Intro to Machine Learning - Assignment 3+4

Maxmillan Ries s3118134, Cristian Rosiu s3742377

October 2020

Contents

1 Outliers detection and Density-based clustering

1.1 DBSCAN Implementation

a In the implementation of the algorithm we used the pseudo-code provided in the slides.

Because of the abstract nature of the pseudo-code, we had to make some design choices regarding the implementation of the *region_query*() function. For better performance, we pre-compute the distances of each point to all other points by using the distance matrix and sort indices based on the distances.

```python
self.distances = distance_matrix(self.points, self.points)
```

```python
# Compute neighbourhood of a core point
def region_query(self, i):
    neighbours = set()
    # Check core point's distances to all points
    for j in range(len(self.points)):
        if self.distances[i][j] <= self.eps:
            # If found point distances is in the radius
            # of the core point, add it to neighbourhood
            neighbours.add(j)
    return neighbours
```

It is also worth mentioning that, in order to check if a point is visited, the distance of a point or if a point it's already a member of another cluster, we decided to create lists for every property of a point. (e.g. The list *status_list* is initialized with status 'New' for each element in the points array. This means that each point in the original list is directly mapped using the index to the respective property list).

```python
# Number of samples / points
self.n_pts = len(points)
# List of labels
self.labels = [0] * self.n_pts
# Property Lists
self.status_list = [Status.New] * self.n_pts
self.member_list = [False] * self.n_pts
self.distances = distance_matrix(self.points, self.points)
```

Full code can be found in the appendix.

1.2 Evaluating DBSCAN on toy-sets

a For this part we have chosen to first create 2 arrays of values. One for the values that ϵ will take and the other array for the values of MinPts variable:

```python
eps_values = [0.15, 0.2, 0.25, 0.3, 0.32, 0.35, 0.4, 0.45,
↪  0.5]
min_pts_values = [3, 4, 5, 6, 7, 8, 9, 10]
```

We then proceeded to compute and plot the labels of each combination (eps, min_pts) in order to better visualize the resulted clusters.

What we observed first is that, changing the value of MinPts had less impact for bigger values on the overall data compared to when the value of ϵ changed. However, it is worth mentioning that for bigger values of MinPts, we got better results. Therefore, our observation is that, it doesn't make sense to choose a threshold value which is less than the dimension of your data (e.g. with a value of 1, there will be no outliers detected because each point will be a core point.).

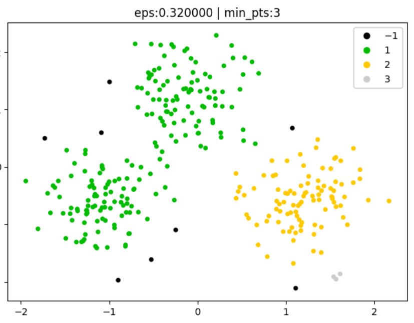


Figure 1: Clustering using min_pts = 3

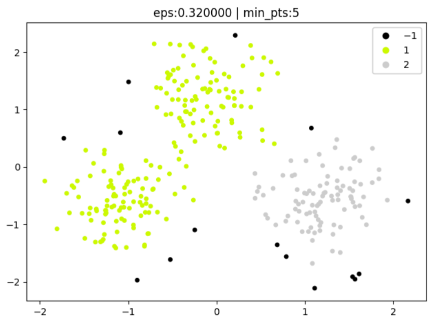


Figure 2: Clustering using min_pts = 5

On the other hand, as we already stated, the value of ϵ seems to have the most impact on data. We have observed that for a lower value, more clusters appear whereas if you keep increasing the value, the data tends to converge towards one larger cluster.

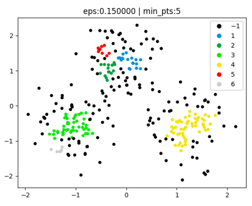


Figure 3: Epsion = 0.15 MinPts = 5

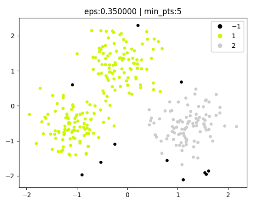


Figure 4: Epsion = 0.25 MinPts = 5

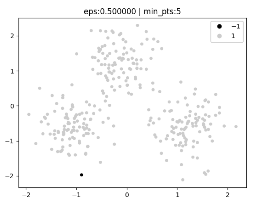


Figure 5: Epsion = 0.5 MinPts = 5

Which set of parameters allow you to cluster the data into 1, 2, and 3 clusters?

We have found the following set of parameters, which we have written underneath the matching graphs:

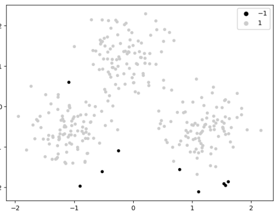


Figure 6: Epsion = 0.351 MinPts = 4

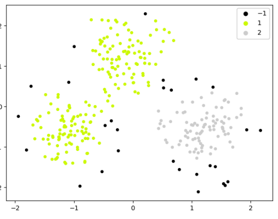


Figure 7: Epsion = 0.256 MinPts = 4

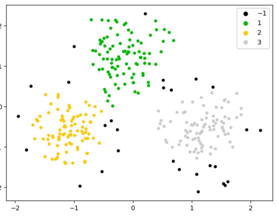


Figure 8: Epsion = 0.255 MinPts = 4

b **Compute and evaluate cluster's Silhouette score**

	1 Cluster	2 Clusters	3 Clusters
Silhouette Score	0.1960	0.3585	0.5316

What we can observe from the silhouette score of each cluster number is that, the most overlapping occurs when the algorithm finds a single clusters.

As an obvious observation based upon both logic and graph, is that the best silhouette score occurs when 3 clusters are found by the algorithm.

1.3 Outlier Detection On Subset MNIST Dataset

a Data Exploration

We started by retrieving the data from the given .mat file. Note that X array represents the actual values and y their lables

```python
# Retrieve Data
X = mnist_data['X']
y = mnist_data['y']
```

Then we proceeded to standardize the data, apply PCA on the result and then plot the newly reduce data:

```python
X = StandardScaler().fit_transform(X)
# Apply PCA on given data
pca = PCA(n_components=2)
X = pca.fit_transform(X)
# Plot the data and color code using the given labels (ground
↪   truth)
plt.scatter(X[:, 0], X[:, 1], c=y)
plt.show()
```

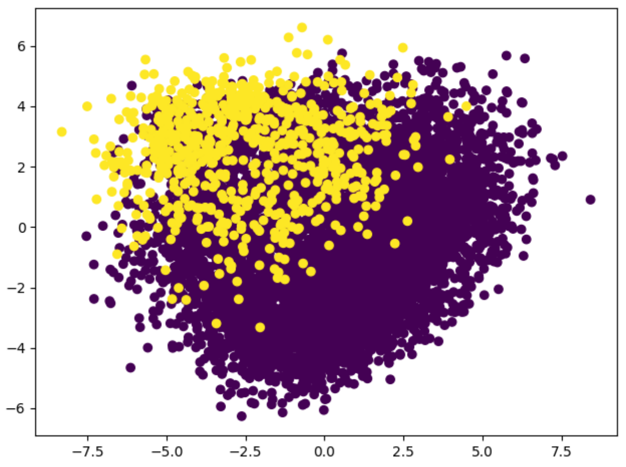


Figure 9: Plot Of MNIST Data Set After Dimensionality Reduction

As we can observe, using the ground-truth labels, the graph clearly shows the presence of 2 types of data. The yellow data points, as also stated in the data set folder, are the outliers and are label with the number 0. The magenta data points are labeled with number 1 and they represent inliers.

b Clustering - Outlier Detection

Firstly, we decided to plot the sorted distance of every point to it's k^{th} nearest neighbour (k = 4):

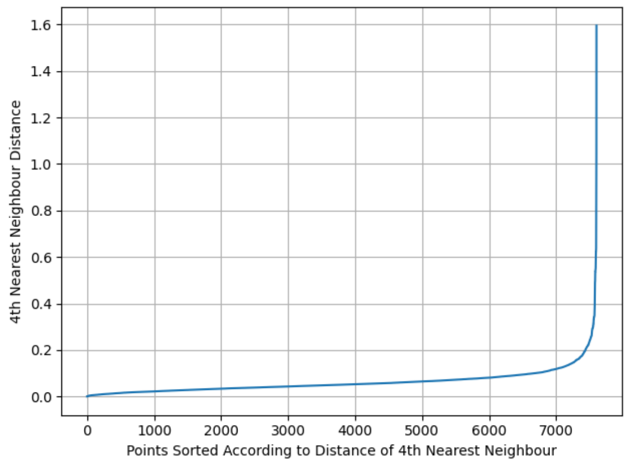


Figure 10: Graph Which Determines Optimal Epsilon

Here we can observe that the point of maximum curvature is around the interval:

$$0.2, 0.4$$

This implies that the optimal radius of our neighbourhood is in that interval. Everything above this value will be just too big to fit the graph.

So with a value of 0.35 for ϵ and MinPts = 20 we have got the following graph (-1 represents outliers)

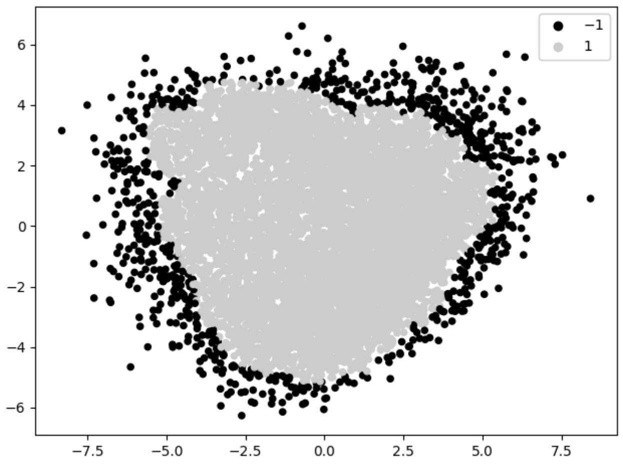


Figure 11: Epsilon = 0.35 MinPts = 5

c Evaluation

For the computation of accuracy and F1 scores we decided to use the confusion matrix in the following way:

```python
from sklearn.metrics import confusion_matrix

def evaluate(self, y_true):
    # Get the values of the confusion matrix
    tn, fp, fn, tp = confusion_matrix(y_true,
    ↪  self.labels).ravel()
    # Compute Accuracy score.
    accuracy = float((tp + tn)/(tp + tn + fp + fn))
    # Precision and recall will help calculate the F1 score.
    precision = float(tp/(tp + fp))
    recall = float(tp/(tp + fn))
    F1 = 2*precision*recall/(precision+recall)
    # Print the results.
    print('Accuracy: %f' % accuracy)
    print('F1 Score: %f' % F1)
```

Here is a table with the final scores:

Accuracy Score	0.157832
F1 Score	0.149781

As DBSCAN clusters according to the density of the data, it is not a suitable choice for the given data, which consists of handwritten labelled numbers. As such, we can see in the table above, that the accuracy and fscore are lower than expected.
A more suitable algorithm for this kind of clustering, we think, is one that focuses on detecting clusters, rather than detecting outliers based on the density of the data.

Therefore, DBSCAN is not suitable for the given data set (handwritten numbers from 0 to 9).

2 Vector Quantization

2.1 Winner-Takes-All unsupervised competitive learning (VQ)

For this assignment we decided to put every implemented algorithm in a single package and import them separately when testing their functionalities. Therefore, the VQ and DBSCAN algorithms can be found in the AssignmentAlgorithms.py, and can be imported as follows:

```python
from AssignmentAlgorithms import VQ, DBSCAN
```

Further more, the way we implemented Winter-Takes-All algorithms is quite simple and pretty intuitive. It's important to mention that our implementation is heavily inspired by the slides and instruction that we have got.

First of all, the most important function of our class is the following one:

```python
# X is the desired dataset.
def fit(self, X):
    # Randomly initialize K prototypes with actual points from the
    ↪  dataset.
    self.init_prototypes(X)
    # Plot first epoch where the prototypes are randomly
    ↪  positioned.
    self.plot_epoch(X, 0)
    for epoch in range(self.epochs):
        # Change data order in every epoch.
        randomized_data = permutation(X)
        sum_squared = 0.0
        for point in randomized_data:
            # Get the closest prototype to current point.
            winner_index, distance = self.evaluate_winner(point)
            # Calculate Squared error.
            for i in range(len(point)):
                error = point[i] - self.prototypes[winner_index][i]
                sum_squared += error**2
            # Move prototype towards point using the learning_rate.
            self.update_prototype(winner_index, point)
        self.squared_errors.append(sum_squared)
    # Plot final result.
    self.plot_epoch(X, 0)
```

It updates each prototype, plots the result of each epoch and computes the squared error distance. Moreover, another somewhat important function which we thought it would be important to talk about is the one that finds the winner prototype:

```python
# Determines the winner prototype for a specific point
def evaluate_winner(self, data_point):
    winner = (0, sys.maxsize)
    # Go over the list of prototypes and find the closest using
    ↪  euclidean distance
    for index, prototype in enumerate(self.prototypes):
        distance = np.linalg.norm(prototype - data_point)
        # Find prototype with the minumim distance.
        if distance < winner[1]:
            winner = (index, distance)
    # Return the winner index
    return winner[0], winner[1]
```

The update winner prototype function follows the pseudocode described closely, iterating over each prototype and updating it if its the winner prototype, rpeating the process for each data point.
An important detail is that we chose to calculate the Hvq directly as part of our running algorithm, rather than create a separate function.
After implementing this, we then tested the functionality on the given data set.

2.2 Simple VQ Data Set

a As stated in the assignment, we proceeded to test our algorithm on the given data set by using different values for K and 0.1 for the learning rate. We would like to precise that the given data set did not contain any names for each component of the vectors, and as such as have kept them blank in our graphs. First, we found the following results using K=2:

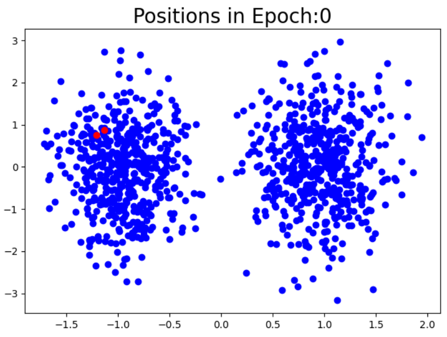


Figure 12: Initial guess using K = 2

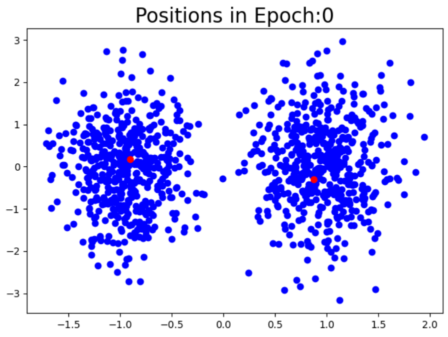


Figure 13: Final guess after all iterations using K = 2

The left picture shows the initial location of the 2 prototypes after randomly assigned positions whereas the right one shows the same prototypes, but in their final position. We can clearly observe that the algorithm managed to find the 2 obvious clusters. However, the two prototypes are not exactly in the center of each cluster, but slightly off. There may well be a combination of causes which result in this, but our current hypothesis is that, due to the randomly selected data, the final few evaluated points consisted of the outer values of the respective clusters, causing the prototypes to be "pulled" a little away from the center.
We are also fairly confident in stating that the learning rate has a strong impact on the visual and mathematical shifting of the winner prototype and may partially be responsible for the lack of a perfect center.
On the other hand, we found the following results for K=4 and the same learning rate

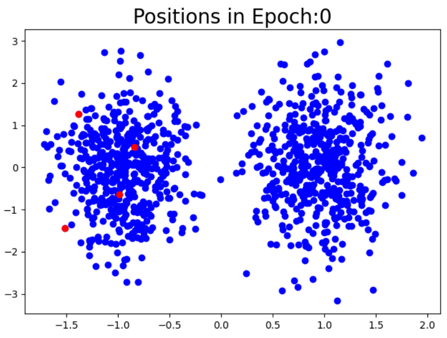


Figure 14: Initial guess using K = 4

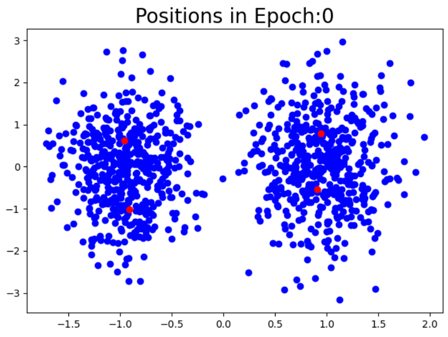


Figure 15: Final guess after all iterations using K = 4

Here, we can observe a similar behaviour to the example where we had K = 2. However, the distance between one prototype and the center of a cluster is much bigger now. This is caused by what is described as the "ill-defined" issue with clustering. From a mathematical and logical standpoint, the algorithm has function, of splitting the data both horizontally and vertically once.
Visually for us as observers however, the clustering using K=4 does not seem to have resulted in something tangible and properly done. To evaluate this further, the elbow method will be used later in the assignment, as to find the best number of clusters for this set of data. Our current hypothesis is that the elbow method will result in an optimum number of clusters being equal to 2.
For comparison, we have also tried a smaller value for the learning rate (learning_rate=0.05). We have got the following results:

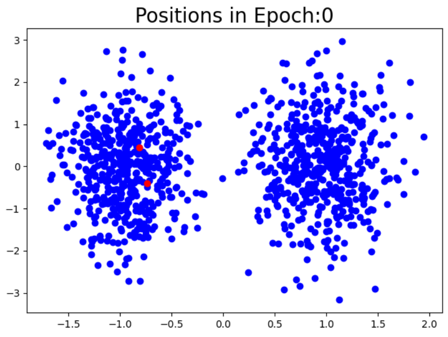


Figure 16: Initial guess using learning_rate = 0.05

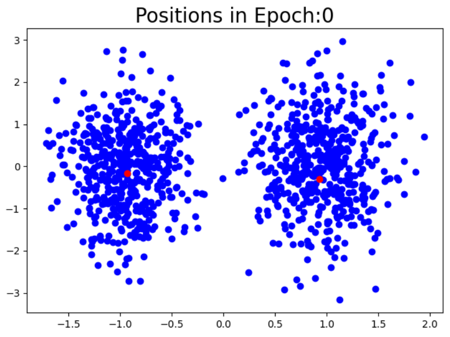


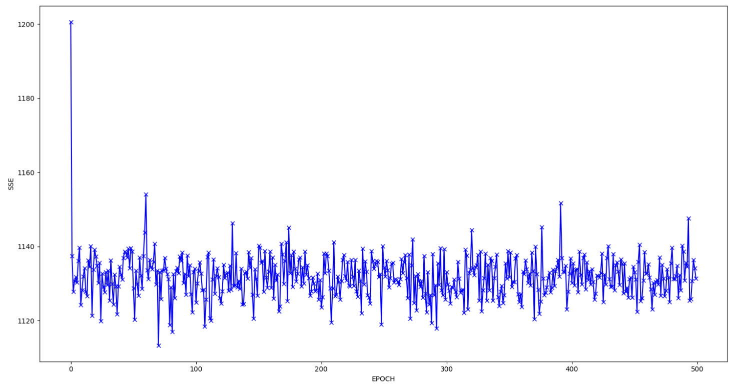
Figure 17: Final guess after all iterations using learning_rate = 0.05

It seems that, running the algorithm with a smaller learning rate value, significantly increased the chance of a better guess. This was expected as the distance traveled by a winner prototype is directly proportional with the value of the learning_rate. Formula proof: (n is learning rate, e is current data point):

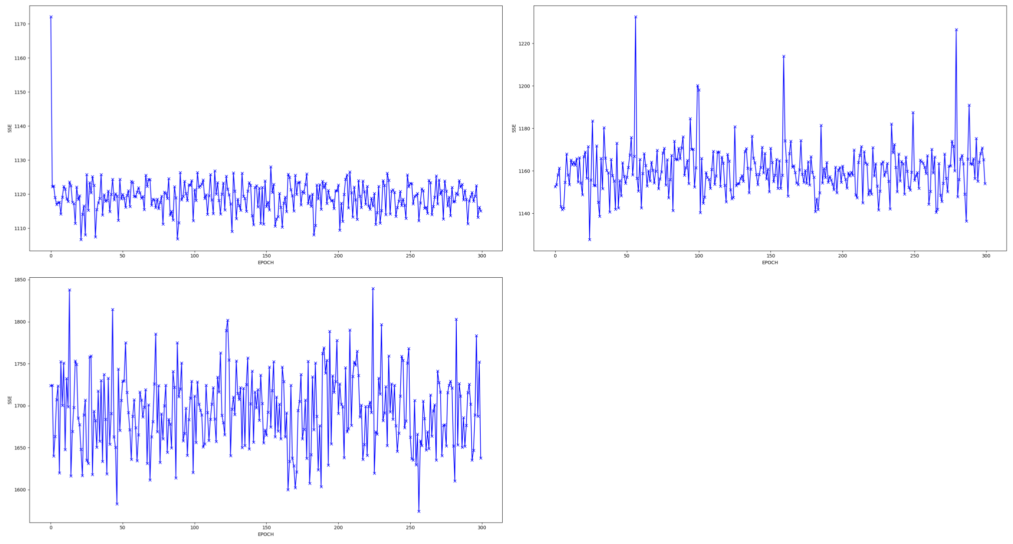
```
w^{i*} = w^{i*} + n(e^{t}w^{i*})
```

N, or the learning rate, is a ratio that scales the difference in the distance (between the winner and the data point), before adding it to the current position of the winner prototype, effectively determining how far the prototype will travel. A lower value of n, will result in a shorter distance travelled, which can prevent any cluster outliars/values from causing the prototypes to shift drastically away from the center of the cluster.

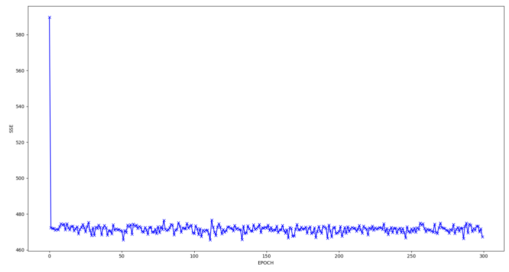
For a K value of 2, we have found that a reasonable value of epochs is 100. Beyond this number of epochs, the value of Hvq does not seem to steadily decrease towards a minimum any further.

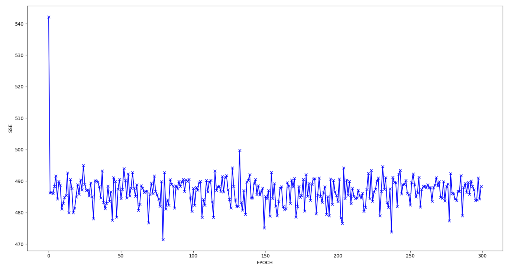


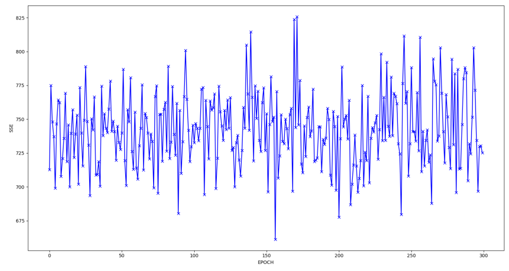
b The following graphs present the variation of Hvq in each epoch, for K=2 and K=4 for the following values of learning rate: 0.04, 0.1, 0.6:



In the above iterations (K = 2), we can clearly observe the larger variation in error with a slow learning rate compared to the increasing variation of the other graphs. This seems to go against our hypothesis. This will be further discussed below.

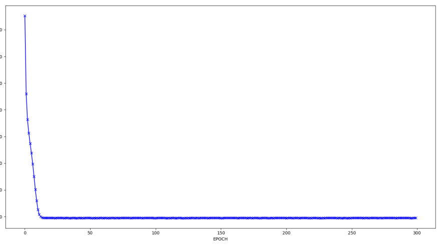






On the other hand, it seems that if the number of prototypes K is increased, the variation in error is much smaller compared to a smaller K. With a value of K, the data is ”split” according the winner takes all rules K times. As the value of K goes up, the prototypes will be scattered such that the difference between the data surrounding each prototype and each respective prototype will be smaller. As such, due to the smaller euclidean distances obtained as a result, the value of Hvq, which is the sum of these distances, will be smaller.
It is for this reason, that, assuming perfect clustering, if K = number of data points, the value of Hvq will go to 0.

How does the final value of the cost function change with n?
The learning rate is a ratio which affects the extent to which the prototype is updated. Our hypothesis, is that a too high or too low learning will lead to insufficient clustering.
To further prove our point we decided to apply the algorithm with extreme values for the learning rate (i.e. 0.0001 and 1) and plot it to see what would the result be.



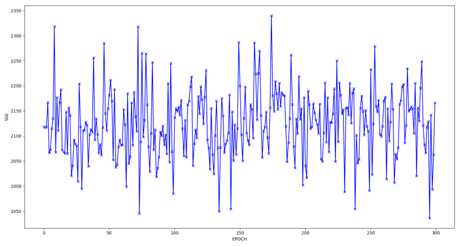


Figure 18: Learning_rate = 0.001

Figure 19: Learning_rate = 1

From the graph with a learning rate of 0.01, we can see exactly what was expected. With a too low learning rate, the equation will simply equal the following:

```
w^{i*} = w^{i*}
```

This means that, whatever data points are chosen, the prototypes are simply not updated, or sufficiently updated, and as such, the clustering will equal the random selection was was initially done before the first epoch. This translates to a flat line for the Hvq value as nothing changes.

If the value of the learning rate is too high (in this case 1), the Hvq will continuously spike as the prototypes will continuously be updated to the position of the data point they are compared to. As such, the clustering will not really occur, in the sense that the final prototype values will simply equal the coordinates of the data points processed last as a result of the randomization.

Beyond the extremes, we found that a good learning rate value, n, seemed to be located in the range between 0.05 and 0.1. With this value, or range of values, we found that the clustering seemed optimal, in the sense that the final prototype placements were in the center of the respective clusters and that the travel path of each prototype, which is graphed further down in this report, shows a smooth and continuous shift.

Repeat the training from at least two different initializations (one of them "clever", one of them "stupid"). Discuss the observed effects.

Taking a learning rate of 0.05 and a K value of 4, we took what we considered to be a "clever" and a "stupid" start and plotted the trajectories of each prototype.
Clever: For the clever start, we positioned the prototypes initially close to their final value and observed that small incremental shifts caused the

points to get closer and closer to the center. We found that the Hvq values were significantly lower with the clever start on initialization, due to the euclidean distances between each data point and matching prototype being smaller.
We can see in the graph below that the trajectories spend most of the time shifting around the "ideal" center, something which is caused by the fact that we are using a static learning rate. This will be further addressed below.

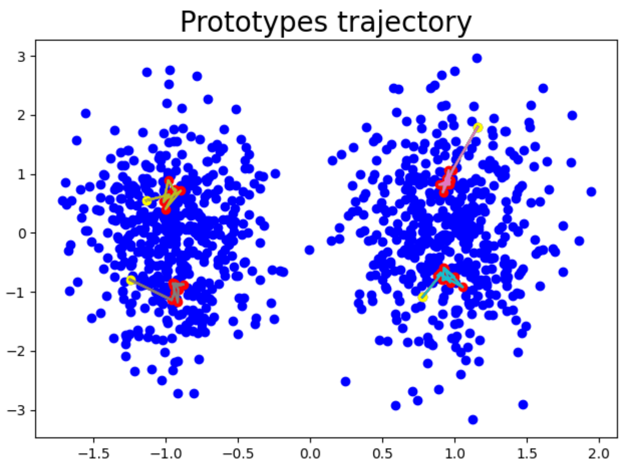


Figure 20: Plot Of Prototype Trajectories with Clever Start. Yellow = Initial Position

Stupid: The stupid start was to position all 4 prototypes in the right cluster.
When observing their trajectories, we can immediately see that the first update for the 2 prototypes closest to the left consists of both being immediately shifted into the cluster they will remain in.
The H_{VQ} values we observed were higher in this case as the difference first iteration yielded a big euclidean difference for 2 of the prototypes.

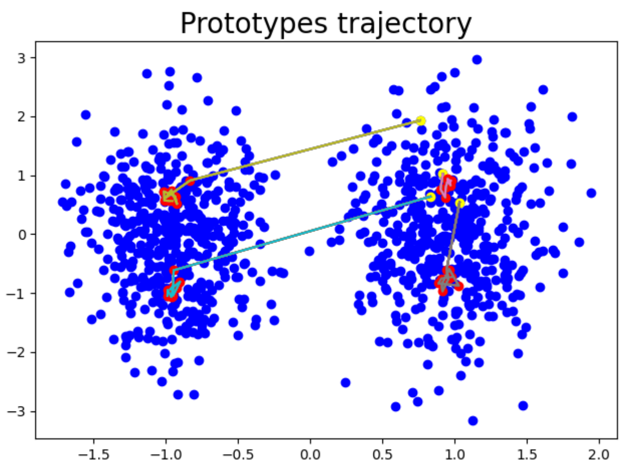


Figure 21: Plot Of Prototype Trajectories with Stupid Start. Yellow = Initial Position

As a conclusion to this question, while a clever start is a very useful starting point, we experimented further with the learning curve and found a better solution which yielded lower Hvq values over time.
This solution consisted of adapting the learning rate at every iteration, decreasing it at each epoch, as at each time unit, the prototypes get closer to their ideal centering and hence do not require as high of a translation. This solution however, was not expected of us, and shall not be discussed any further in this report.

2.3 Bonus - Elbow Method

For the bonus question we have chosen to try and use the elbow method in order to find the most suitable K for our data set. Below we plotted H_{VQ} as a function of K.

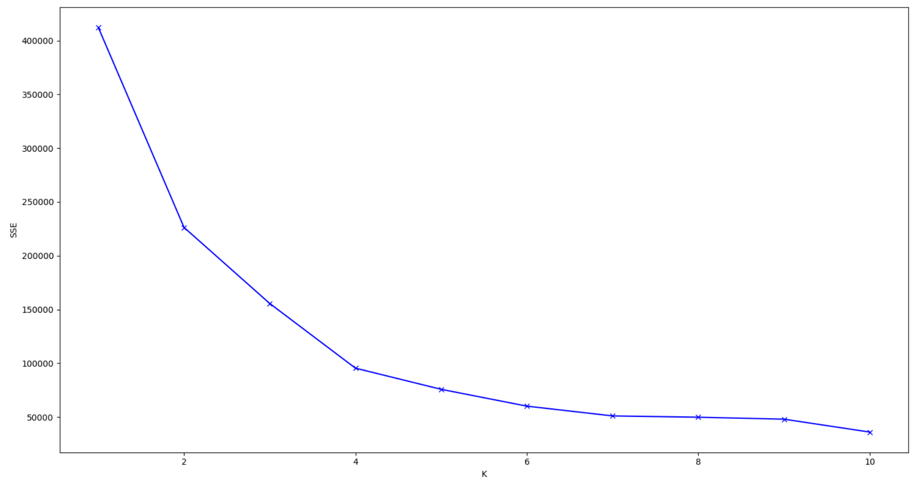


Figure 22: Elbow Method Plot

From the graph, we can clearly observe that the ”elbow” is at K = 4. It became apparent the moment we plotted the H_{VQ} graph, that our initial hypothesis of K = 2 being the optimal number of clusters was incorrect and that K = 4 was the better number of prototypes.
Despite K = 4 being the optimal number however, we can observe that K = 2 is not far off in terms of efficiency and when comparing the H_{VQ} values between each, while there is a difference, it is noticeably smaller than the difference between K = 4 and other value than 2.

3 Appendix

3.1 Code

3.1.1 DBSCAN

```
# Create simple class used as Enum
class Status:
    def __init__(self):
        pass

    New, Visited, Noise = range(3)


def distance(a, b):
    return np.linalg.norm(a - b)


```

```python
class DBscan():
    def __init__(self, points, eps=0.15, min_pts=2):
        self.points = points
        # Maximum radius of a point in order to be regarded as
        ↪  neighbour
        self.eps = eps
        # Minimum number of neighbours that a point needs to have
        ↪  in order for it's density
        # to be regarded as "high density"
        self.min_pts = min_pts
        # Number of samples / points
        self.n_pts = len(points)
        # List of labels
        self.labels = [0] * self.n_pts
        # Property Lists
        self.status_list = [Status.New] * self.n_pts
        self.member_list = [False] * self.n_pts
        self.distances = distance_matrix(self.points, self.points)

    def fit(self):
        # Initial cluster label
        cluster = 0
        for i in range(self.n_pts):
            if self.status_list[i] == Status.Visited:
                continue
            self.status_list[i] = Status.Visited
            neighbours = self.region_query(i)
            if len(neighbours) < self.min_pts:
                self.status_list[i] = Status.Noise
            else:
                cluster = cluster + 1
                self.add_to_cluster(cluster, i)
                self.expand_cluster(neighbours, cluster)

        # Set noise points in the label list as the value -1
        for i, label in enumerate(self.labels):
            if self.status_list[i] == Status.Noise:
                self.labels[i] = -1

    # Compute neighbourhood of a core point
    def region_query(self, i):
        neighbours = set()
        # Check core point's distances to all points
        for j in range(len(self.points)):
            if self.distances[i][j] <= self.eps:
                # If found point distances is in the radius
                # of the core point, add it to neighbourhood
                neighbours.add(j)
        return neighbours

    def expand_cluster(self, neighbours, cluster):
        while neighbours:
            index = neighbours.pop()
            if self.status_list[index] != Status.Visited:
                # If Noise or Undefined, transform the point into a
                ↪  border
                self.status_list[index] = Status.Visited
                extended_neighbours = self.region_query(index)
                if len(extended_neighbours) >= self.min_pts:
                    neighbours.update(extended_neighbours)
            if not self.member_list[index]:
                self.add_to_cluster(cluster, index)

    def add_to_cluster(self, cluster, i):
        self.labels[i] = cluster
        self.member_list[i] = True

    def evaluate(self, y_true):
        # Get the values of the confusion matrix
        tn, fp, fn, tp = confusion_matrix(y_true,
        ↪  self.labels).ravel()
        # Compute Accuracy score.
        accuracy = float((tp + tn)/(tp + tn + fp + fn))
        # Precision and recall will help calculate the F1 score.
        precision = float(tp/(tp + fp))
        recall = float(tp/(tp + fn))
        # Print the results.
        print('Accuracy: %f' % accuracy)
        print('F1 Score: %f' % float((2 * precision *
        ↪  recall)/(precision+recall)))
```

3.1.2 Toy Dataset Code

```
from AssignmentAlgorithms import DBscan
import numpy as np
from sklearn.metrics import silhouette_score
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler


def plot_data(data, labels):
    fig, ax = plt.subplots()

    scatter = ax.scatter(data[:, 0], data[:, 1], c=labels,
    ↪  cmap='nipy_spectral', s=15)
    legend = ax.legend(*scatter.legend_elements())
```

```python
    ax.add_artist(legend)
    plt.show()


if __name__ == "__main__":
    X = np.load('toy_set.npy')
    X = StandardScaler().fit_transform(X)

    eps_values = [0.15, 0.2, 0.25, 0.3, 0.32, 0.35, 0.4, 0.45, 0.5]
    min_pts_values = [3, 4, 5, 6, 7, 8, 9, 10]
    for epsilon in eps_values:
        for min_samples in min_pts_values:
            dbscan = DBscan(X, epsilon, min_samples)
            dbscan.fit()
            plot_data(X, dbscan.labels)
            print(silhouette_score(X, dbscan.labels))
```

3.1.3 Minst Dataset Code

```python
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
import scipy.io as sio
import matplotlib.pyplot as plt
from AssignmentAlgorithms import DBscan
from sklearn.neighbors import NearestNeighbors
import numpy as np


def plot_data(data, labels):
    fig, ax = plt.subplots()

    scatter = ax.scatter(data[:, 0], data[:, 1], c=labels,
    ↪  cmap='nipy_spectral', s=20)
    legend = ax.legend(*scatter.legend_elements())
    ax.add_artist(legend)
    plt.show()


mnist_data = sio.loadmat('mnist.mat')

# --------- Data Exploration ---------
X = mnist_data['X']
y = mnist_data['y']
# Standardize the data
X = StandardScaler().fit_transform(X)
# Apply PCA on given data
pca = PCA(n_components=2)
X = pca.fit_transform(X)
```

```python
# Plot the data and color code using the given labels (ground
↪  truth)
plt.scatter(X[:, 0], X[:, 1], c=y, s=5)
plt.show()

# --------- Clustering - Outlier Detection ---------

# Find optimal value for Epsilon and plot the graph
neigh = NearestNeighbors(n_neighbors=4)
nbrs = neigh.fit(X)
distances, indices = nbrs.kneighbors(X)
distances = np.sort(distances, axis=0)
distances = distances[:, 1]
plt.xlabel('Points Sorted According to Distance of 4th Nearest
↪  Neighbour')
plt.ylabel('4th Nearest Neighbour Distance')
plt.grid()
plt.plot(distances)

# Apply algorithm using the optimal value
dbs = DBscan(X, eps=0.35, min_pts=20)
dbs.fit()
y_pred = dbs.labels
plot_data(X, y_pred)
# Transform -1 labels to 0 in order to mach the ground-truth data
for i in range(len(y_pred)):
    if y_pred[i] == -1:
        y_pred[i] = 0
# Evaluate Scores
dbs.evaluate(list(y))
```

3.1.4 VQ Code

```python
class VQ:
    def __init__(self, K, learning_rate, epochs):
        self.K = K
        self.learning_rate = learning_rate
        self.epochs = epochs
        self.prototypes = []
        self.squared_errors = []
        self.prototypes_trajectory = {}
        self.is_fit = False

    # Initialize the prototypes list with random points from the
    ↪  dataset.
    def init_prototypes(self, data):
        for i in range(self.K):
            self.prototypes.append(np.copy(data[random.randint(0, len(data))]))
            self.prototypes_trajectory[i] = []
        self.prototypes = np.array(self.prototypes)

    # Used to update a winner prototype.
    def update_prototype(self, prototype_index, point):
        self.prototypes[prototype_index] += self.learning_rate * (point - self.prototypes[prototype_index])

    # Plot all prototypes as red points.
    def plot_epoch(self, X, epoch_number, trajectory=False):
        if not self.is_fit:
            raise Exception('Use fit before you plot')
        if not trajectory:
            plt.title('Positions in Epoch:%d' % epoch_number, fontsize=20)
            plt.scatter(X[:, 0], X[:, 1], c='b')
            plt.scatter(self.prototypes[:, 0], self.prototypes[:, 1], c='r')
            plt.show()
        else:
            plt.title('Prototypes trajectory', fontsize=20)
            if epoch_number == 0:
                plt.scatter(X[:, 0], X[:, 1], c='b')
                plt.scatter(self.prototypes[:, 0], self.prototypes[:, 1], c='yellow')
            else:
                plt.scatter(self.prototypes[:, 0], self.prototypes[:, 1], c='r')

            for element in self.prototypes_trajectory.values():
                plt.plot(np.array(element)[:, 0], np.array(element)[:, 1])

    # Plots the error curve.
    def plot_error(self):
        if not self.is_fit:
            raise Exception('Use fit before you plot')
        epochs = range(self.epochs)
        plt.figure(figsize=(15, 8))
        plt.plot(epochs, self.squared_errors, 'bx-')
        plt.xlabel('EPOCH')
        plt.ylabel('SSE')
        plt.show()

    def evaluate_winner(self, data_point):
        winner = (0, sys.maxsize)
        for index, prototype in enumerate(self.prototypes):
            distance = np.linalg.norm(prototype - data_point)
            if distance < winner[1]:
                winner = (index, distance)
        return winner[0], winner[1]

    # X is the desired dataset.
    def fit(self, X, show_trajectory=False, show_plot=False):
        self.is_fit = True
        # Randomly initialize K prototypes with actual points from the dataset.
        self.init_prototypes(X)
        for epoch in range(self.epochs):
            # Change data order in every epoch.
            randomized_data = permutation(X)
            sum_squared = 0.0
            for i in range(len(self.prototypes)):
                # Keeps track of the history of prototype positions.
                self.prototypes_trajectory[i].append(list(self.prototypes[i]))
            if show_trajectory:
                self.plot_epoch(X, epoch, show_trajectory)
            if show_plot:
                self.plot_epoch(X, epoch)
            for point in randomized_data:
                # Get the closest prototype to current point.
                winner_index, distance = self.evaluate_winner(point)
                # Calculate Squared error.
                for i in range(len(point)):
                    error = point[i] - self.prototypes[winner_index][i]
                    sum_squared += error ** 2
                # Move prototype towards point using the learning_rate.
                self.update_prototype(winner_index, point)
            self.squared_errors.append(sum_squared)
        if show_trajectory:
            plt.show()
```

3.1.5 VQ Application Code

```
from AssignmentAlgorithms import VQ
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
import pandas

```

```python

def plot_elbow(X):
    SSE = []
    for k in range(1, 21):
        vq = VQ(k, 0.1, 100)
        vq.fit(X)
        SSE.append(sum(vq.squared_errors))
    K = range(1, 21)
    plt.figure(figsize=(15, 8))
    plt.plot(K, SSE, 'bx-')
    plt.xlabel('K')
    plt.ylabel('SSE')
    plt.show()


if __name__ == '__main__':
    X = pandas.read_csv('simplevqdata.csv').to_numpy()
    X = StandardScaler().fit_transform(X)

    # Plot the points and the error graph for various values of K
    ↪  and learning rate
    K = [2, 4]
    rates = [0.04, 0.1, 0.7]
    for k in K:
        for rate in rates:
            vq = VQ(k, rate, 100)
            vq.fit(X)
            vq.plot_error()

    # Plot the K vs SSE graph in order to find the most optimal K
    plot_elbow(X)
```