Introduction to Machine Learning
Assignment 6
Density-based clustering and outlier detection

Group 05
Bora Yilmaz (s3903125) & Felix Zailskas (s3918270)

October 27, 2021

Contents

1 Introduction **2**

2 Method **2**
2.1 DBSCAN Algorithm 2
2.2 Hyper-Parameter Selection 3
2.3 Code Tricks 4

3 Results **5**
3.1 Minimum Neighbors = 3 6
3.2 Minimum Neighbors = 4 7
3.3 Minimum Neighbors = 5 8
3.4 Silhouette Scores 8

4 Discussion **9**
4.1 Cluster Results 9
4.1.1 Minimum Points = 3 9
4.1.2 Minimum Points = 4 9
4.1.3 Minimum Points = 5 9
4.1.4 Noise Points (Outliers) 10
4.2 Silhouette Scores 10
4.3 Parameters and Best Choices 10

5 Contribution **11**
5.1 Code 11
5.2 Report 11

6 Code Appendix **11**
6.1 main.py 11
6.2 dbscan.py 12
6.3 knn.py 13
6.4 silhouette.py 14
6.5 plotting.py 16

Density-based clustering and outlier detection

1. Introduction

In this report, the DBSCAN clustering algorithm is implemented and discussed, along with a K-nearest neighbours search and silhouette scores of the resulting clusters. The acronym "DBSCAN" stands for "Density-based spatial clustering of applications with noise". It is density-based, meaning that it will refer to density of a point, in this case the density is the neighborhood of a point as defined in section 2. The algorithm can identify noise points, as well as identify clusters in the data. This results in clusters that ignore the noise points, which would otherwise decrease the clustering accuracy.
Compared to other clustering algorithms, DBSCAN has multiple upsides. It is known to be robust against noise in a data set. Additionally, it is not dependent on clusters having similar density. Also, it is not needed to define the amount of clusters before applying the algorithm. One of the drawbacks of DBSCAN is hyper-parameter sensitivity. This will be discussed further in section 4. One way to deal with this is to apply the K-nearest neighbour analysis to the data as presented in subsection 2.2.

2. Method

2.1. DBSCAN Algorithm

The DBSCAN algorithm is an algorithm to determine clusters inside of a data set. It uses information on the density of the data points to create the clusters. Due to the way this algorithm works, it can also detect outliers inside of the data set.
The algorithm essentially separates all data points into one of three classes. A data point can either be a noise point (outlier), a center point or a border point. Center and border points are both part of clusters, while noise points are not part of any clusters, and can therefore be identified as outliers in the data set.
There are two hyper-parameters used in the DBSCAN algorithm. The first is ϵ, the distance around each data point in which neighbors are searched. The second is *min_pts*, which determines how many neighbors a data point needs to have to be identified as a center point. How to determine these hyper-parameters will be discussed in more detail in subsection 2.2.
The density of a data point is defined as the amount of neighbours in its ϵ-neighborhood (including itself).

The ϵ-neighbourhood N_ϵ of a point x is defined as:

$$N_\epsilon(x) = \{y \mid distance(x, y) \leq \epsilon\} \tag{1}$$

In our implementation, we used Euclidean distance as the distance measure between two data points.

Given two data points x, y, their distance $distance(x, y)$ is defined as:

$$distance(x, y) = \sum_{i=1}^{n} (x_i - y_i)^2 \tag{2}$$

Note: For the calculation of silhouette scores, Euclidean distance is used for the sake of comparing it to the Hierarchical Clustering algorithm results which also used Euclidean distance.

In order to assign the correct label to each data point the algorithm examines each data point in the data set. For each point the ϵ-neighborhood is retrieved. The correct label for that data point is determined by the following properties of center, border and noise points.

1. **Center Points**:
   Center points are all points that have at least *min_pts* neighbors within their ϵ-neighborhood.

2. **Border Points**:
   Border points are all points that have less than *min_pts* neighbors within their ϵ-neighborhood, but are in the ϵ-neighborhood of a center point.

3. **Noise Points**:
   Noise points are those points that are neither center nor border points.

To achieve this assignment a data point is evaluated based on the number of neighbors. Then each of those neighbors is evaluated afterwards. If a point has less neighbors than *min_pts* it is initially marked as noise. Every point that has at least *min_pts* neighbors is marked as a center of the current cluster and its neighborhood is evaluated. If a neighbor of a data point with at least *min_pts* neighbors has less than *min_pts* neighbors it will be marked as a border point. If a neighbor of a data point with at least *min_pts* neighbors also has at least *min_pts* neighbors then the cluster of that neighbor and the initial data point are merged together. After evaluating all data points reached by neighboring data points the current cluster is set to a new cluster and the process is repeated for all unvisited data points.
Each data point must only be visited once, hence data points must be marked as visited when they have been evaluated. The aforementioned operations are only executed on unvisited data points.

2.2. Hyper-Parameter Selection

In our analysis we investigated three different values for *min_pts*, namely 3, 4 and 5. For each of those we have to chose an appropriate value for ϵ. To do this we can use a k-NN graph. This graph displays the average distance of the k nearest neighbors for each data point, sorted in ascending order. We can assume that data points with large values in the k-NN graph are likely to be outliers due to the large distance to all other data points. Hence, an appropriate value for ϵ is the distance value of the elbow of the k-NN graph. Choosing an ϵ value like this allows the DBSCAN algorithm to properly detect data points that should be clustered together and outliers at the same time.

We have obtained the following k-NN graphs for $k = 3, 4, 5$.

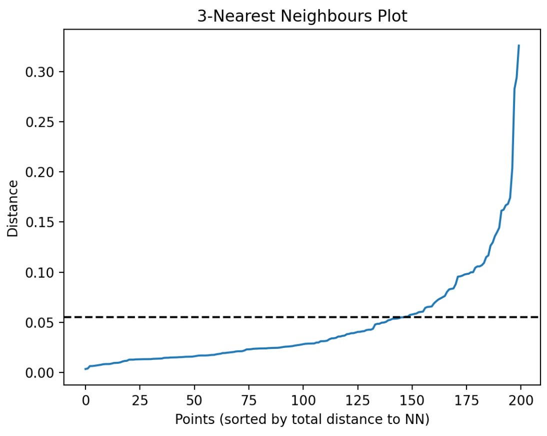

Figure 1: *k-NN Graph for $k = 3$, with a Cutoff Value of 0.055.*

In Figure 1 we can see that the elbow of the curve is around a distance of 0.055. This is the ϵ we will use for our analysis with *min_pts* $= 3$.

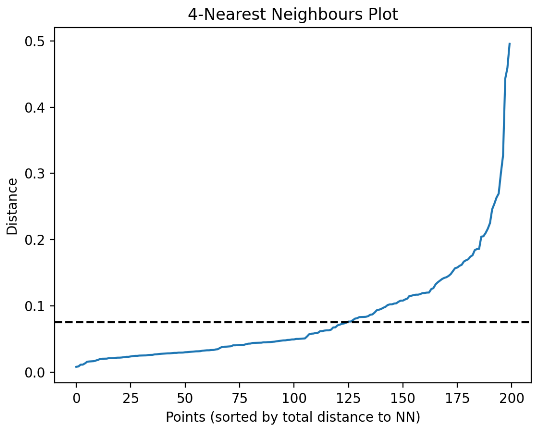

Figure 2: *k-NN Graph for $k = 4$, with a Cutoff Value of 0.075.*

In Figure 2 we can see that the elbow of the curve is around a distance of 0.075. This is the ϵ we will use for our analysis with $min\_pts = 4$.

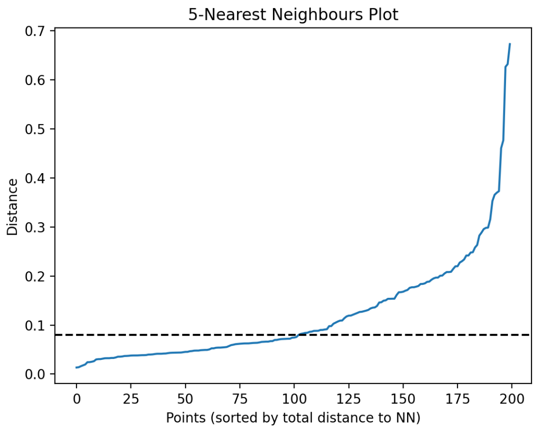

Figure 3: *k-NN Graph for $k = 5$, with a Cutoff Value of 0.08.*

In Figure 3 we can see that the elbow of the curve is around a distance of 0.08. This is the ϵ we will use for our analysis with $min\_pts = 5$.

2.3. Code Tricks

One problem that we had to solve in our implementation of the DBSCAN algorithm is that of keeping track of the clusters the data points are part of and whether a certain data point has been visited before.
To do this we added another two dimensions to each data point.
The first of these dimensions indicates the cluster the data point is part of. This is an integer, that represents the cluster index of that data point. In the case that the data point is a noise point it will have a cluster value (label) of -1. Each cluster value is initialized as *None* at the beginning of the algorithm.
The second dimension we added is Boolean and it indicates whether a point has been visited or not. A *False* indicates a data point has not been visited, a *True* indicates that a data point has

been visited. Hence, these values are initialized as *False* at the beginning of the algorithm. This means for each data point four values can be accessed. For each `point`:

1. `point[0]` ≡ x-location of the data point
2. `point[1]` ≡ y-location of the data point
3. `point[2]` ≡ cluster index of the data point, -1 if noise
4. `point[3]` ≡ visited state of the data point, False if unvisited, True if visited

The silhouette score calculation is implemented by the team on `silhouette.py`, which has commented code on how it works.

3. Results

In this section we will present the clusters and outliers produced by the DBSCAN algorithm for values 3,4 and 5 for the minimum neighbors and the ϵ values determined in subsection 2.2.
Each cluster has it's own color and marker shape so that they are easily distinguishable. Noise points are plotted as a cross sign with no color (white). The original data set without clustering is the following.

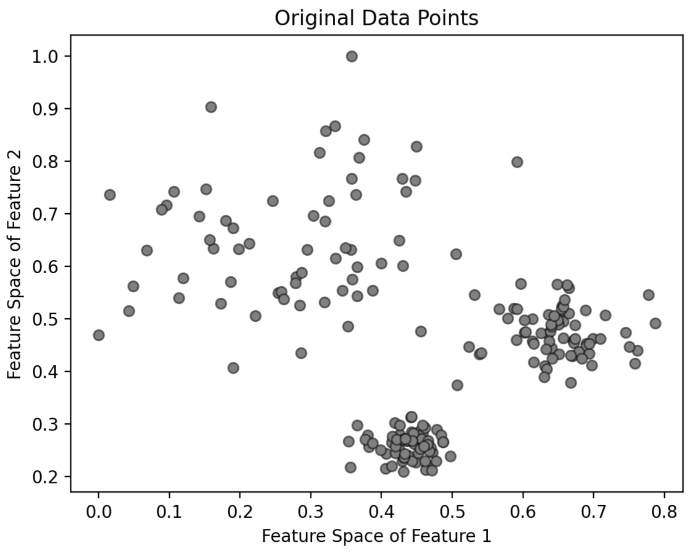

Figure 4: *All Data Points not organized in clusters.*

3.1. Minimum Neighbors = 3

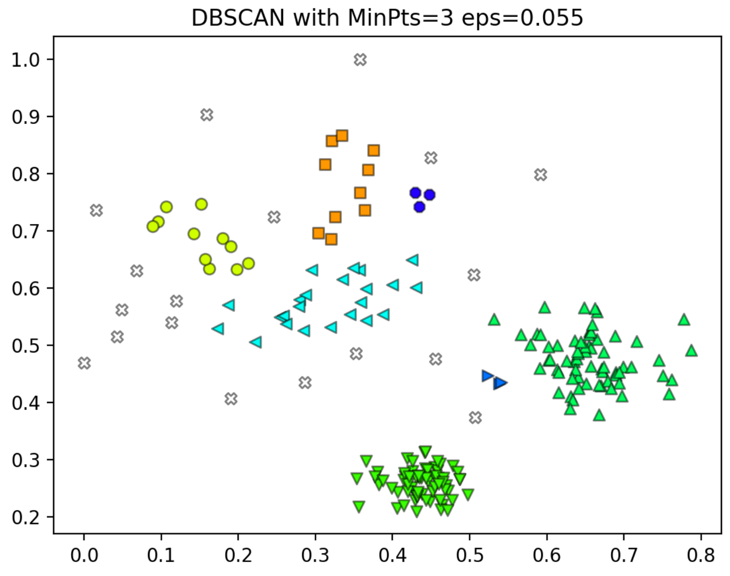

Figure 5: *Clusters and Outliers after running DBSCAN with* $\epsilon = 0.055$ *and* $min\_pts = 3$.

In Figure 5 we can see that we obtained seven different clusters. The data is split firstly into the bottom right and the top left. Then the bottom right data is again split into top right and bottom left. At the bottom right 2 dense clusters can be seen (green triangles pointing up and down) and one cluster with the minimum amount of points in it located between the other two clusters (3 dark blue triangles, 2 of them are nearly on top of each other). The top left data is split into four clusters. One of them is located at the bottom (blue triangles), one at the top (orange squares), one at the left (yellow circles) and a small one of minimum size at the right (blue circles). Outliers are spread around the top left cluster of data. There is a total of 18 outliers, they are somewhat evenly distributed throughout the data set.

3.2. Minimum Neighbors = 4

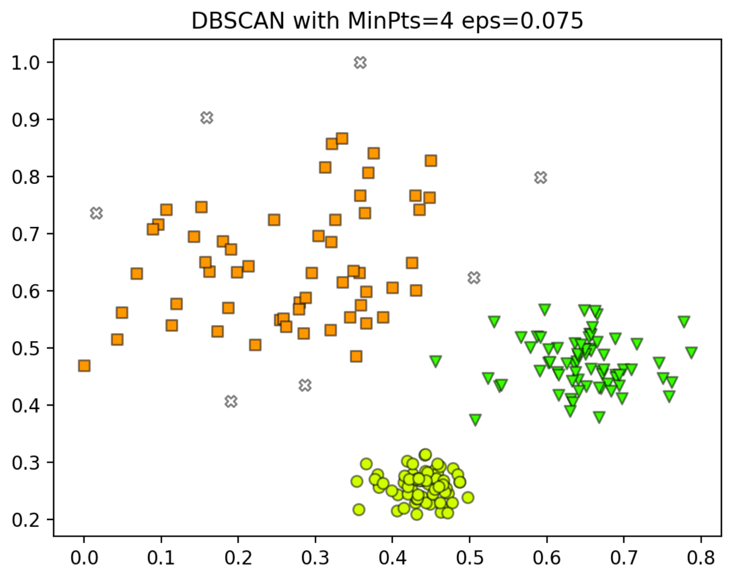

Figure 6: *Clusters and Outliers after running DBSCAN with* $\epsilon = 0.075$ *and* $min\_pts = 4$.

In Figure 6 we can see that we obtained three different clusters. The data is split firstly into the bottom right and the top left. Then the bottom right data is again split into top right (green triangles) and bottom left (yellow circles), both of these clusters are relatively dense. The cluster at the top left (orange squares) is more sparse. There is a total of seven outliers, they are evenly distributed around the sparse cluster.

3.3. Minimum Neighbors = 5

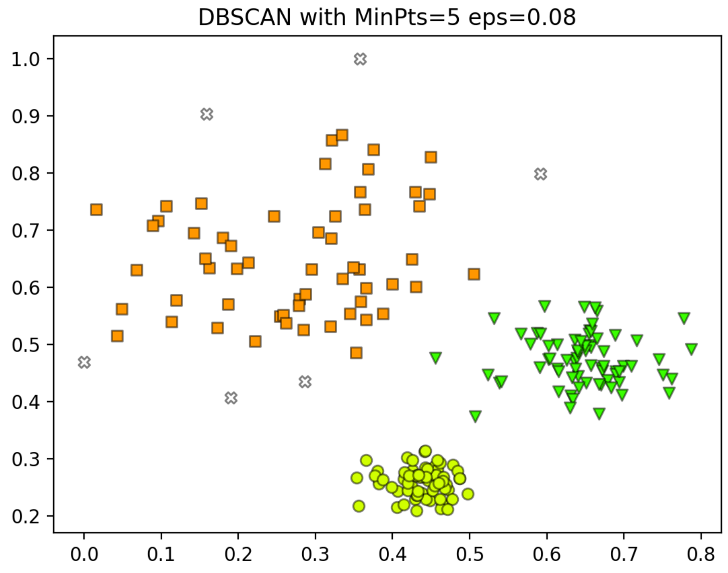

Figure 7: *Clusters and Outliers after running DBSCAN with* $\epsilon = 0.08$ *and* $min\_pts = 5$.

In Figure 7 we can see that we obtained three different clusters. The data is split firstly into the bottom right and the top left. Then the bottom right data is again split into top right (green triangles) and bottom left (yellow circles), again both of these clusters are relatively dense. The cluster at the top left (orange squares) is more sparse. There is a total of six outliers, they are evenly distributed around the sparse cluster. The clusterings in Figure 6 and Figure 7 seem very similar.

3.4. Silhouette Scores

The silhouette score of a clustering can give an insight on the effectiveness and goodness of the clustering. It's value is between -1 and 1, where a higher value indicates better matching clusters.

Minimum Neighbors / ϵ	3 / 0.055	4 / 0.075	5 / 0.08
Silhouette Score	0.532	0.695	0.700

Table 1: *Silhouette Scores for the different Minimum Neighbor values of 3, 4 and 5 and the corresponding* ϵ *values.*

In Table 1 we can see that the silhouette score for the minimum neighbor value of 3 is the lowest. For the values 4 and 5 for minimum neighbors the silhouette score is almost the same.

4. Discussion

As also mentioned in the previous report by our team on Hierarchical Clustering, just by observing Figure 4 without any other information, 3-4 clusters seem plausible. The positions of those clusters would be top left, bottom right and bottom center in Figure 4. However, just by looking, the noise points are not so clear to the eye, except for possibly one or two data points which are clearly further away from any plausible dense cluster.
Each choice of hyper-parameters seems to have generated meaningful clusters without any bugs or unreasonable clusters or noise points. So we can analyze each of them separately in **??**. Then the silhouette scores are analyzed (and also compared with Hierarchical clustering algorithm as a bonus). Finally, how the parameters change the behaviour of DBSCAN and what the best parameter choices are is discussed.

4.1. Cluster Results

4.1.1. Minimum Points = 3

With this hyper-parameters, result in Figure 5, the top left cluster is split into many small clusters and it seems off because the clusters look dense and closely packed. Meanwhile, the dark and bright green colored clusters towards the bottom look plausible and correct. These results can be tied into the density and ϵ choice. With a minimum points of 3, the clustering limits itself more than necessary, leading to more clusters. The expansion of a cluster stops more often than not because it can not reach other points within it's ϵ-range, due to it being very low at 0.055. Many points are identified as noise because they have less than 3 neighbours, including themselves. This DBSCAN resulted in the lowest silhouette score, meaning that it's clustering could be better. The context of the data and it's dimensions are not known, so it is not plausible to strongly judge the performance of this DBSCAN result without that context. But DBSCAN aims to perform clustering without any information or context, so in this case it can be said that this result is the worst clustering compared to the other results.
So the critical observation here is that, as DBSCAN uses the density as a means to form it's clusters, a sparse data set is not preferred. As it can be seen from Figure 4, the top left part of the data is sparse especially when compared to the other densely packed data towards the bottom and right. Thus, with these parameters, the algorithm struggles and makes an abundance of clusters for that part of the data set while clustering the other parts correctly, as seen in Figure 5.

4.1.2. Minimum Points = 4

By rule of thumb, choosing the minimum points as $2 * dimension$ is wise[1]. In that case, minimum points of 4 should be a good choice. As seen from Figure 6, the clusters are clearly separated and there is not an abundance of noise points. With it's good silhouette score, it can be said that these parameters worked very well with this data set.

4.1.3. Minimum Points = 5

This hyper-parameter choice result in Figure 7, has nearly the same results as the ones with minimum points as 4, Figure 6. There are only a few points which are clustered differently. So then, based on our discussion of minimum points = 4, this DBSCAN is also good with the data set. It is worth noting that the ϵ value has been increased by 0.005 to compensate for the stricter neighborhood of size 5.

[1]Campello, Ricardo J. G. B.; Moulavi, Davoud; Zimek, Arthur; Sander, Jörg (2015). "Hierarchical Density Estimates for Data Clustering, Visualization, and Outlier Detection". ACM Transactions on Knowledge Discovery from Data. 10 (1): 1–51. doi:10.1145/2733381. ISSN 1556-4681. S2CID 2887636.

4.1.4. Noise Points (Outliers)

From all DBSCAN result plots combined, some points can be identified towards the top and other areas in which there are noise points. When looking at the top left cluster, it is seen that it is less dense and more sparse compared to the others, so then the outliers close to it could be a part of that cluster, but due to DBSCAN's nature and detection of outliers, those points are not added to that, nor any, cluster.

4.2. Silhouette Scores

Looking at Table 1, the different silhouette scores for each minimum neighbors selection can be seen. As mentioned, when it is 3, we have the lowest silhouette score of 0.532. When we increase the minimum points and also ϵ, there is a drastic increase in the score, of about 0.2. Comparing these results with hierarchical clustering (previous assignment), they are nearly the same, with a difference of about 0.04, and that is surely tied to the fact that noise points are ignored in DBSCAN, the silhouette scores are higher (better). While in hierarchical clustering those are also parts of clusters, so the silhouette score is lowered not drastically but significantly compared to these results in Table 1. The lower difference for results of DBSCAN and hierarchical clustering could also be due to the outliers not being so clearly identifiable. If the outliers were more clear than it can be expected that they affect the hierarchical clustering silhouette score more severely.

4.3. Parameters and Best Choices

Firstly, minimum points (in neighbourhood) is decided on, before actually deciding on ϵ with the help of K-Nearest Neighbour search. The ϵ can be decided using the elbow method on the K-Nearest neighbours search graph as seen in Figure 1, Figure 2 and Figure 3. Choosing a value approximately close to the elbow point is expected to provide good density-based clustering results and it did so in this analysis. With our current selections of ϵ based on the nearest neighbour search and trials:

1. For minimum points 3, values around $\epsilon = 0.055$ was guessed from the K-nearest neighbours. For this case, through intuition it can be proposed that ϵ could be increased because it looks like the clusters can not expand and the ϵ is too small. After increasing it to $\epsilon = 0.08$, the silhouette score increases from 0.532 to 0.696 and the cluster is nearly the same as the one in Figure 6. However, based on the rule of thumb (subsubsection 4.1.2), using 3 as the minimum neighbours is not ideal. Still the significant improvement of results by increasing ϵ for this small value of minimum neighbors is worth taking note of. So then, the decision of $\epsilon = 0.08$ as the best choice is supported.

2. For minimum points 4, the $\epsilon = 0.075$, which is very close to 0.08, and the results achieved are plausible and optimal.

3. For minimum points 5, with an $\epsilon = 0.08$, the largest silhouette score was achieved, with a value of 0.700. Nevertheless, the difference between this silhouette score and that achieved by setting minimum neighbors to 4 is only 0.005. So both this parameter combination, as well as $min\_pts = 4$ and $\epsilon = 0.075$ are plausible.

In conclusion, the best hyper-parameter selection for this data set when applying DBSCAN, $\epsilon \approx 0.008$ and $min\_pts = 4, 5$, as it provides good clusters and good silhouette scores with all of the minimum neighbor values. Also, for all K-nearest neighbour plots 0.008, if not the elbow point, is close to the plausible elbow points so it is also suggested from that search analysis. If $\epsilon = 0.008$ resulted in too few clusters (1 or 2), then it would have meant that it was too large of a value, but clearly that is not the case as seen from the cluster results. When it comes to the minimum neighbor points, it is not so certain as to which value is the best. Still, 4 is the preferable choice, since it is not as strict as 5, not as flexible as 3, and is supported by previous research.

5. Contribution

Both team members attended the lab session for this assignment and both agree that the contribution to this assignment was equal and fair.

5.1. Code

The development of the code for the assignment was done in a lab session. Therefore, the contribution to the code base was entirely equal for both group members.

5.2. Report

Sections were split in half and thus the report was worked on by both members. After looking over the report together one last time, it was finalized and ready to submit.

6. Code Appendix

6.1. main.py

```python
from dbscan import *
from knn import *
from plotting import *
from silhouette import *
import copy


def prepare_data(data):
    for p in data:
        p.append(None)
        p.append(False)


if __name__ == "__main__":
    # Read data
    path = "data/data_clustering.csv"
    read_data = np.loadtxt(open(path, "r+"), delimiter=",")
    read_data = read_data.tolist()
    data =  copy.deepcopy(read_data)

    # Prepare data format
    prepare_data(data)

    # K-nearest neighbors search
    k_nearest_neighbours(read_data)

    # Parameters
    # (Min pts) | (eps estimate / guess from K-NN search)
    #   3 | ~0.055
    #   4 | ~0.075
    #   5 | ~0.08
    all_min_pts = [3,4,5]
    epsilons = [0.055, 0.075, 0.08]

    # For each min_pts and eps combination, run DBSCAN and display
   results.
    for i in range(len(all_min_pts)):
        # Initialize data and parameters
        dbscan_data = copy.deepcopy(data)
        min_pts = all_min_pts[i]
        eps = epsilons[i]

        # Call dbscan
        dbscan(dbscan_data, eps, min_pts)

        # Results and Plots
        plot_data_points(dbscan_data, eps, min_pts)

        # Silhouette Score
        s = calculate_silhouette(dbscan_data)
        print(f"Silhouette score with min_pts = {min_pts} and eps = {
   eps} -> {s}")
```

6.2. DBSCAN.PY

```
# Find given data point in data and update it.
def update_data(data, _p):
    for idx in range(len(data)):
        if data[idx][0] == _p[0] and data[idx][1] == _p[1]:
            data[idx] = _p


# Set union for joining neighbourhood 2 to neighbourhood 1
def join(hood1, hood2):
    # For each point in hood2, append it to hood1 if it does not
   already exist.
    for p in hood2:
        found = False
        for _p in hood1:
            if p[0] == _p[0] and p[1] == _p[1]:
                found = True
                break
        if not found:
            hood1.append(p)
    return hood1


# Apply DBSCAN on data with given parameters
def dbscan(data, eps, min_pts):
    cluster = 0
    # For each data point in data
    for p in data:
        if not p[3]:
            p[3] = True # Mark as visited
            neighbor_pts = region_query(data, p, eps)
            if len(neighbor_pts) < min_pts:
                p[2] = -1  # Mark as noise
            else:  # is core
                cluster = cluster + 1  # next cluster
                expand_cluster(data, p, neighbor_pts, cluster, eps,
   min_pts)  # expand from core


def expand_cluster(data, p, neighbor_pts, cluster, eps, min_pts):
    p[2] = cluster  # Add p to cluster
    # For each neighbour of p
    for _p in neighbor_pts:
        if not _p[3]:
            _p[3] = True  # Mark as visited
            # Update in data itself
            _neighbor_pts = region_query(data, _p, eps)  # Get
   neighbours and add to current neighbours
            if len(_neighbor_pts) >= min_pts:  # if _p is core add its
    neighbours
                neighbor_pts = join(neighbor_pts, _neighbor_pts)
        if _p[2] is None or _p[2] == -1:
            _p[2] = cluster  # Add to cluster


def region_query(data, p, eps):
    neighborhood = []
    for _p in data:
        if distance(p, _p) <= eps**2:
            neighborhood.append(_p)
    return neighborhood


# Squared Euclidean distance
def distance(_p, p):
    return (_p[0] - p[0]) ** 2 + (_p[1] - p[1]) ** 2
```

6.3. KNN.PY

```
from sklearn.neighbors import NearestNeighbors
import numpy as np
from plotting import plot_knn


def sort_distances(distances):
    # Use column 0 as sum of columns
    # Because column 0 is always 0 for all (nearest neighbour is
   itself)
    for distance in distances:
        distance[0] = sum(distance[1:len(distance)])
    return distances[np.argsort(distances[:, 0])]


# 3 plots of knn search for k = 3,4,5 with marked thresholds at elbow
   points
def k_nearest_neighbours(data):
    # Obtain distance matrix for k
    # Use k = 4,5,6 because itself is counted as a closest neighbour
   when using sci-learn
    # Should we also?
    for k in range(3, 6):
        neighbours = NearestNeighbors(n_neighbors=k, algorithm='auto').
   fit(data)
        distances, indices = neighbours.kneighbors(data)
        # Sort in ascending order
        sorted_distances = sort_distances(distances)
        # sorted_distances = sorted_distances[:,1:] removes 0th column
        # Plot
        plot_knn(sorted_distances, k)
```

6.4. SILHOUETTE.PY

```
import math


# Normal Euclidean distance (so we can also compare with assignment 5
   silhouette scores)
def distance(_p, p):
    return math.sqrt((_p[0] - p[0]) ** 2 + (_p[1] - p[1]) ** 2)


# Average distance of a point to points within cluster
def calculate_a_i(p, cluster):
    distances = []
    for other_point in cluster:
        if p[0] == other_point[0] and p[1] == other_point[1]: continue
        distances.append(distance(p, other_point))
    return sum(distances) / len(distances)


# Average distance of a point to points within closest cluster
def calculate_b_i(p, closest_cluster):
    distances = []
    for other_point in closest_cluster:
        distances.append(distance(p, other_point))
    return sum(distances) / len(distances)
```

```python
# Silhouette score
def calculate_s_i(a, b):
    return (b-a)/max((a,b))


def find_closest_cluster(p, p_cluster, clusters):
    closest_distance = float('inf')
    closest_cluster = None
    clusters.remove(p_cluster)

    for cluster in clusters:
        for other_point in cluster:
            d = distance(p, other_point)
            if d < closest_distance:
                closest_distance = d
                closest_cluster = cluster
    return closest_cluster


def number_is_found(x, arr):
    for val in arr:
        if val == x:
            return True
    return False


def find_cluster_numbers(data):
    cluster_numbers = []
    for p in data:
        if number_is_found(p[2], cluster_numbers):
            continue
        cluster_numbers.append(p[2])
    return cluster_numbers


def calculate_silhouette(data):
    # Silhouette score
    # Create clusters from the data
    clusters = []
    cluster_numbers = find_cluster_numbers(data)
    for cluster_num in cluster_numbers:
        if cluster_num == -1: continue  # skip noise
        c = []
        for p in data:
            if p[2] == cluster_num:
                c.append([p[0], p[1]])
        clusters.append(c)
    # Now, clusters[0..len(clusters] are the clusters.
    # Calculate silhouette score for each data point
    silhouette_scores = []
    for cluster in clusters:
        for p in cluster:
            # Calculate a(i)
            a = calculate_a_i(p, cluster)
            # Calculate b(i)
            closest_cluster = find_closest_cluster(p, cluster,
   clusters.copy())
            b = calculate_b_i(p, closest_cluster)
            # Calculate s(i)
            s = calculate_s_i(a, b)
            silhouette_scores.append(s)

    # Mean of silhouette scores is the result
    return sum(silhouette_scores) / len(silhouette_scores)
```

6.5. PLOTTING.PY

```
import numpy as np
from matplotlib import pyplot as plt

# Markers to see different clusters
markers = [".", ",", "o", "v", "^", "<", ">", "8", "s", "p", "P", "*",
    "h", "H", "D", "d", "|", "_", ".", ",", "o", "v",
           "^", "<", ]


# Source: https://stackoverflow.com/questions/14720331/how-to-generate-
   random-colors-in-matplotlib
def get_cmap(n, name='hsv'):
    """
    Returns a function that maps each index in 0, 1, ..., n-1 to a
   distinct
    RGB color; the keyword argument name must be a standard mpl
   colormap name.
    """
    return plt.cm.get_cmap(name, n)


def plot_data_points(data, eps, min_pts):
    fig, ax = plt.subplots()
    ax.title.set_text(f"DBSCAN with MinPts={min_pts} eps={eps}")
    cluster_amount = 1000
    cmap = get_cmap(cluster_amount)
    edge_color = (0, 0, 0, 0.5)
    for p in data:
        cluster = p[2]
        # If noise
        if cluster == -1:
            color = "white"
            marker = "X"
        else:
            color = cmap(cluster*100)
            marker = markers[cluster]
        ax.scatter(p[0], p[1], color=color, marker=marker, edgecolors=
    edge_color)
    plt.show()


def plot_knn(distances, k):
    # Trick: Plot column 0 of sorted distances because col 0 is the
    sum of that point's distances
    plt.plot(distances[:, 0])
    plt.title(f"{k}-Nearest Neighbours Plot")
    plt.ylabel("Distance")
    plt.xlabel("Points (sorted by total distance to NN)")
    # Cutoffs for Ks
    #   3 | ~0.055
    #   4 | ~0.075
    #   5 | ~0.08

    # Threshold estimates for elbow method
    if k == 3:
        plt.axhline(0.055, linestyle='--', color='black')
    if k == 4:
        plt.axhline(0.075, linestyle='--', color='black')
    if k == 5:
        plt.axhline(0.08, linestyle='--', color='black')
    plt.show()
```