Introduction to Intelligent Systems
Lab Session 3

Group 17
Tobias Pucher (s5751659) & Thanakit Chaichanawong (s5752094)

October 4, 2023

Assignment 3

1. Introduction

The amount of data for analysis purpose is getting larger and larger as time goes on and technology improves. Therefore traditional analysis on raw data is going to be harder as well. Fortunately, with the introduction of machine learning, there exist various algorithms, functions, and libraries to help mitigate such problems. One of the example is clustering and cluster analysis.

Cluster analysis is the analysis of data by partitioning groups of data points by the factor of similarity between data points. More similarity between data points means more probability they will be in the same cluster, and vice versa, the criteria is depending on the configuration or methods used for determining such similarity. Cluster analysis is considered to be a unsupervised learning technique. The analysis is static with the result variation dependent on the algorithm used and of course the data.

There are many different clustering algorithms available with different purposes and results. Centroid based clustering is the analysis with non-hierarchical cluster but high efficiency. K-means is a common algorithm of Centroid based analysis. Next, Density based clustering is the clustering of data points by detecting the area where data is similarly dense. DBSCAN is one of such density algorithm which allows to label points as part of clusters but also as part of noise. od is the focus of this assignment.

2. Methods

2.1. General idea

The DBSCAN (TODO) algorithm tries to cluster points based on their density. To do so a suited distance measurement is needed, in this case the squared euclidean distance. The algorithm looks at all points in a data-set sequentially. For each point it decides if it is a core point or just a noise point. To do so it gets all neighbors of the point in question using the chosen distance method. This euclidean distance epsilon neighborhood can be thought of as all points surrounding the point in a circle with a radius of epsilon. If a point has less points than required by minPts it is a noise point. Otherwise it is the first point of a new cluster which gets expanded by taking all core-neighbor points of this point and all neighbors which expands only on other core neighbors and stops expanding on noise neighbors. This forms a cluster. The algorithm marks all these points as visited and continues with all other unvisited points in the same manner until all points are marked as noise or as part of a cluster.

2.2. DBSCAN Algorithm implementation

The methods needed to implement the DBSCAN algorithm are described in this section. The DBSCAN algorithm was implemented in `python` (jupyter notebook) using several helper methods. Their functionality as well as the importing of the the dataset is elaborated in the following subsections.

In the following code the complete DBSCAN algorithm is shown split up into smaller snippets. The helper methods which are described seperately. The full code of the DBSCAN function can be found in the end of this subsection.

First the return types of the DBSCAN algorithm are defined. The type int inside the lists is the index of a point in the enumterated dataset.

```
Noise : TypeAlias = list[int]
Cluster  : TypeAlias = list[int]
DBSCANResult : TypeAlias = tuple[Noise, list[Cluster]]
```

The DBSCAN method takes in the desired dataset and two parameters. The parameter `eps` is the float value representing the circular epsilon region surrounding a data point. The parameter `minPts` represents the minimum number of neighborhood points needed for a particular point to be considered a core point.

The method defines three lists to keep track of the points. The `noise` list tracks the indices of the data points that are flagged to be noise. The `clusters` list contains all found clusters. Each found cluster is a lists of points that correspond to it. Another list called `visited` keeps track of every data point in the dataset and stores if it was already visited.

The algorithm starts by going over every data point in the enumerated dataset. For each point it checks if it was already visited by checking the index against the visited list. In case of a point being already visited the loop continues with the next point.

```
for i, p in enumerate(D):
    if visited[i]: continue
    visited[i] = True
```

When an unvisited point is found its neighbors are retreived with the method `regionQuery` which is described in detail in subsection 2.4. These neighbor points of the epsilon neighborhood of the point `p` are taken into account for deciding if `p` is a core point because only core points can form a new cluster. This is done by comparing the number of neighbors of `p` to `minPts` as seen by the variable `is_core_point`.

```
neighs = regionQuery(D, p, eps)
is_core_point = len(neighs) >= minPts
```

If the point p is not a core point it must per definition be a noise point. In this case the points index is appended to the list noise and the algorithm continues with the next unvisited point in the dataset. If the point p is indeed a core point a new cluster is formed. To do so a list 'C' is created which tracks all indices of points of this new cluster and is itself appended to the 'clusters' list.

```
if is_core_point:
      C = []
      clusters.append(C)
      expandCluster((i,p),neighs,C,eps,minPts,clusters,visited)
else:
      noise.append(i)
```

To find all points of the cluster originating from p the method expandCluster is called. It is described in subsection 2.3. Upon returning from the method call to expandCluster the algorithm continues with the next point in the dataset. Because expandCluster also marks points as visited it may be possible that an already visited point is encountered, which is handled at the beginning of the loop. Upon visiting every point the loop ends and the algorithm returns a tuple containing the lists noise and clusters according to the type DBSCANResult. These lists contain the index of every data point and the length of the combined flattened result should equal the length of the original dataset.

Below is the full DBSCAN method code:

Listing 1: *DBSCAN algorithm method*

```
def DBSCAN(D, eps, minPts) -> DBSCANResult:
  visited = [False] * len(D) # allocate predefined length every node
    has to be visited
  noise = []
  clusters = []
  for i, p in enumerate(D):
    if visited[i]: continue
    visited[i] = True

    neighs = regionQuery(D, p, eps)
    is_core_point = len(neighs) >= minPts
    if is_core_point:
      C = []
      clusters.append(C)
      expandCluster((i,p),neighs,C,eps,minPts,clusters,visited)
    else:
      noise.append(i)

  return (noise, clusters)
```

2.3. Expanding a cluster

To expand a cluster (corepoint) to as many neighbor points as possible the below method is defined. The method receives a tuple of the current point p and its index i as well as the neighbors of p neighs, the cluster C, the parameters eps and minPts and references to the list of all clusters and noise. The expandCluster method is described in the code below.

The method adds the index `i` of the origin point of the cluster `C`. It then loops over the already indexed neighborhood points of `p`. For each unvisited neighbor point the visited flag is set to `True` and its neighborhood points are also queried using the already discussed method. These neighbor hood points are added to the list of neighbors of the original point, therefore expanding the loop. If the neighborhood point is not in any cluster its index is assigned to the cluster. Then the loop continues with the next point in the nieghbors list until all neighbors and expanded neighbors have been checked. The list returns nothing as it operates on the references of the provided lists.

Listing 2: *expandCluster method*

```
def expandCluster(P: Point, NeighborPts, C, eps, minPts, clusters,
    visited):
  idx, _ = P
  C.append(idx)
  for i, p in NeighborPts:
    if not visited[i]:
      visited[i] = True
      neighs = regionQuery(D, p, eps)
      if len(neighs) >= minPts:
        NeighborPts += neighs
    # add point to cluster
    if i not in list(chain.from_iterable(clusters)):
      C.append(i)
```

2.4. Neighborhood of a point

The following method `regionQuery` finds the neighborhood points of a point according to its Epsilon-neighborhood. The method takes in the dataset `D`, a point `P` and the parameter `eps` representing the epsilon value of the region. The method tracks the found neighbors using the list `N` which holds tuples of the found point and its index in the enumerated dataset. The method loops over all points including the point `P`. For each point it decides according to a distance measure and the epsilon value if the point lies within epsilon distance. In this case we chose the simple euclidean distance which is defined in another method in subsection 2.5. If the squared euclidean distance of `p` and `P` is smaller or equal to the epsilon parameter than the point p lies in the epsilon neighborhood of `P` and is added to `N` with its corresponding index in the data set. After checking all points the method returns the list `N` containing indicies for `P` and its neighbors.

```
def regionQuery(D, P, eps):
  N = []
  for i, p in enumerate(D):
    distance = euclidean_distance(p, P)
    if distance <= eps:
      N.append( (i,p) )
  return N
```

2.5. Euclidean distance

A method is defined for calculating the euclidean distance of two points. This method takes two points as its arguments. Each point is a list or `np.array` of arbitrary dimension. According the the squared euclidean distance formula the method returns the square root of the sum of differences of features of the two vectors (points). It uses higher order functions to accomplish this task. The result is a positive number.

```
def euclidean_distance(p1, p2):
    vals = list(zip(p1, p2))
    return (sum(map(lambda x: (x[0]-x[1])**2, vals))) ** 0.5
```

2.6. k-NN search for eps value

```
from sklearn.neighbors import NearestNeighbors
neigh = NearestNeighbors(n_neighbors=n).fit(D)
distances = neigh.kneighbors(D)[0]
distances = np.sort(distances[:, -1])
```

Starting from setting n variable to the minPts for each k-NN run. Then the data D will be fitted in the Nearest-Neighbors algorithm with number of neighbor equal to the given minPts. Then the result will be stored in the "distances" and sorted from ascending.

```
eps_threshold = 0.029
```

After the graph has been constructed and observed the appropriate elbow point based on the change of curve of the graph by eyes, the threshold will be set and re-run to draw the horizontal line intersecting the distance curve.

```
plt.plot(distances)
plt.axhline(y=eps_threshold, color='r', label=f'eps Threshold (eps={
    eps_threshold:.2f})')
plt.xlabel('Index')
plt.ylabel(f'{n}-Nearest Neighbor Distance')
plt.title(f'Elbow Point Search k={n}')
plt.legend()
plt.show()
```

After obtain the eps threshold, the plot can be constructed. The plot consist of the distance graph as a blue line with exponential-like curve from index 0 to 200 (x-axis) and distance from almost 0 to beyond 0.2. The red line indicating the threshold that has been chosen.

2.7. Obtaining plot of clustering

```
res = DBSCAN(D, eps, minPts)
clusters = res[1]
coloring = [0] * 200
color=1
for cluster_no in clusters:
  color+=1
  for cluster_index in cluster_no:
      coloring[cluster_index] = color
```

To obtain the plot of clustering, DBSCAN algorithm is the first function to run to obtain the cluster and noise. Then the noise is filtering out for labeling. In labeling array, 0 represents the noise and other numbers represents the data points belong to the given cluster. The cluster label is obtained by setting array of 0s to be other numbers with respect to the index from the clusters and index of the clusters in the array produced from filtering process. The 2 loops is implemented, the outer loop is for repeat in different clusters, the inner loop is for labeling throughout the cluster.

```
plt.scatter(D[:, 0], D[:, 1],c=coloring, cmap='viridis')
plt.title(f"Clustered data points using {eps} and K={minPts}")
plt.show()
```

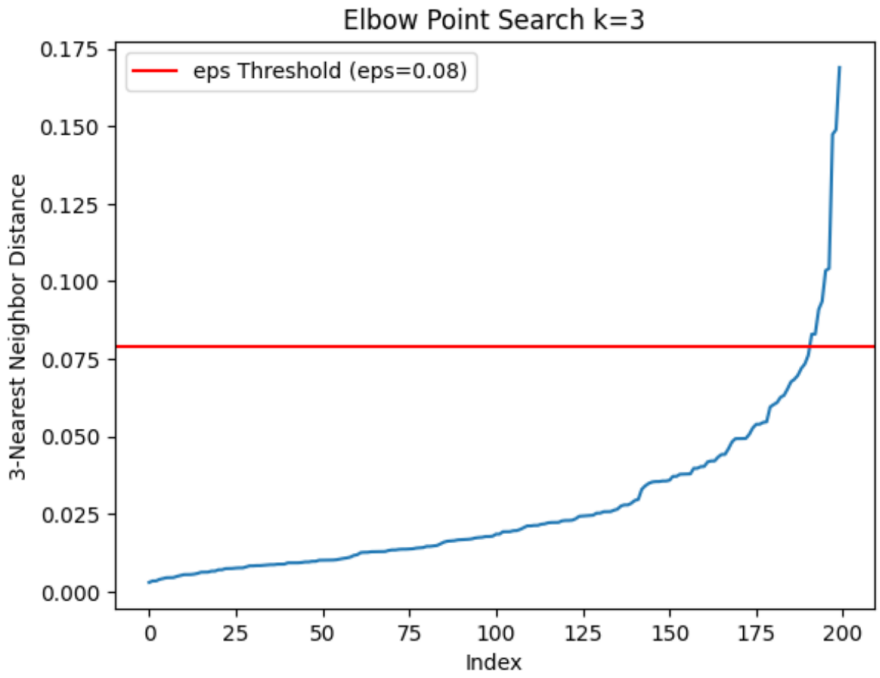
After obtaining the cluster label (in this case in array "coloring") the graph can be constructed. Similar to the previous assignment, the data array can be obtained array D and pass to the plotting function with labeling array.

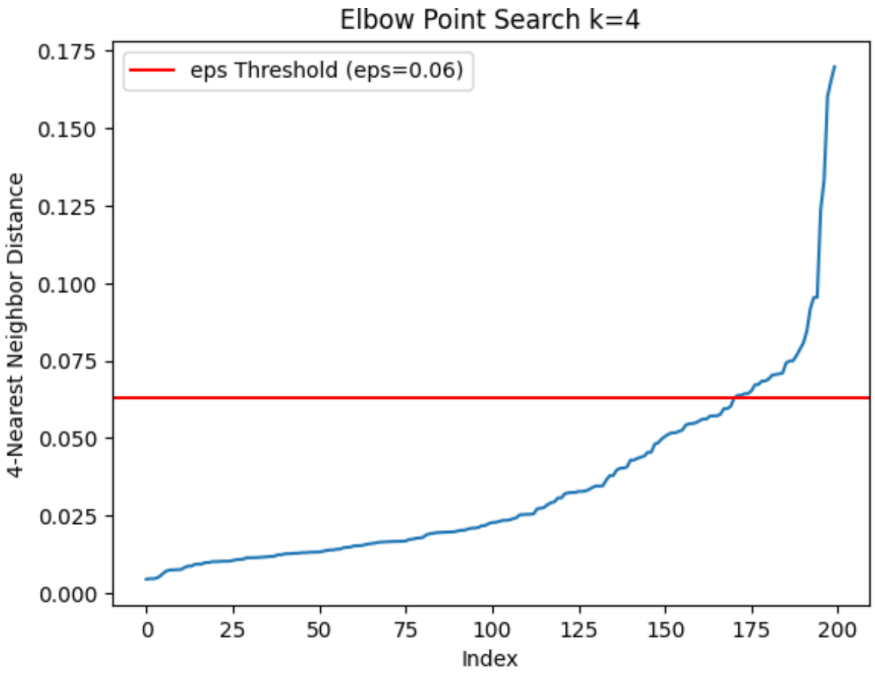
2.8. Silhouette score

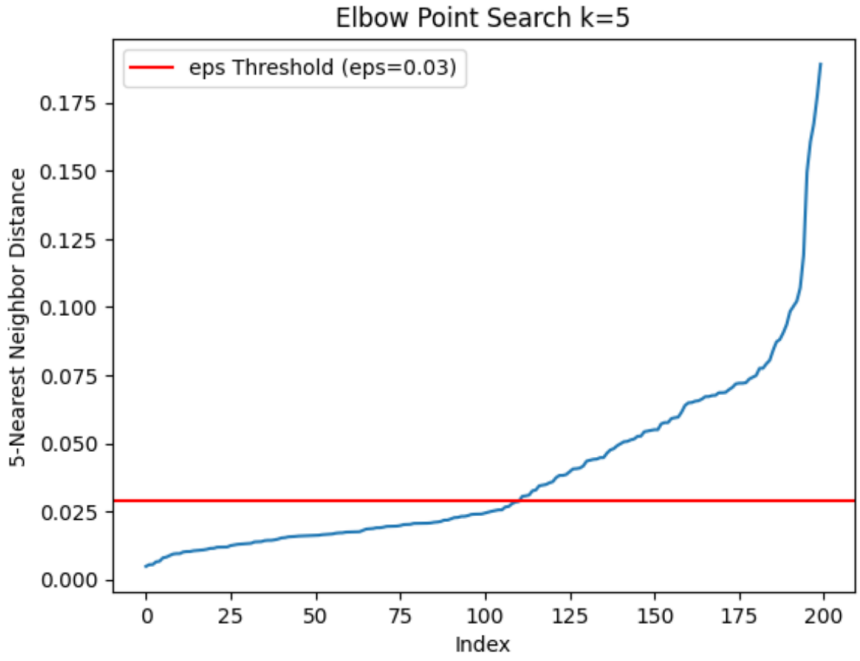
Silhouette score is a internal metric for determining the goodness of a clustering method. The score is calculated based on the intra-cluster distance and nearest cluster distance. The intra-cluster distance should be relatively lower while nearest cluster distance should be relatively higher for the score to be considered good. The score ranges from -1 to 1 and it is the average distance of all clusters. The arrays taken by this function are the label and data arrays.

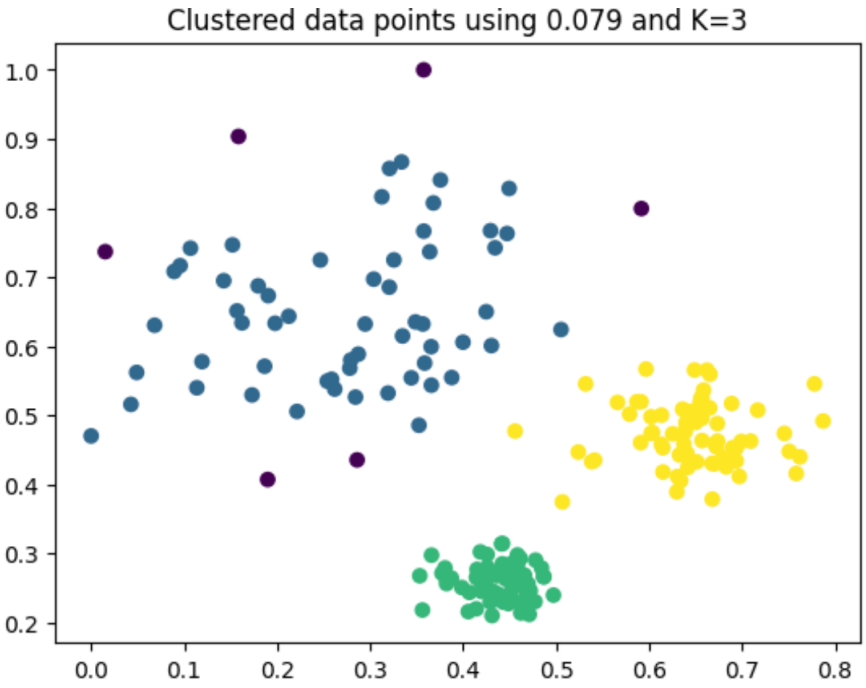
```
from sklearn.metrics import silhouette_score
silhouette_avg = silhouette_score(D, coloring)
```

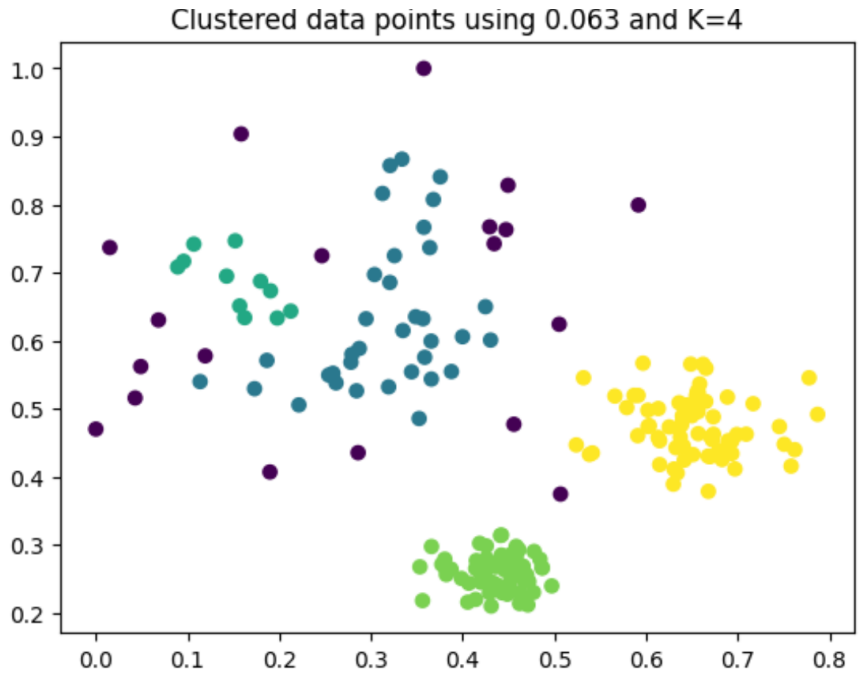
3. Result

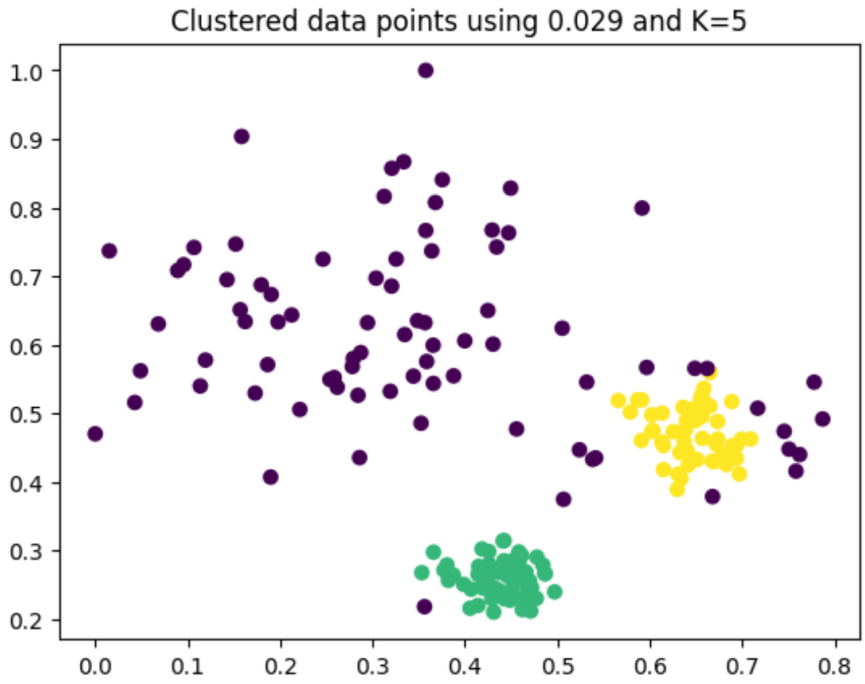



Elbow Point Search k=4
eps Threshold (eps=0.06)
4-Nearest Neighbor Distance
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0.000
0
25
50
75
100
125
150
175
200
Index


Elbow Point Search k=5
eps Threshold (eps=0.03)
5-Nearest Neighbor Distance
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0.000
0
25
50
75
100
125
150
175
200
Index


Clustered data points using 0.079 and K=3
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8


Clustered data points using 0.063 and K=4
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8



The following table shows the silhouette score for each min-pts value

eps	min-pts	Silhouette score
0.079	3	0.6062
0.063	4	0.5586
0.029	5	0.5201

4. Discussion

First and foremost, there is significant difference in the Silhouette score when changing the elbow point or eps value due to the fact that clustering is totally different throughout all values from the graph. When plotting the Silhouette score with the eps value on the other axis, the score is similar to a bell-curved, thus there is best value around the middle point of eps value on different each minPts.

In this experiment, many eps value has been tested on each 3 minPts number. And the best result could be obtained from searching the value is around 0.60 silhouette score with respectable tuning definition. To obtain a higher score, the eps value will have to be fine-tuned in the 5 decimal points level.

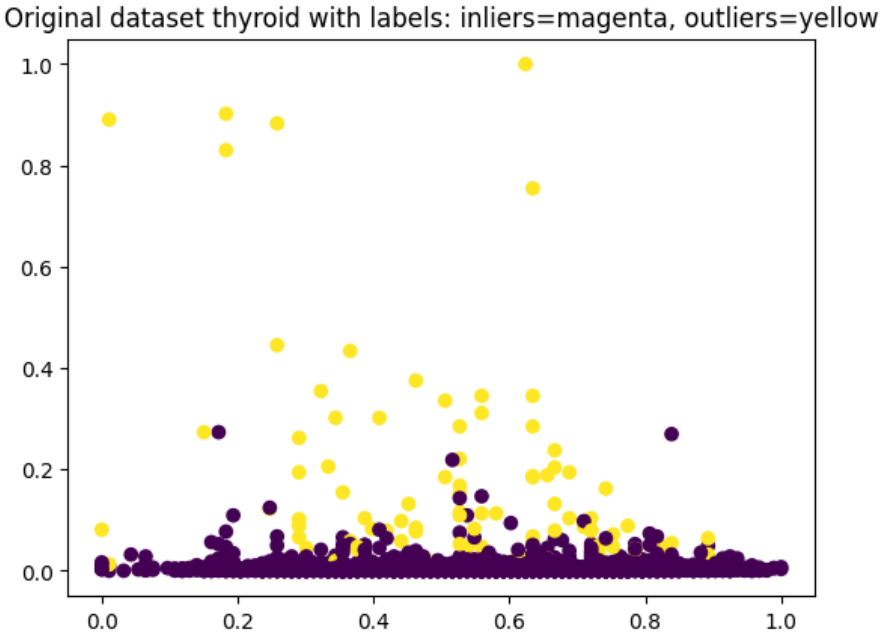
We can observed that the eps of the minPts = 3,4 is significantly different from minPts = 5 according to the Silhouette score. This could be the result from having different number of cluster present on the graph.

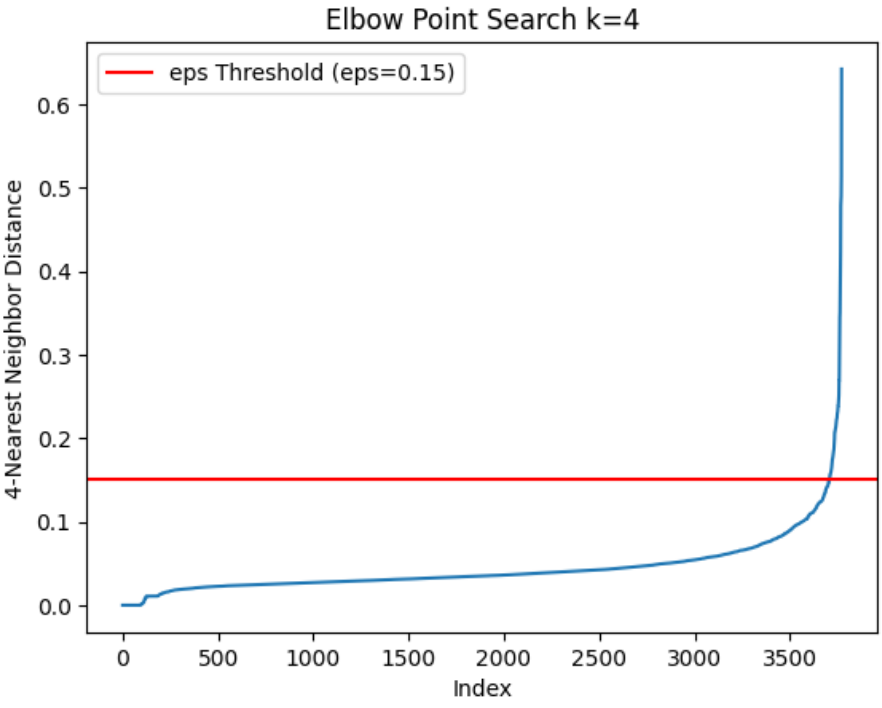
The most optimized minPts and eps value from this experiment is min-pts = 3 and eps = 0.079, coming with the Silhouette score of 0.6062

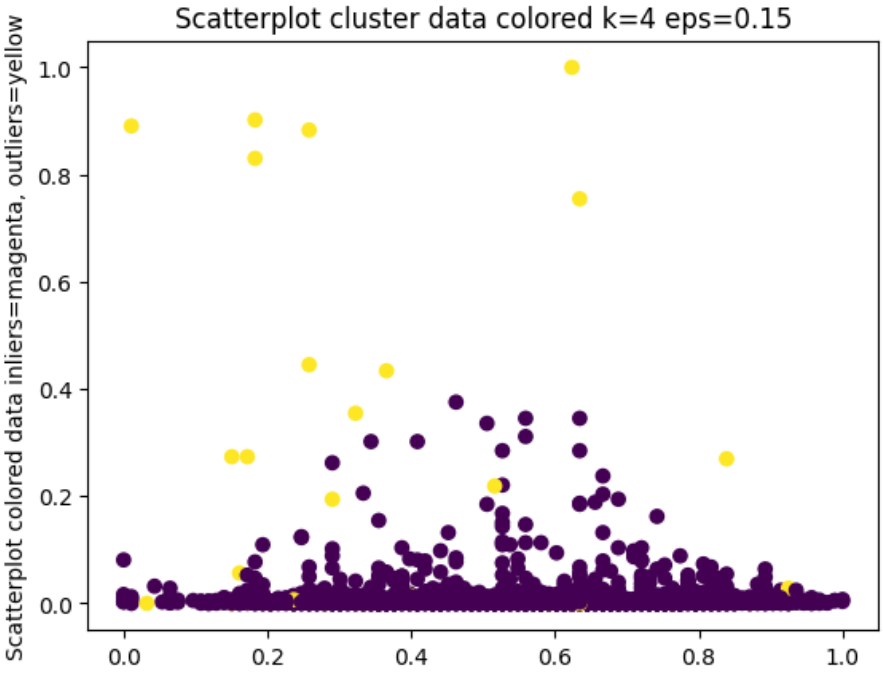
5. Bonus

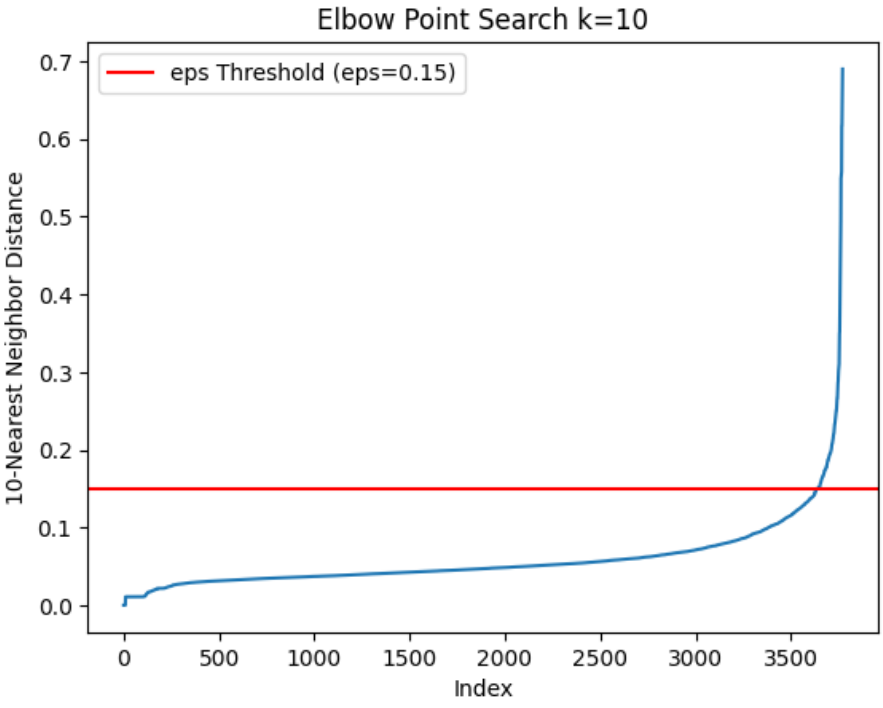
Bonus exercise on the Thyroid dataset.

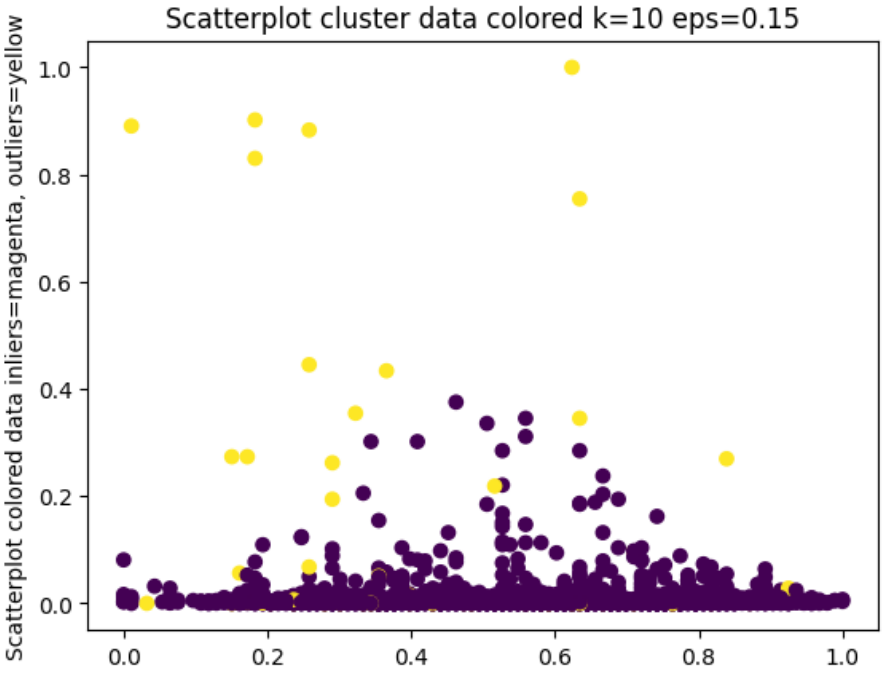
5.1. Results

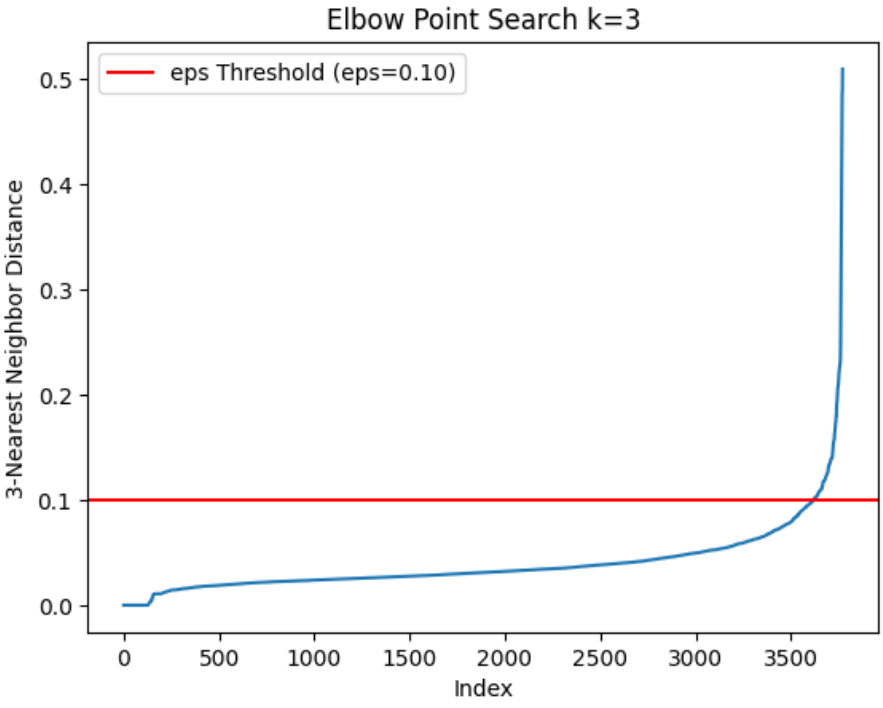

Original dataset thyroid with labels: inliers=magenta, outliers=yellow
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

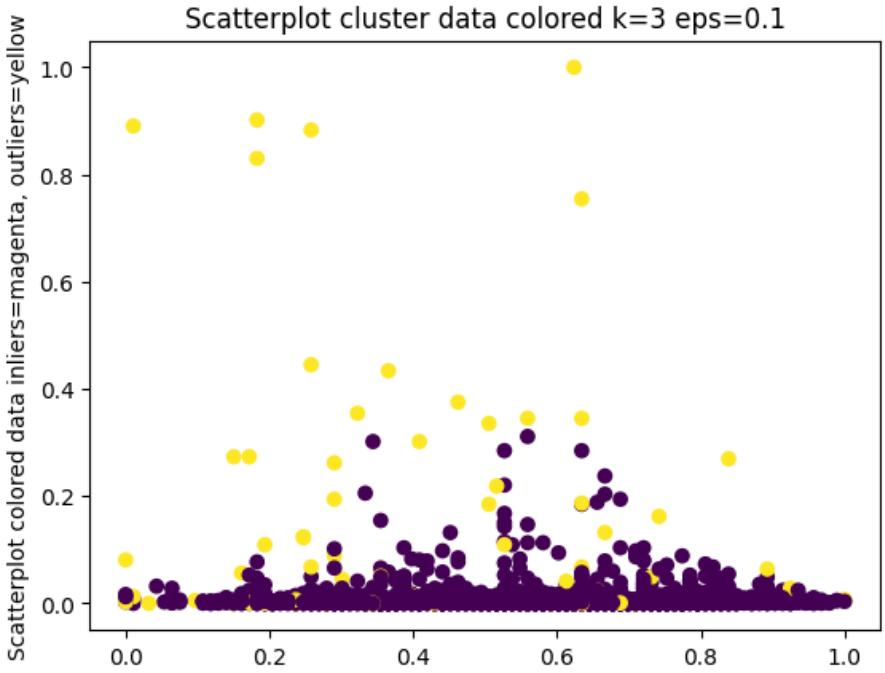

Elbow Point Search k=4
eps Threshold (eps=0.15)
4-Nearest Neighbor Distance
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0
500
1000
1500
2000
2500
3000
3500
Index


Scatterplot cluster data colored k=4 eps=0.15
Scatterplot colored data inliers=magenta, outliers=yellow
0.0
0.2
0.4
0.6
0.8
1.0


Elbow Point Search k=10
eps Threshold (eps=0.15)
10-Nearest Neighbor Distance
Index
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0
500
1000
1500
2000
2500
3000
3500


Scatterplot cluster data colored k=10 eps=0.15
Scatterplot colored data inliers=magenta, outliers=yellow
0.0
0.2
0.4
0.6
0.8
1.0


Elbow Point Search k=3
eps Threshold (eps=0.10)
3-Nearest Neighbor Distance
0.0
0.1
0.2
0.3
0.4
0.5
0
500
1000
1500
2000
2500
3000
3500
Index



5.2. Metrics

minPts	eps	Precision	Recall	F-score
4	0.15	0.2075	0.1182	0.1506
3	0.1	0.256	0.3440	0.2935
10	0.15	0.1707	0.1505	0.16

5.3. Bonus: Discussion

We have found from a visual perspective that for such a dataset the dbscan method is not well suited. In most test cases the truth labels are far different and more distinct an might not be well described using density. This intuitive idea is also shown in the several metrics that were run using `sklearn.metrics` package for calculating the Precision, F-score and Recall metrics. The results seem similar to the original but with some noticable differences in the labeled outliers. It seems like the labels /clusters in the original dataset might not represnt data that is dependent on the density of the point as such, but maybe other more hidden measurements or distances.

6. Contributions

For Thanakit's responsible are coding the method of k-NN algorithm, and constructing the plot by creating labeling array. In the report, the responsible are methods that Thanakit done on the code, result, and discussion.

Tobias did the bonus exercise as well as the methods section and implementation of the DBSCAN algorithm and helper methods.

We did both the work equally resulting in a 50:50, and spent time revising the report as well as finalising the report together as a team.