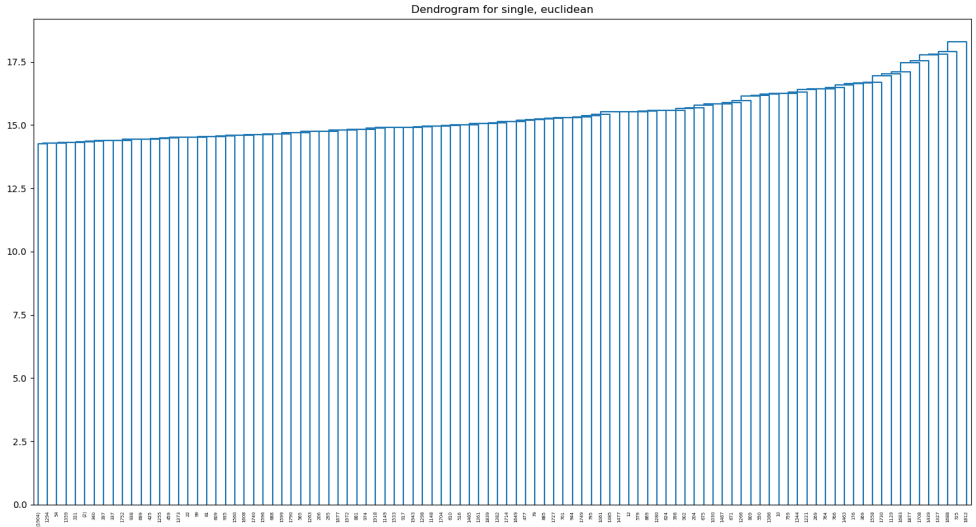

Figure 19: Dendrogram for Single, Euclidean configuration

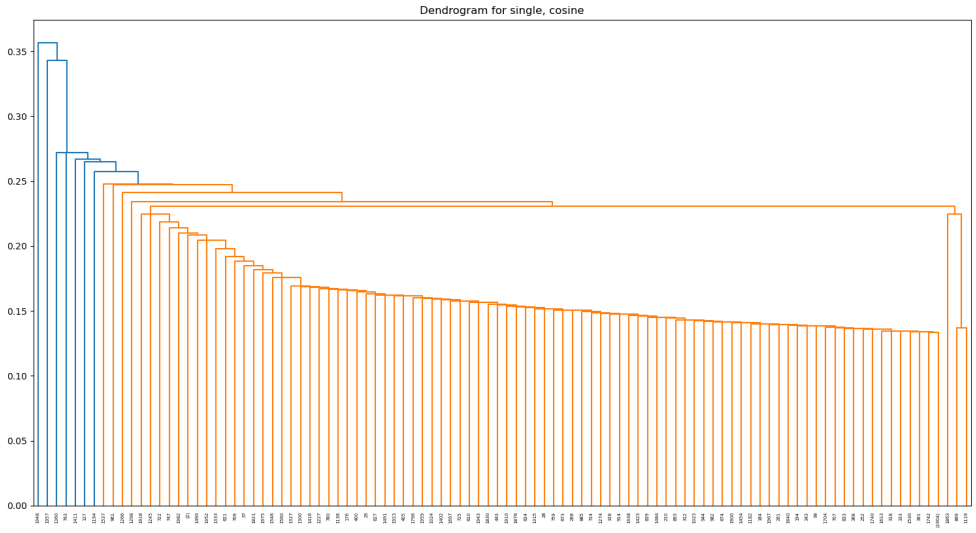

Figure 20: Dendrogram for Single, Cosine configuration

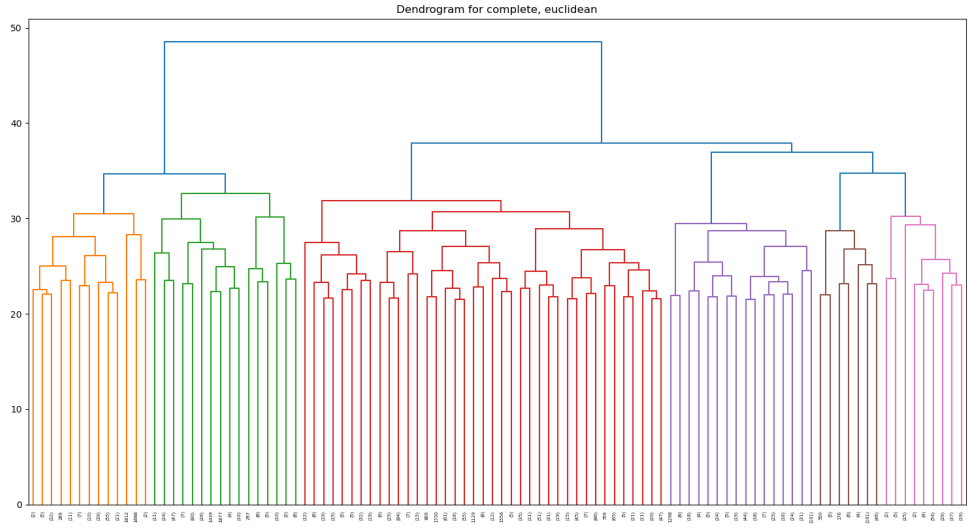

Figure 21: Dendrogram for Complete, Euclidean configuration

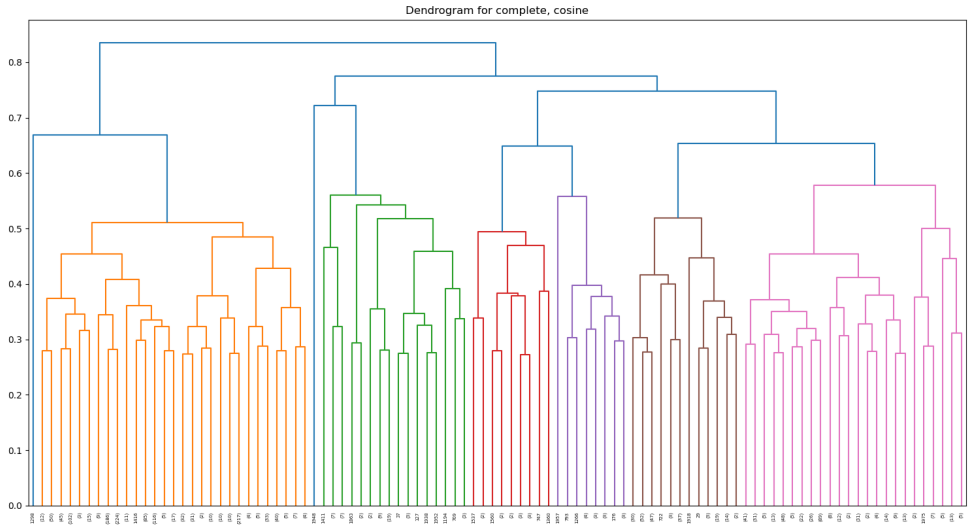

Figure 22: Dendrogram for Complete, Cosine configuration

For each hyperparameter configuration, the resulting silhouette value plots are:

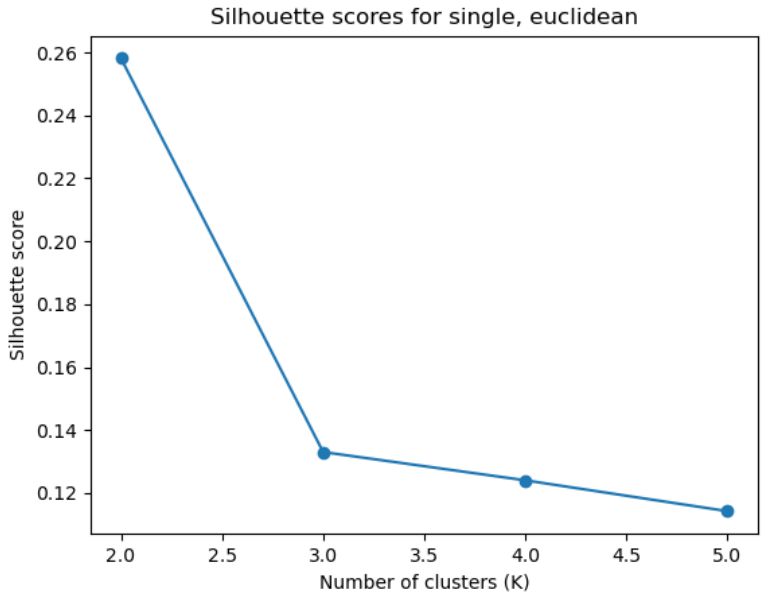

Figure 23: Silhouette plot for Single, Euclidean configuration

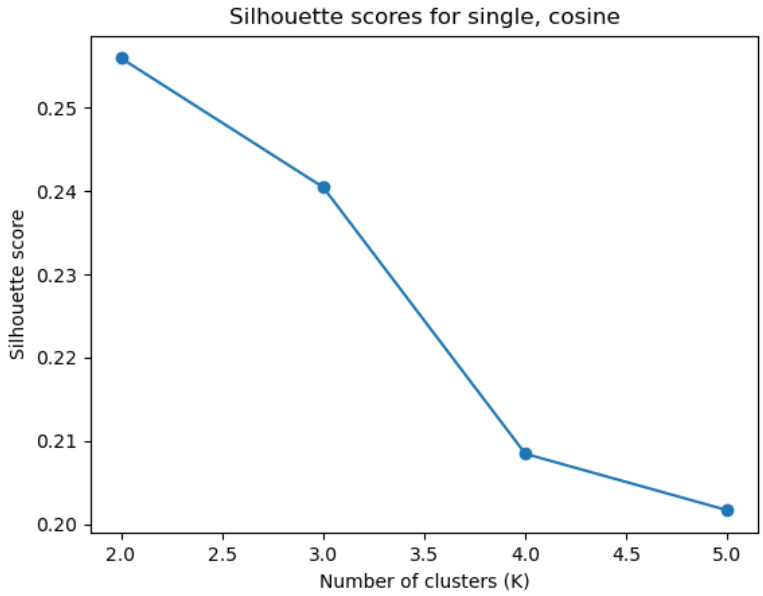

Figure 24: Silhouette plot for Single, Cosine configuration

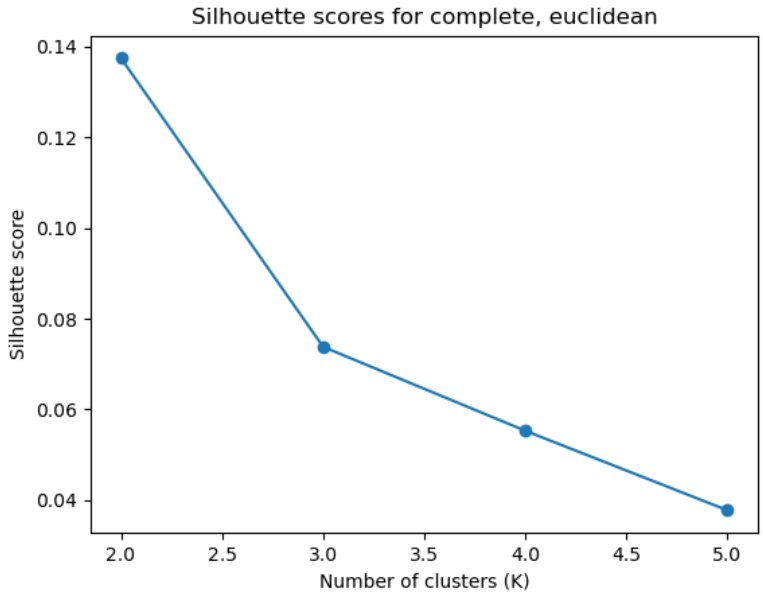

Figure 25: Silhouette plot for Complete, Euclidean configuration

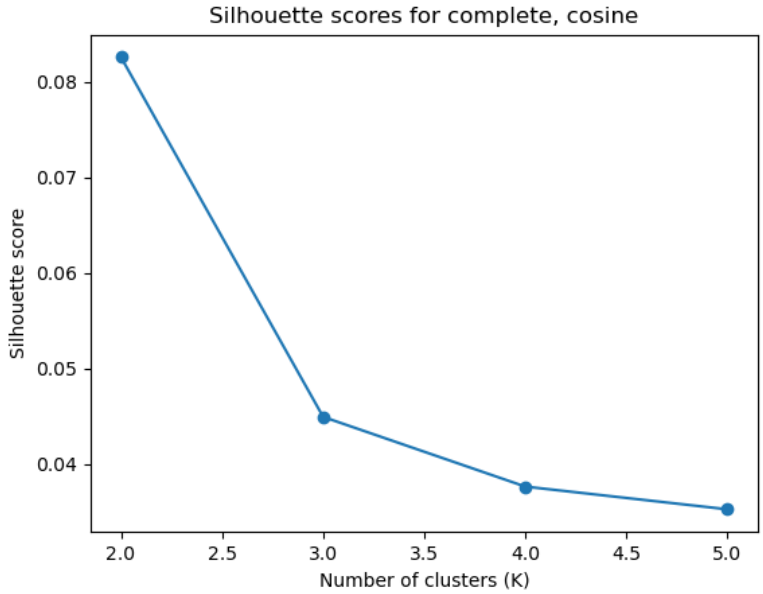

Figure 26: Silhouette plot for Complete, Cosine configuration

As we can see, the silhouette plots are just decreasing, and none of the configurations have particularly high silhouette scores. This most probably indicates that the clustering is not performed well, either due to the algorithm's limitations or because of the nature of the dataset. The dataset probably cannot be clustered well because of the distribution of its datapoints, or due to its high number of features (3072 features!).

Now for the DBSCAN algorithm, I wrote its code in "part3.py" as well. First I definied the hyperparameters:
eps = [0.05, 0.1, 0.2]
Distance Metrics = [Euclidean, Cosine]
Mininum Neighbouring Instances = [1, 2, 4]

Initially, I had also included 0.5 and 1 for eps, "Manhattan" for distance metric, and 3, 5 for Minimum Neighbouring Instances, but they either only gave 1 cluster or made each datapoint its own cluster, so I removed them from the hyperparameters. Then, I performed grid search over the final hyperparameters configurations (total of 18 configurations), and selected the top 4 with highest silhouette values. The results are:

```
###############
Best four configurations:

4) eps = 0.05, dist metric = cosine, min_samples = 4
Clusters formed: 2, Silhouette score: 0.0834

3) eps = 0.2, dist metric = cosine, min_samples = 1
Clusters formed: 22, Silhouette score: 0.1057

2) eps = 0.1, dist metric = cosine, min_samples = 2
Clusters formed: 3, Silhouette score: 0.1110

1) eps = 0.2, dist metric = cosine, min_samples = 2
Clusters formed: 3, Silhouette score: 0.1565
```

Figure 27: Results of DBSCAN

We can see that "Cosine" is better than "Euclidean" as a distance metric in all cases. For the best silhouette score, the configuration of [eps = 0.2, metric = Cosine, min_samples = 2] gives the highest average silhouette score of 0.1565 and creates three clusters.

Since we cannot tell DBSCAN to create a K amount of clusters, it decides the number of clusters itself, I could not create Silhouette vs K line graphs as in the previous algorithms. I instead created scatter plots, where each point represents each data instance and the height represents their silhouette values. I marked the average silhouette score with a red horizontal line. Each color represents a different cluster.

The scatter plots for the best four configurations are:

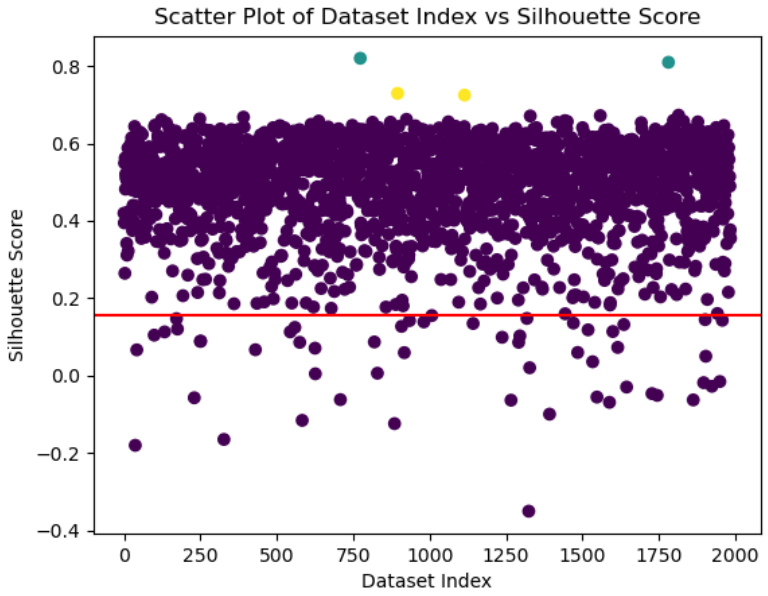

Figure 28: Scatter plot for the 1st best configuration

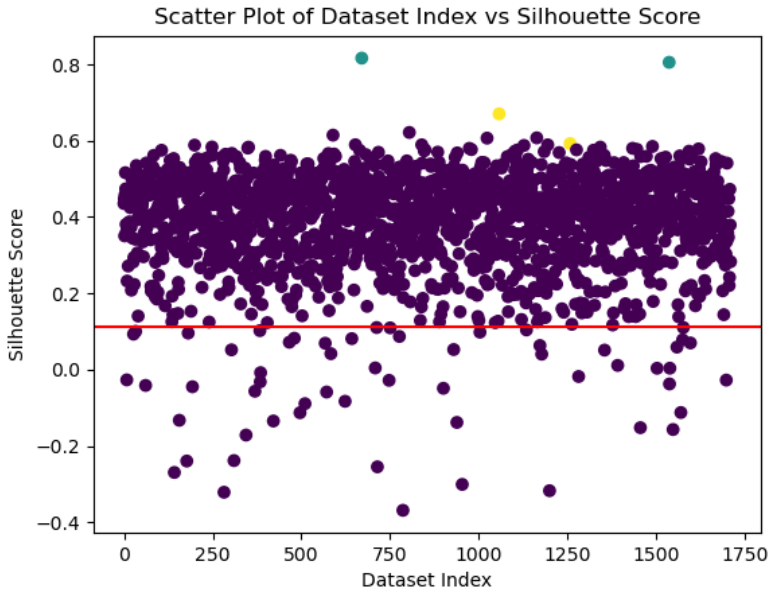

Figure 29: Scatter plot for the 2nd best configuration

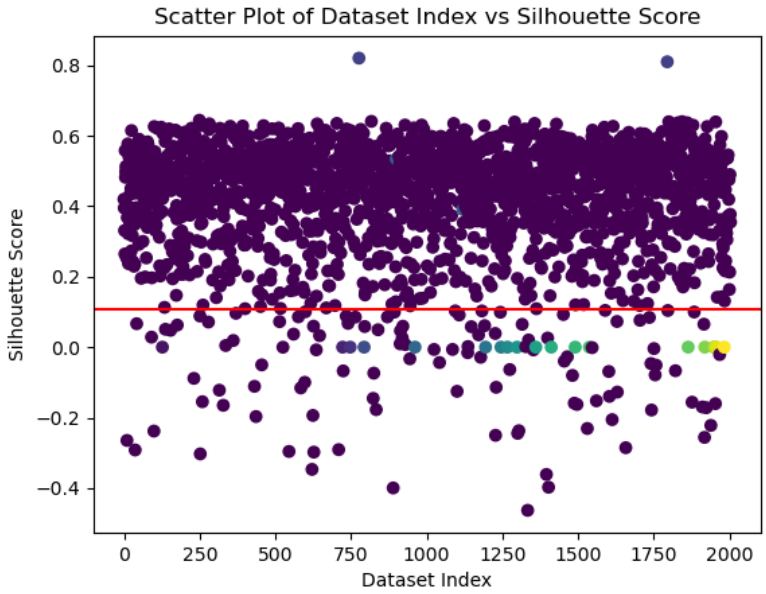

Figure 30: Scatter plot for the 3rd best configuration

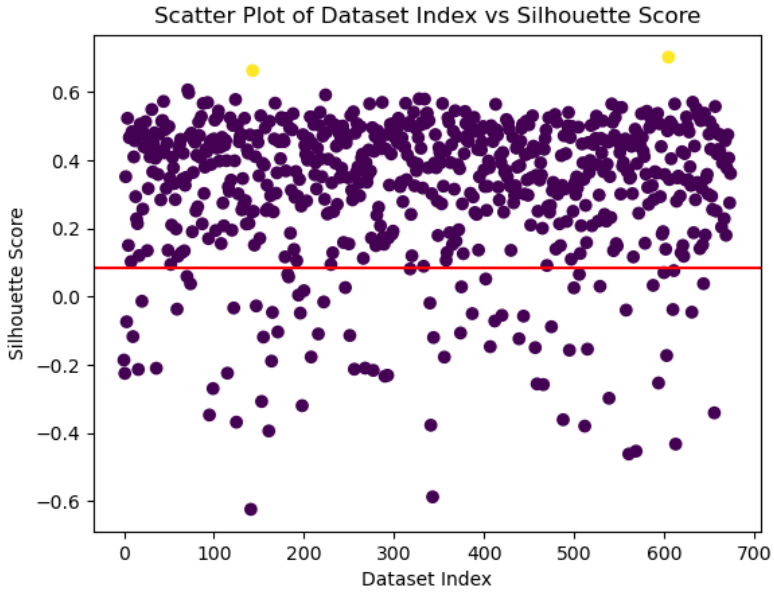

Figure 31: Scatter plot for the 4th best configuration

From the scatter plots, we can see that DBSCAN failed to create any meaningful clusters as well. It has clustered almost all the data points together, and the additional clusters are just for very few outliers. This further encourages the fact that the data is unable to be clustered due to its nature as stated for HAC.

For the dimensionality reduction section of part3, I filled in the "part3_dimensionalityreduction.py" file. The code is very short, I call the library functions for PCA, TSNE, and UMAP on the given dataset and plot the resulting 2-dimensional dataset. Most of the hyperparameters for the methods are set as default. The plots are:

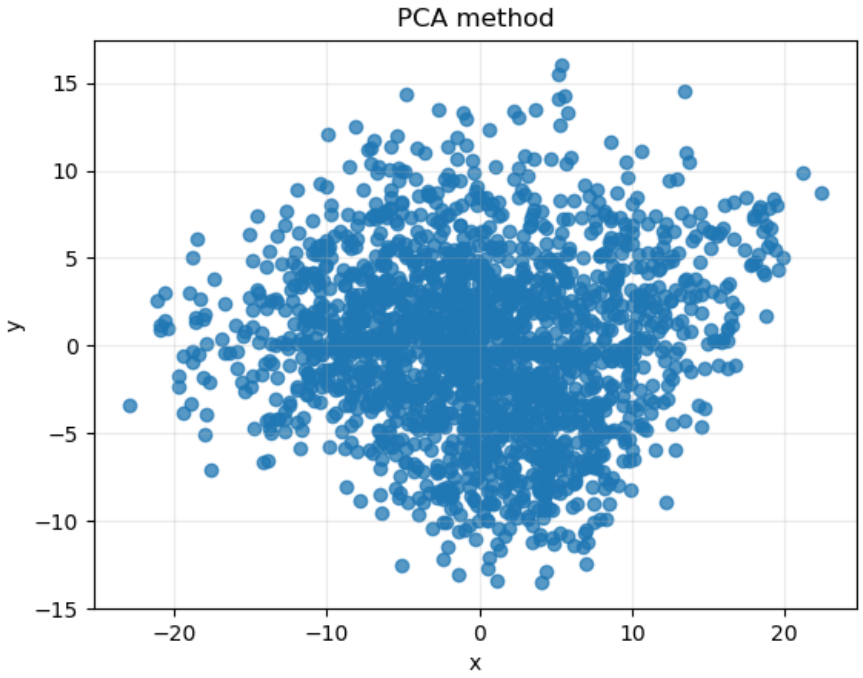

Figure 32: Dimensionality reduction with PCA

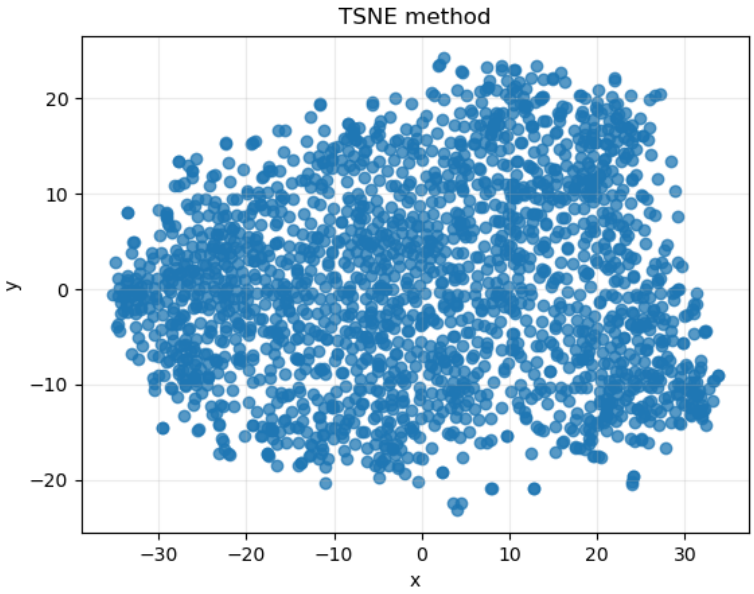

Figure 33: Dimensionality reduction with TSNE

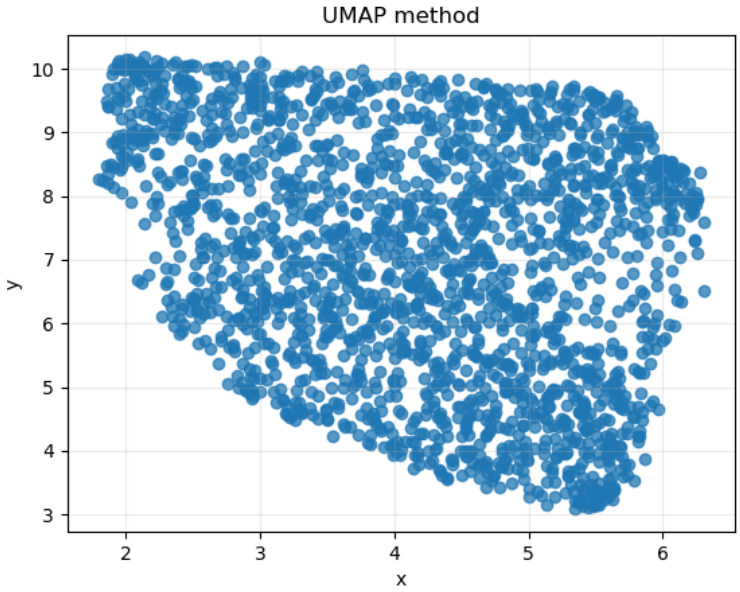

Figure 34: Dimensionality reduction with UMAP

As we can clearly see, none of the three dimensionality reduction methods were capable of showing any meaningful clusters for the dataset. This further cements our theory that the provided dataset cannot be clustered, either due to its data seperation or due to the high number of features.

Lastly, we do the runtime analysis of HAC. Firstly, we compute the pairwise distance (N^2) matrix of each datapoint, for each dimension (D). This takes $O(N^2 * D)$ time. Then, for each merge (N merges), we search for the lowest distance in the matrix (N^2). So, to merge all the clusters and get 1 final cluster, it takes $O(N^3)$ time. Adding the time taken for these two steps, we get the worst-case run time for HAC: $\boldsymbol{O(N^2 * D + N^3)}$.

For a huge dataset with 1 million data points and 120,000 dimensions, I would prefer to use K-Means as it scales linearly with both the datapoints N and the number of dimensions d. In contrast, HAC scales exponentially with N, and with the number of datapoints greater than the dimensions, HAC's runtime would be $O(N^3)$, which would take much more time than K-Means.