Data Mining - Handin 1 - Clustering

Welcome to the handin on clustering algorithms and outlier detection. This handin corresponds to the topics in Week 5--9 in the course.

The handin is

- done in the chosen handin groups
- worth 10% of the final grade

For the handin, you will prepare a report in PDF format, by exporting the Jupyter notebook. Please submit

1. The jupyter notebook file with your answers
2. The PDF obtained by exporting the jupyter notebook

Submit both files on Brightspace no later than **March 10th kl. 23:59**.

The grading system: Tasks are assigned a number of points based on the difficulty and time to solve it. The sum of the number of points is 100. For the maximum grade you need to get at least *80 points*. The minimum grade (02 in the Danish scale) requires **at least** 30 points, with at least 8 points on of the first three Parts (Part 1,2,3) and 6 points in the last part (Part 4).

The exercise types: There are three different types of exercises

1. **[Compute by hand]** means that you should provide NO code, but show the main steps to reach the result (not all).
2. **[Motivate]** means to provide a short answer of 1-2 lines indicating the main reasoning, e.g., the PageRank of a complete graph is 1/n in all nodes as all nodes are symmetric and are connected one another.
3. **[Describe]** means to provide a potentially longer answer of 1-5 lines indicating the analysis of the data and the results.
4. **[Prove]** means to provide a formal argument and NO code.
5. **[Implement]** means to provide an implementation. Unless otherwise specified, you are allowed to use helper functions (e.g., `np.mean`, `itertools.combinations`, and so on). However, if the task is to implement an algorithm, by no means a call to a library that implements the same algorithm will be deemed as sufficient!

!!! IMPORTANT: YOU ARE NOT ALLOWED TO USE LIBRARY FUNCTIONS (SCIPY, NUMPY etc.) UNLESS EXPLICITY MENTIONED !!!

Good luck!

In [ ]:
```
import sys
#!conda install --yes --prefix {sys.prefix} seaborn
```

In [ ]:
```
## DO NOT TOUCH
import numpy as np
import pandas as pd
import warnings
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
from sklearn.cluster import KMeans, DBSCAN, OPTICS
from sklearn.preprocessing import StandardScaler, KBinsDiscretizer
import time
import seaborn as sns

RANDOM_SEED = 132414
## DO NOT TOUCH
warnings.filterwarnings('ignore')
%matplotlib inline
import matplotlib.pyplot as plt

wq = pd.read_csv("./data/winequality-red.csv", sep=';')
toy = wq[wq['quality'].isin([4, 8])].sample(n=20, random_state=RANDOM_SEED)
```

Intro Excercises

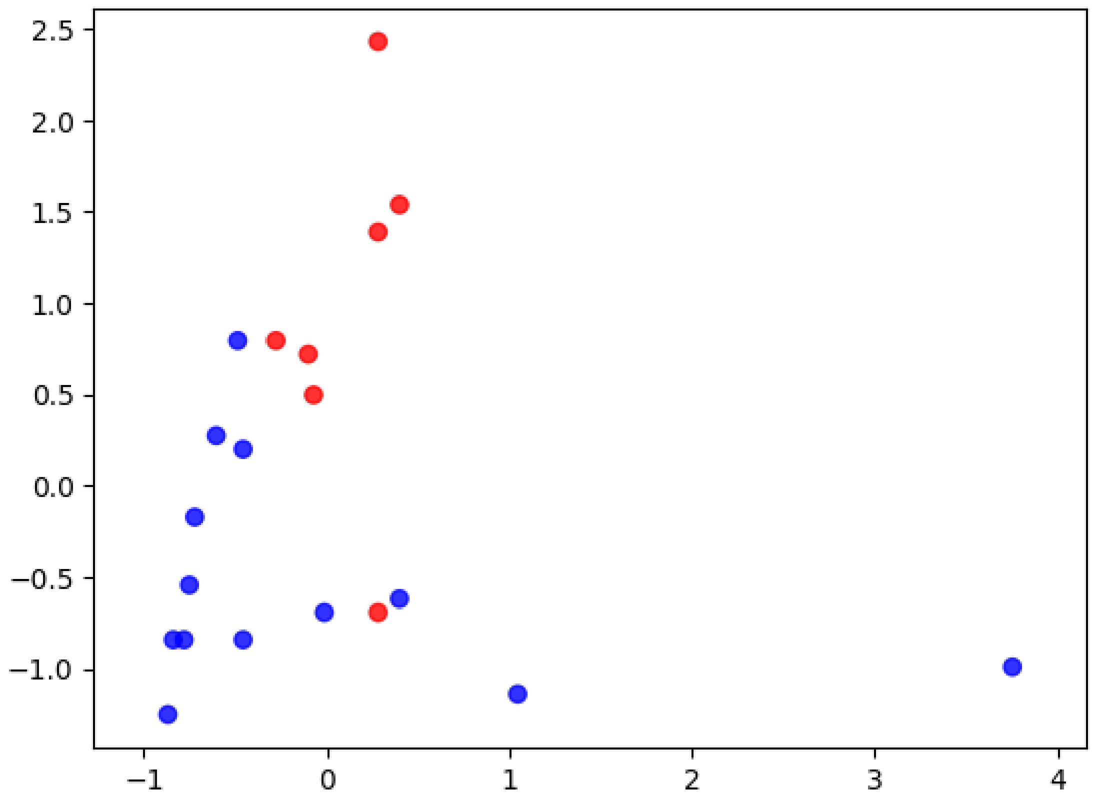
Task 1.1 K-Means and DBScan

Task 1.1.1 (5 points)

[Compute by hand] the cluster assignments *for the dataset below* using k-means and $k = 2$, with initial centroids being (0, 0) and (1,1)

To evaluate (i.e., only to control the correctness and not to solve the exercise) your results you can use **sklearn.cluster.KMeans**.

In [ ]:
```
color_map = {4:'Blue', 8:'Red'}
X_kmeans = toy[["sulphates", "alcohol"]]

scaler = StandardScaler().fit(X_kmeans)
X_scaled = scaler.transform(X_kmeans)

plt.scatter(X_scaled[:, 0], X_scaled[:, 1], alpha=0.8, c=toy['quality'].map(color_r
plt.axis('equal');
```



Answer

We have that, when assigning points to a cluster, it is assigned as follows:

$$C_i = \{x \in X : \text{\argmin}_i ||x - \mu_i||^2\}$$

meaning that, for each point $x \in X$, we compute the distance to each centroid μ_i and assign it to the cluster with the closest centroid. Thus forming the clusters C_i, for i assigning all the different clusters.

So, for the point in the lower left corner, we have that its coordinates are $p \approx (-0.869, -1.252)$. We compute the distance to each centroid, and we have that:

$$||p - \mu_1||^2 = \sqrt{(-0.869 - 0)^2 + (-1.252 - 0)^2}^2 = (-0.869)^2 + (-1.252)^2 = 2.323$$

$$||p - \mu_2||^2 = \sqrt{(-0.869 - 1)^2 + (-1.252 - 1)^2}^2 = (-1.869)^2 + (-2.252)^2 = 8.565$$

So, we assign the point to the cluster with the closest centroid, which is the cluster with centroid $\mu_1 = (0, 0)$, and we have that $C_1 = \{p\}$.

Then we go through all the other points, to assign them to a cluster. We have that:

$$C_1 = [(-0.751, -0.545), (-0.456, 0.199), (-0.015, -0.694), (-0.456, -0.843), (-0.839, -0.843), (-0.604, 0.273), (-0.103, 0.720), (0.280, -0.694), (-0.721, -0.173), (0.397, -0.619), (-0.780, -0.843), (-0.486, 0.794), (-0.074, 0.497), (-0.280, 0.794), (1.045, -1.140), (-0.869, -1.252)]$$
$$C_2 = [(0.280, 1.389), (0.280, 2.431), (3.754, -0.991), (0.397, 1.538)]$$

In [ ]:

```python
# Setting the initial centroids
mu1, mu2 = [0, 0], [1, 1]

def clustering_k2(X, mu1, mu2):

    # Initializing the clusters
    C1, C2 = [], []

    # Set an empty list to store the labels
    labels = []

    # Loop over all the points in X_scaled using the method described above to assi
    for p in X:
        # Distance from each centroid
        d1 = (p[0] - mu1[0])**2 + (p[1] - mu1[1])**2
        d2 = (p[0] - mu2[0])**2 + (p[1] - mu2[1])**2

        # Assigning the point to the closest centroid
        if d1 < d2:
            C1.append(p)
            labels.append(0)
        else:
            C2.append(p)
            labels.append(1)


    print(f'The points falling in cluster C_1 are:\n {C1}\n')
    print(f'The points falling in cluster C_2 are:\n {C2}\n')
    print(f'We get the following labels:\n {labels}\n in the same order as the poi

clustering_k2(X_scaled, mu1, mu2)
```

```
The points falling in cluster C_1 are:
 [array([-0.75082858, -0.54497977]), array([-0.456386  ,  0.19901992]), array([-0.
01472213, -0.69377971]), array([-0.456386  , -0.84257965]), array([-0.83916135, -
0.84257965]), array([-0.60360729,  0.27341989]), array([-0.1030549,  0.7198197]),
array([ 0.27972045, -0.69377971]), array([-0.72138432, -0.17297993]), array([ 0.39
749748, -0.61937974]), array([-0.78027283, -0.84257965]), array([-0.48583026,  0.7
9421967]), array([-0.07361064,  0.49661979]), array([-0.27972045,  0.79421967]), a
rray([ 1.04527116, -1.14017952]), array([-0.86860561, -1.25177947])]

The points falling in cluster C_2 are:
 [array([0.27972045, 1.38941942]), array([0.27972045, 2.43101898]), array([ 3.7541
4288, -0.99137958]), array([0.39749748, 1.53821935])]

We get the following labels:
 [0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0]
 in the same order as the points in X_scaled
```

In order to evaluate the correctness of the results, we can use the sklearn.cluster.KMeans function. Which gives us the following results:

In [ ]:

```python
# Assining the initial centroids to be used in the KMeans function below
start_centroids = np.array([[0,0], [1,1]])

# Setting up the KMeans function
Centroids = KMeans(n_clusters=2, init=start_centroids, n_init=1, random_state=RAND

# Fitting the KMeans function to the data
Centroids.fit(X_scaled)
```

```
# Retrieving the labels from the KMeans function
print(f'Thus we get that the labels are:\n {Centroids.labels_}\n in the same order
```

```
Thus we get that the labels are:
 [0 0 0 0 1 0 1 0 0 0 0 1 0 0 0 0 0 1 0 0]
 in the same order as the points in X_scaled
```

Thus validating our clustering results.

Task 1.1.2 (2 point)

[Compute by hand]

A) Show two examples with two different initial cluster assignments that lead to a different result.

Answer

Two examples with two different initial cluster assignments that lead to a different result are:

initial_cluster_1 = np.array([[0, 0], [1, 1]])
initial_cluster_2 = np.array([[0, 0], [4, -1]])

To see this, we can run the same function as above, but with the second initial cluster assignment, as we have already seen the results for the first one. Thus we get that:

```
In [ ]: mu1, mu2 = [0,0], [4, -1]

        clustering_k2(X_scaled, mu1, mu2)
```

```
The points falling in cluster C_1 are:
 [array([-0.75082858, -0.54497977]), array([-0.456386   ,  0.19901992]), array([-0.
01472213, -0.69377971]), array([-0.456386  , -0.84257965]), array([0.27972045, 1.3
8941942]), array([-0.83916135, -0.84257965]), array([0.27972045, 2.43101898]), arr
ay([-0.60360729,  0.27341989]), array([-0.1030549,  0.7198197]), array([ 0.2797204
5, -0.69377971]), array([-0.72138432, -0.17297993]), array([ 0.39749748, -0.619379
74]), array([-0.78027283, -0.84257965]), array([-0.48583026,  0.79421967]), array
([-0.07361064,  0.49661979]), array([-0.27972045,  0.79421967]), array([0.3974974
8, 1.53821935]), array([ 1.04527116, -1.14017952]), array([-0.86860561, -1.2517794
7])]

The points falling in cluster C_2 are:
 [array([ 3.75414288, -0.99137958])]

We get the following labels:
 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
 in the same order as the points in X_scaled
```

Showing that, with the second initial centroid assignment, we only get one point into the second cluster, and the rest of the points are assigned to the first cluster. Thus resulting in a different clustering then the first initilization of the centroids.

[Motivate]

B) How you explain the difference between the two cluster assignments in point A)?

Answer

This difference, is caused by the fact, that our second initialization of the centroids, had the second cluster right next to the one outlier in the dataset. This resulted in all the other point except for the outlier to be assigned to the first cluster. Thus resulting in a different clustering then the first initilization of the centroids.

Task 1.1.3 (5 points)

[Compute by hand] the dendrogram for the dataset of Task 1.1.1. using **average-link**.

Answer

Initially, each point is in its own cluster, so we start by having 20 individual clusters. We compute the distance between each cluster using average link $D(C_1, C_2) = \frac{1}{|C_1||C_2|} \sum_{x \in C_1} \sum_{y \in C_2} dist(x, y)$

We have that the distance between the first and second cluster (first point to second point) C0 = [-0.75082 -0.54497], C1 = [-0.456, 0.199]: is

$$D(C_0, C_1) = \frac{1}{1 \cdot 1} \cdot \sqrt{(-0.75082 - (-0.456))^2 + (-0.54497 - 0.199)^2} = 0.800144$$

first point C0 =[-0.75082 -0.54497] to third point C2 =[-0.01472213 -0.69377971]

$$D(C_0, C_2) = \frac{1}{1 \cdot 1} \cdot \sqrt{(-0.75082 - (-0.01472213))^2 + (-0.54497 - (-0.69377971))^2} =$$

Till we get a 20x20 distance matrix between each cluster. We then compute the minimum distance in this matrix to see which distance is the smallest, and we have that the minimum distance is between cluster 5 and cluster 13, with a distance of 0.05888851. We then merge the two clusters. Such that we have 19 clusters where cluster one cluster is C(5, 13).

We compute the distance between each cluster again using average link for example between the cluster a calculation for the two point cluster C(5,13) [-0.83916135 -0.84257965] [-0.78027283 -0.84257965] and C0, we have that:

$$D(C(5, 13), C0) = \frac{1}{2 \cdot 1} \cdot (\sqrt{(-0.75082 - (-0.8391))^2 + (-0.54497 - (-0.84257))^2} +$$

$$= 0.30474273$$

and between cluster C(5,13) and C1, we have that:

$$D(C(5, 13), C1) = \frac{1}{2 \cdot 1} \cdot (\sqrt{(-0.456 - (-0.8391))^2 + (0.199 - (-0.84257))^2} + \sqrt{(-0.$$

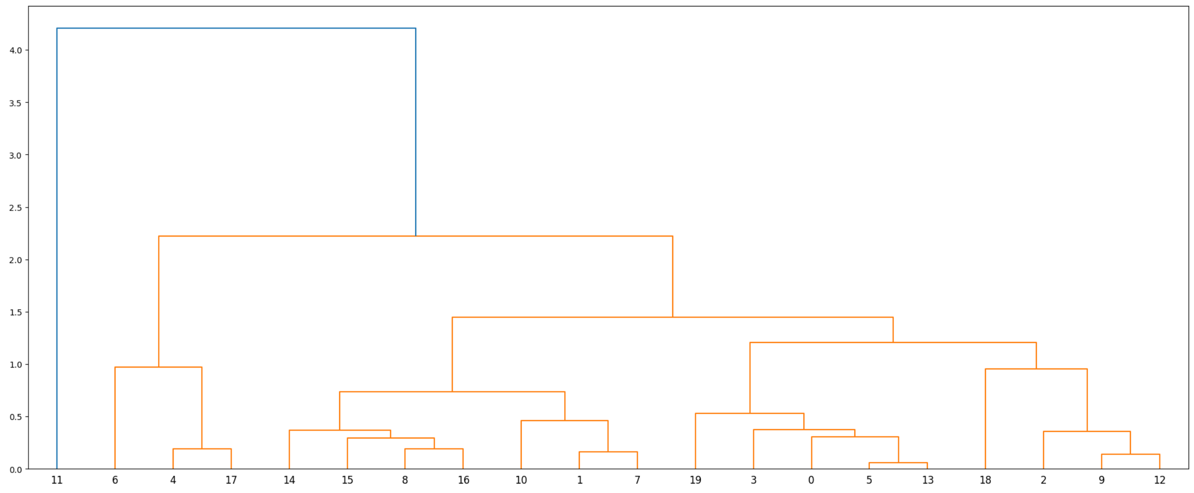
$$= 1.10025$$

We calculate all distances again and merge the two clusters with the smallest distance. We end up with the following dendrogram, where each height shows at which iteration the clusters were merged.

The next merge we make is then 9 and 12, and so on.

For visualization purposes, we can also plot the dendrogram using the scipy.cluster.hierarchy.dendrogram function. We get the following dendrogram:

In [ ]:
```
from scipy.cluster.hierarchy import dendrogram, linkage
from matplotlib import pyplot as plt

Z = linkage(X_scaled, 'average')
fig = plt.figure(figsize=(25, 10))
dn = dendrogram(Z)
plt.show()
```



In [ ]:
```
def averagelink(C1,C2):
    result = 0
    for i in range(len(C1)):
        for j in range(len(C2)):
            result += np.sqrt(np.sum((C1[i] - C2[j]) ** 2))

    return result/(len(C1)*len(C2))


print(len(X_scaled))

# #we delete the 5th and 13th element from the list
# X_scaled = np.delete(X_scaled, [5,13], axis=0)

# #we add the 5th and 13th element to the list
# new = list(X_scaled) + [C1]
# print(len(new))


C1 = [X_scaled[5], X_scaled[13]]
C2 = [X_scaled[1]]


print(averagelink(C1,C2))
```

```
dist_matrix = np.zeros((n,n))

for i in range(n):
    for j in range(n):
        if i == j:
            dist_matrix[i,j] = np.inf
        else:
            dist_matrix[i,j] = averagelink([X_scaled[i]],[X_scaled[j]])
            dist_matrix[j,i] = dist_matrix[i,j]


#print(np.where(dist_matrix == np.min(dist_matrix)))
# print(np.min(dist_matrix))
```

```
71
1.7561221884459437
```

In [ ]:
```
X_kmeans = toy[["sulphates", "alcohol"]]
scaler = StandardScaler().fit(X_kmeans)
X_scaled = scaler.transform(X_kmeans)

print(len(X_scaled))
```

```
71
```

Task 1.1.4 (2 points)

A) **[Compute by hand]** the density-based clustering for the dataset of Task 1.1.1 using $\epsilon = 0.35$ and $MinPts = 3$. Present at least 2 iterations of the algorithm.
IMPORTANT: For this exercise you can use the DBSCAN from sklearn ONLY TO CHECK YOUR RESULTS

In [ ]:
```
eps = 0.35
minpts = 3

X = X_scaled
```

In [ ]:
```
# TODO: Compute by hand the density-based clustering for the dataset of Task 1.1.1

# def DBScan(X, eps, minpts):

Clusters = np.zeros(len(X))
core_objects = np.zeros(len(X))
Next_Cluster = 1

for q in range(len(X)):
    # print(Clusters[o])
    if Clusters[q] == 0:
        N_e = 0 # Neighborhood wrt epsilon
        for p in range(len(X)):
            dist = (X[q][0] - X[p][0])**2 + (X[q][1] - X[p][1])**2

            if dist <= eps:
                N_e += 1

                Clusters[p] = Next_Cluster

        if N_e >= minpts:
            Clusters[q] = Next_Cluster
            Next_Cluster += 1
            core_objects[q] = 1
        else:
            Clusters[q] = -1
```

YOUR ANSWER HERE Let's start by randomly selecting a point from the dataset. We'll choose the first point: [-0.75082858, -0.54497977], which we call point 0

We now calculate the distances between this point and all other points in the dataset using the Euclidean distance formula:

d = sqrt((x2 - x1)^2 + (y2 - y1)^2)

d_0 = sqrt((-0.75082858 - (-0.75082858))^2 + (-0.54497977 -(-0.54497977)^2) = 0 d_1 = sqrt((-0.75082858 -(-0.456386))^2 + (-0.54497977 - 0.19901992)^2) = 0.8001449666737707

d2-d10 = [0.7509954222039864, 0.4186431870723032, 2.191787287971323, 0.3104325442991004, 3.1493808695420027, 0.831535991863758, 1.421030763176982, 1.0412361494023215, 0.373163299558396, 4.527034422554428, 1.1507337193253573, 0.29905292167887026, 1.3651663797841393, 1.2423983969364318, 1.419647139702712, 2.378733136316918, 1.892151419342693, 0.7165453577794116]

We now check if there are at least min_pts (3) points within the epsilon radius of 0.35 of the first point. We have that: [-0.83916135 -0.84257965], [-0.78027283 -0.84257965] is within epsilon radius of 0.35 of the first point, so we assign them to the same cluster as the point chosen and classify [-0.75082858, -0.54497977] as a core point.

We now move on to the next point in our dataset and repeat the process for each of the unvisited points in the dataset until all points have been assigned to clusters or marked as noise.

For the second point: point 1: [-0.456386, 0.19901992] we have the distances to all other points: d_0 = sqrt((-0.75082858 - (-0.456386))^2 + (-0.54497977 - 0.19901992)^2) = 0.8001449666737707 d_1 = sqrt((-0.456386 -(-0.456386))^2 + (0.19901992 - 0.19901992)^2) = 0

d2-d10 = [0.9960713547514121, 1.0415995627365955, 1.3996083998766535, 1.1097056456194665, 2.350249459105429, 0.16495291286596545, 0.6293451159653702, 1.1571274232008908, 0.4567362412714291, 4.375569041881148, 1.1827489143896828, 1.090794358413927, 0.5959276020502993, 0.4848532318824955, 0.6208650885982008, 1.5882607251489445, 2.01207090879681, 1.5082254076828203]

here only point 7: [-0.60360729, 0.27341989] is within epsilon radius, which is less than min_pts in a cluster, so we assign point 1 [-0.456386, 0.19901992] as NOISE

NOISE = [point 1: [-0.456386, 0.19901992]]

We keep repeating for the remaining points in the dataset and we get Core_points = points[0,5,9,13,16]

The remaining points are classified as NOISE:

NOISE = ponts[1,2,3,4,6,7,10,11,14,17,18,19]

We now collect all objects density-reachable from a core point and assign them to a new cluster. eg. for point 0 we can create a cluster

Cluster 0: [point 0: [-0.75082858 -0.54497977], point 5: [-0.83916135 -0.84257965], point 13: [-0.78027283 -0.84257965]]

furthermore since point 3: [-0.456386 -0.84257965] is density reachable from the core point point 13, we add it to the cluster

We keep repeating and finally have

Cluster 0: [point 0: [-0.75082858 -0.54497977], point 3: [-0.456386 -0.84257965], point 5: [-0.83916135 -0.84257965], point 13: [-0.78027283 -0.84257965]]

Cluster 1: [point 8: [-0.1030549 0.7198197], point 14: [-0.48583026 0.79421967], point 15: [-0.07361064 0.49661979], point 16: [-0.27972045 0.79421967]]

Cluster 2: [point 2: [-0.01472213 -0.69377971], point 9: [0.27972045 -0.69377971], point 12: [0.39749748 -0.61937974]]

The remaining 9 points are classified as NOISE:

NOISE = ponts[1,4,6,7,10,11,17,18,19]

In [ ]:
```python
distances = []
neighbors =[]


point = X_scaled[0]
for p in range(len(X_scaled)):
        # Distance from each centroid
        d = np.linalg.norm(point - X_scaled[p])
        distances.append(d)
        if d <= 0.35:
                neighbors.append(f" point: {[X_scaled[p],p]} with distance {d}")
for i in neighbors:
        print(i)
```

```
point: [array([-0.75082858, -0.54497977]), 0] with distance 0.0
point: [array([-0.83916135, -0.84257965]), 5] with distance 0.3104325442991004
point: [array([-0.78027283, -0.84257965]), 13] with distance 0.29905292167887026
```

B) **[Describe]** the difference between the clusters obtained with DBSCAN and those obtained with KMeans in Task 1.1.1?

YOUR ANSWER HERE K-Means is clustering algorithm where centroids are calculated given the choice of K. K-means partitions the data into K clusters, with each cluster represented by its centroid. Each data point is iteratively assigned to the closest centroid and then updates the centroids based on the new assignments. K-Means is suitable for data that is well-separated

DBSCAN, on the other hand, is a density-based clustering algorithm that clusters the data based on the density of the points. It groups together points that are close to each other and separates points that are far apart or in low-density areas.

The difference between K-means and DBSCAN is that DBScan does not require a specified number of clusters in advance, and it can handle arbitrary shapes of clusters. However, it may not perform well on data with varying density or high-dimensional data.

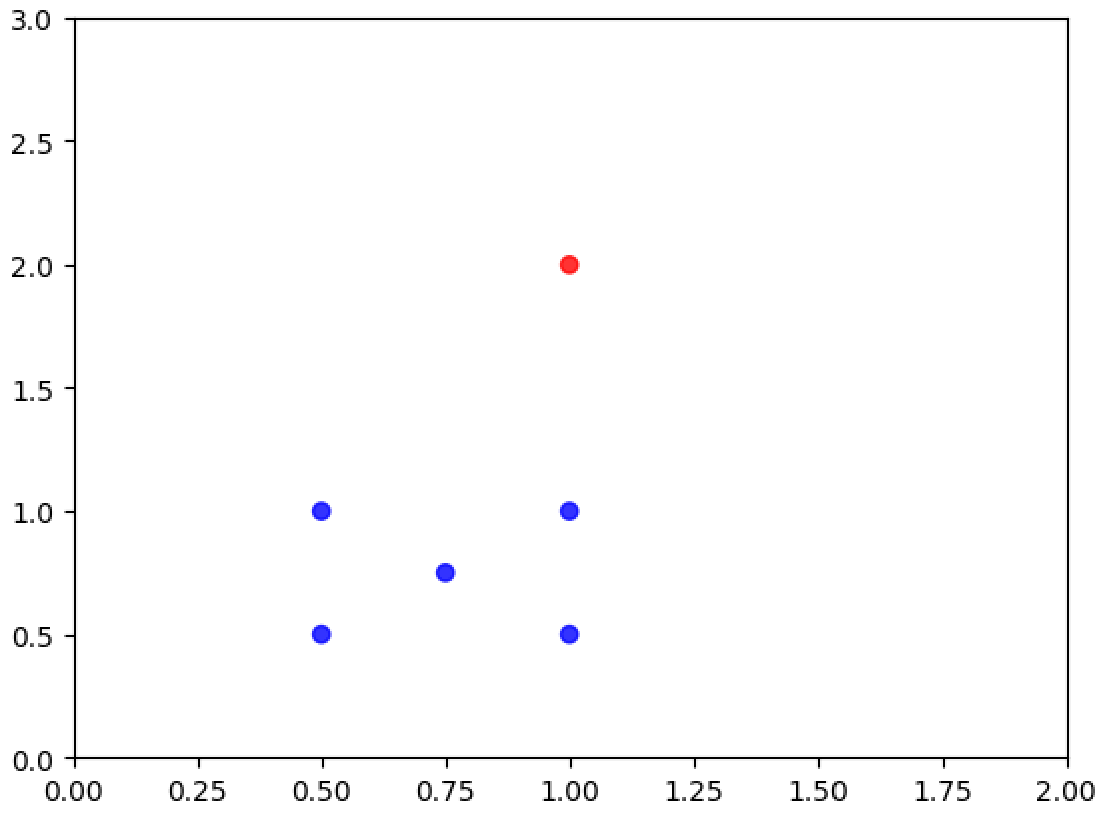
Task 1.2 Elliptic data set (2 points)

[Describe]
After looking at the dataset *below*, you want to detect the red outlier point, assuming you know that it is an outlier.

Which approach would be the most obvious to find the red outlier? Please (1) check the box and (2) motivate your answer below:

- ✓ Distance based approach (with parameteres $\pi = 0.5$, $\epsilon = 2$ and euclidean distance)
- ☐ Angle based approach
- ☐ Depth based approach

```
In [ ]: D_new = np.array([[1.0, 2.0], # Red
                         [1., 1.0],
                         [0.5, 0.5],
                         [1, 0.5],
                         [0.5, 1],
                         [0.75, 0.75]
                          ])

        plt.scatter(D_new[:, 0], D_new[:, 1], alpha=0.8, c = ['red' if i == 0 else 'blue'
        plt.axis([0, 2, 0,3])
        plt.show()
```



Answer The best approach to find the red outlier would be a distance-based approach with the parameters $\pi = 0.5$, $\epsilon = 2$. Distance-based approaches are commonly used for outlier detection, and the Euclidean distance is a good choice for measuring distances between points. The reason we set the parameters $\pi = 0.5$ and $\epsilon = 2$ is because, they would help to define a region around the majority of the points in which points that are farther away would be considered outliers.

An angle-based approach may not be as suitable in this case since the points are not arranged in a way that would make it easy to define angles between them. A depth-based approach could also be used, but it may not be as intuitive for detecting outliers in this particular dataset.

Task 1.3 Theoretical questions (4 points)

[Prove]

1. You are given a measure $d(x, y) = |x - y|$, prove that the measure is a metric
2. Prove that $\hat{\Sigma}=

\frac{1}{n}\sum_{i=1}^n (x_i -\hat{\mu}^\top)\cdot(x_i -\hat{\mu}^\top)^\top=E[(X-\hat{\mu})(X-\hat{\mu})^\top]$

Answer

- 1. If $d(\cdot, \cdot)$ is a metric, it must satisfy the following axioms for all p_i and p_j:

a) $d(p_i, p_j) \geq 0$
b) $d(p_i, p_j) = 0 \iff p_i = p_j$
c) $d(p_i, p_j) = d(p_j, p_i)$
d) $d(p_i, p_j) \leq d(p_i, p_k) + d(p_k, p_j)$ (the triangle inequality)

Thus we need to check if our metric satisfies the criteria's above.

a) $d(x, y) = |x - y| \geq 0 \quad \forall x, y$
b) $d(x, y) = |x - y| = 0 \iff x = y$
c) $d(x, y) = |x - y| = |y - x| = d(y, x)$
d) $d(x, y) = |x - y| = |x - z + z - y| \leq |x - z| + |z - y| = d(x, z) + d(z, y)$

The first two criteria's are trivially satisfied, and the last two are also trivally satisfied by the properties of the absolute value function.

- 2. Not sure if we have to prove LLN, or if we just have to show that vector multiplication can be written more compact in expectation form?

We start with the left-hand side of the equation:

$$\frac{1}{n}\sum_{i=1}^{n}(x_i - \hat{\mu})(x_i - \hat{\mu})^\top$$

We can expand this expression as follows:

$$\begin{aligned}
\frac{1}{n}\sum_{i=1}^{n}(x_i - \hat{\mu})(x_i - \hat{\mu})^\top &= \frac{1}{n}\sum_{i=1}^{n}(x_i x_i^\top - x_i\hat{\mu}^\top - \hat{\mu}x_i^\top + \hat{\mu}\hat{\mu}^\top) \\
&= \frac{1}{n}\sum_{i=1}^{n} x_i x_i^\top - \frac{1}{n}\sum_{i=1}^{n} x_i\hat{\mu}^\top - \frac{1}{n}\sum_{i=1}^{n}\hat{\mu}x_i^\top + \frac{1}{n}\sum_{i=1}^{n}\hat{\mu}\hat{\mu}^\top \\
&= \frac{1}{n}\sum_{i=1}^{n} x_i x_i^\top - 2\hat{\mu}\frac{1}{n}\sum_{i=1}^{n} x_i^\top + \frac{1}{n}\cdot n\hat{\mu}\hat{\mu}^\top \\
&= \frac{1}{n}\sum_{i=1}^{n} x_i x_i^\top - 2\hat{\mu}\hat{\mu}^\top + \hat{\mu}\hat{\mu}^\top \\
&= \frac{1}{n}\sum_{i=1}^{n} x_i x_i^\top - \hat{\mu}\hat{\mu}^\top \\
&= E[XX^\top] - \hat{\mu}\hat{\mu}^\top
\end{aligned}$$

where we used the fact that $\hat{\mu} = \frac{1}{n}\sum_{i=1}^{n} x_i$ and the linearity of expectation.

Now we consider the right-hand side of the equation:

$$E[(X - \hat{\mu})(X - \hat{\mu})^\top]$$

We can expand this expression as follows:

$$E[(X - \hat{\mu})(X - \hat{\mu})^\top] = E[XX^\top - X\hat{\mu}^\top - \hat{\mu}X^\top + \hat{\mu}\hat{\mu}^\top] \qquad = E[XX^\top] - E[X]\hat{\mu}^\top - \hat{\mu}$$

where we used the fact that $E[X] = \hat{\mu}$ and $E[X^\top] = E[X]^\top = \hat{\mu}^\top$.

Therefore, we have shown that the left-hand side of the equation is equal to the right-hand side of the equation, and thus:

$$\frac{1}{n}\sum_{i=1}^{n}(x_i-\hat{\mu})(x_i-\hat{\mu})^\top = E[(X-\hat{\mu})(X-\hat{\mu})^\top]$$

Further remark, the first equality holds per definition from the top of page 48 in [DMA] (equation 2.37) as the sample covariance matrix.

Part 2 Exploratory data analysis

In this section, you will perform preliminary analysis on your data. These preliminary analysis are useful to understand how the data behaves, before running complex algorithms.

This dataset is about red wine variants of the Portuguese "Vinho Verde" wine. It only contains physicochemical and sensory variables, so no prices, grape types and such. Every sample has also a class of quality which has scores between 1 and 10. It has been used and published with Cortez et al., 2009

In [ ]:
```
toy = wq[wq['quality'].isin([4, 8])]
data_np = toy.to_numpy()
headers = ["fixed acidity","volatile acidity","citric acid","residual sugar","chlor
X = data_np[:,:10]
y = data_np[:,11]
y = y.astype(int) - 1
rows, cols = np.shape(X)
toy.head()
```

Out[ ]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol
18	7.4	0.590	0.08	4.4	0.086	6.0	29.0	0.9974	3.38	0.50	9.0
38	5.7	1.130	0.09	1.5	0.172	7.0	19.0	0.9940	3.50	0.48	9.8
41	8.8	0.610	0.30	2.8	0.088	17.0	46.0	0.9976	3.26	0.51	9.3
45	4.6	0.520	0.15	2.1	0.054	8.0	65.0	0.9934	3.90	0.56	13.1
73	8.3	0.675	0.26	2.1	0.084	11.0	43.0	0.9976	3.31	0.53	9.2

Task 2.1 Correlation matrix

Task 2.1.1 (5 points)

A) **[Implement]** in the code-box below the **correlation matrix** (not covariance matrix) among all the attributes.
To CHECK your results you can use **numpy.corrcoef**.

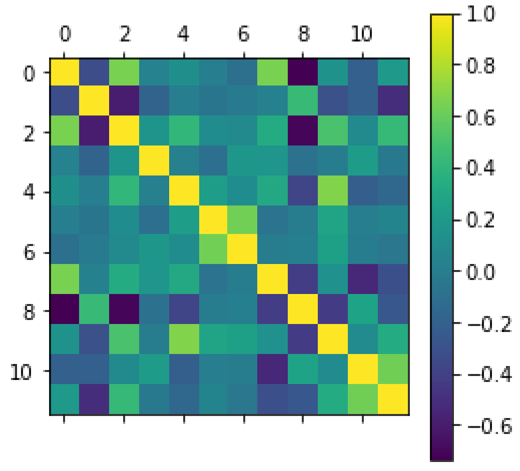
In [ ]:
```
X
```

```
Out[ ]:  array([[7.40000e+00, 5.90000e-01, 8.00000e-02, 4.40000e+00, 8.60000e-02,
                 6.00000e+00, 2.90000e+01, 9.97400e-01, 3.38000e+00, 5.00000e-01],
                [5.70000e+00, 1.13000e+00, 9.00000e-02, 1.50000e+00, 1.72000e-01,
                 7.00000e+00, 1.90000e+01, 9.94000e-01, 3.50000e+00, 4.80000e-01],
                [8.80000e+00, 6.10000e-01, 3.00000e-01, 2.80000e+00, 8.80000e-02,
                 1.70000e+01, 4.60000e+01, 9.97600e-01, 3.26000e+00, 5.10000e-01],
                [4.60000e+00, 5.20000e-01, 1.50000e-01, 2.10000e+00, 5.40000e-02,
                 8.00000e+00, 6.50000e+01, 9.93400e-01, 3.90000e+00, 5.60000e-01],
                [8.30000e+00, 6.75000e-01, 2.60000e-01, 2.10000e+00, 8.40000e-02,
                 1.10000e+01, 4.30000e+01, 9.97600e-01, 3.31000e+00, 5.30000e-01],
                [8.30000e+00, 6.25000e-01, 2.00000e-01, 1.50000e+00, 8.00000e-02,
                 2.70000e+01, 1.19000e+02, 9.97200e-01, 3.16000e+00, 1.12000e+00],
                [5.00000e+00, 1.02000e+00, 4.00000e-02, 1.40000e+00, 4.50000e-02,
                 4.10000e+01, 8.50000e+01, 9.93800e-01, 3.75000e+00, 4.80000e-01],
                [9.20000e+00, 5.20000e-01, 1.00000e+00, 3.40000e+00, 6.10000e-01,
                 3.20000e+01, 6.90000e+01, 9.99600e-01, 2.74000e+00, 2.00000e+00],
                [7.60000e+00, 6.80000e-01, 2.00000e-02, 1.30000e+00, 7.20000e-02,
                 9.00000e+00, 2.00000e+01, 9.96500e-01, 3.17000e+00, 1.08000e+00],
                [7.30000e+00, 5.50000e-01, 3.00000e-02, 1.60000e+00, 7.20000e-02,
                 1.70000e+01, 4.20000e+01, 9.95600e-01, 3.37000e+00, 4.80000e-01],
                [7.90000e+00, 8.85000e-01, 3.00000e-02, 1.80000e+00, 5.80000e-02,
                 4.00000e+00, 8.00000e+00, 9.97200e-01, 3.36000e+00, 3.30000e-01],
                [6.90000e+00, 1.09000e+00, 6.00000e-02, 2.10000e+00, 6.10000e-02,
                 1.20000e+01, 3.10000e+01, 9.94800e-01, 3.51000e+00, 4.30000e-01],
                [8.40000e+00, 6.35000e-01, 3.60000e-01, 2.00000e+00, 8.90000e-02,
                 1.50000e+01, 5.50000e+01, 9.97450e-01, 3.31000e+00, 5.70000e-01],
                [7.00000e+00, 9.75000e-01, 4.00000e-02, 2.00000e+00, 8.70000e-02,
                 1.20000e+01, 6.70000e+01, 9.95650e-01, 3.35000e+00, 6.00000e-01],
                [8.10000e+00, 8.70000e-01, 0.00000e+00, 3.30000e+00, 9.60000e-02,
                 2.60000e+01, 6.10000e+01, 1.00025e+00, 3.60000e+00, 7.20000e-01],
                [7.90000e+00, 3.50000e-01, 4.60000e-01, 3.60000e+00, 7.80000e-02,
                 1.50000e+01, 3.70000e+01, 9.97300e-01, 3.35000e+00, 8.60000e-01],
                [1.03000e+01, 3.20000e-01, 4.50000e-01, 6.40000e+00, 7.30000e-02,
                 5.00000e+00, 1.30000e+01, 9.97600e-01, 3.23000e+00, 8.20000e-01],
                [5.60000e+00, 8.50000e-01, 5.00000e-02, 1.40000e+00, 4.50000e-02,
                 1.20000e+01, 8.80000e+01, 9.92400e-01, 3.56000e+00, 8.20000e-01],
                [1.25000e+01, 4.60000e-01, 4.90000e-01, 4.50000e+00, 7.00000e-02,
                 2.60000e+01, 4.90000e+01, 9.98100e-01, 3.05000e+00, 5.70000e-01],
                [1.26000e+01, 3.10000e-01, 7.20000e-01, 2.20000e+00, 7.20000e-02,
                 6.00000e+00, 2.90000e+01, 9.98700e-01, 2.88000e+00, 8.20000e-01],
                [1.13000e+01, 6.20000e-01, 6.70000e-01, 5.20000e+00, 8.60000e-02,
                 6.00000e+00, 1.90000e+01, 9.98800e-01, 3.22000e+00, 6.90000e-01],
                [9.40000e+00, 3.00000e-01, 5.60000e-01, 2.80000e+00, 8.00000e-02,
                 6.00000e+00, 1.70000e+01, 9.96400e-01, 3.15000e+00, 9.20000e-01],
                [1.07000e+01, 3.50000e-01, 5.30000e-01, 2.60000e+00, 7.00000e-02,
                 5.00000e+00, 1.60000e+01, 9.97200e-01, 3.15000e+00, 6.50000e-01],
                [1.07000e+01, 3.50000e-01, 5.30000e-01, 2.60000e+00, 7.00000e-02,
                 5.00000e+00, 1.60000e+01, 9.97200e-01, 3.15000e+00, 6.50000e-01],
                [1.05000e+01, 5.90000e-01, 4.90000e-01, 2.10000e+00, 7.00000e-02,
                 1.40000e+01, 4.70000e+01, 9.99100e-01, 3.30000e+00, 5.60000e-01],
                [9.90000e+00, 5.00000e-01, 2.40000e-01, 2.30000e+00, 1.03000e-01,
                 6.00000e+00, 1.40000e+01, 9.97800e-01, 3.34000e+00, 5.20000e-01],
                [5.00000e+00, 4.20000e-01, 2.40000e-01, 2.00000e+00, 6.00000e-02,
                 1.90000e+01, 5.00000e+01, 9.91700e-01, 3.72000e+00, 7.40000e-01],
                [8.20000e+00, 9.15000e-01, 2.70000e-01, 2.10000e+00, 8.80000e-02,
                 7.00000e+00, 2.30000e+01, 9.96200e-01, 3.26000e+00, 4.70000e-01],
                [1.01000e+01, 9.35000e-01, 2.20000e-01, 3.40000e+00, 1.05000e-01,
                 1.10000e+01, 8.60000e+01, 1.00100e+00, 3.43000e+00, 6.40000e-01],
                [8.30000e+00, 8.45000e-01, 1.00000e-02, 2.20000e+00, 7.00000e-02,
                 5.00000e+00, 1.40000e+01, 9.96700e-01, 3.32000e+00, 5.80000e-01],
                [7.10000e+00, 8.40000e-01, 2.00000e-02, 4.40000e+00, 9.60000e-02,
                 5.00000e+00, 1.30000e+01, 9.97000e-01, 3.41000e+00, 5.70000e-01],
                [7.50000e+00, 3.80000e-01, 4.80000e-01, 2.60000e+00, 7.30000e-02,
                 2.20000e+01, 8.40000e+01, 9.97200e-01, 3.32000e+00, 7.00000e-01],
                [9.10000e+00, 7.65000e-01, 4.00000e-02, 1.60000e+00, 7.80000e-02,
                 4.00000e+00, 1.40000e+01, 9.98000e-01, 3.29000e+00, 5.40000e-01],
                [7.50000e+00, 1.11500e+00, 1.00000e-01, 3.10000e+00, 8.60000e-02,
                 5.00000e+00, 1.20000e+01, 9.95800e-01, 3.54000e+00, 6.00000e-01],
                [6.90000e+00, 3.90000e-01, 2.40000e-01, 2.10000e+00, 1.02000e-01,
                 4.00000e+00, 7.00000e+00, 9.94620e-01, 3.44000e+00, 5.80000e-01],
                [7.80000e+00, 5.70000e-01, 9.00000e-02, 2.30000e+00, 6.50000e-02,
                 3.40000e+01, 4.50000e+01, 9.94170e-01, 3.46000e+00, 7.40000e-01],
                [7.50000e+00, 6.85000e-01, 7.00000e-02, 2.50000e+00, 5.80000e-02,
                 5.00000e+00, 9.00000e+00, 9.96320e-01, 3.38000e+00, 5.50000e-01],
                [1.16000e+01, 4.70000e-01, 4.40000e-01, 1.60000e+00, 1.47000e-01,
                 3.60000e+01, 5.10000e+01, 9.98360e-01, 3.38000e+00, 8.60000e-01],
                [7.30000e+00, 3.50000e-01, 2.40000e-01, 2.00000e+00, 6.70000e-02,
                 2.80000e+01, 4.80000e+01, 9.95760e-01, 3.43000e+00, 5.40000e-01],
                [7.10000e+00, 4.70000e-01, 0.00000e+00, 2.20000e+00, 6.70000e-02,
                 7.00000e+00, 1.40000e+01, 9.95170e-01, 3.40000e+00, 5.80000e-01],
                [8.40000e+00, 6.70000e-01, 1.90000e-01, 2.20000e+00, 9.30000e-02,
                 1.10000e+01, 7.50000e+01, 9.97360e-01, 3.20000e+00, 5.90000e-01],
                [1.20000e+01, 6.30000e-01, 5.00000e-01, 1.40000e+00, 7.10000e-02,
                 6.00000e+00, 2.60000e+01, 9.97910e-01, 3.07000e+00, 6.00000e-01],
                [9.10000e+00, 4.00000e-01, 5.00000e-01, 1.80000e+00, 7.10000e-02,
                 7.00000e+00, 1.60000e+01, 9.94620e-01, 3.21000e+00, 6.90000e-01],
                [1.00000e+01, 2.60000e-01, 5.40000e-01, 1.90000e+00, 8.30000e-02,
                 4.20000e+01, 7.40000e+01, 9.94510e-01, 2.98000e+00, 6.30000e-01],
                [7.90000e+00, 5.40000e-01, 3.40000e-01, 2.50000e+00, 7.60000e-02,
                 8.00000e+00, 1.70000e+01, 9.92350e-01, 3.20000e+00, 7.20000e-01],
                [6.50000e+00, 5.80000e-01, 0.00000e+00, 2.20000e+00, 9.60000e-02,
                 3.00000e+00, 1.30000e+01, 9.95570e-01, 3.62000e+00, 6.20000e-01],
                [6.50000e+00, 8.80000e-01, 3.00000e-02, 5.60000e+00, 7.90000e-02,
                 2.30000e+01, 4.70000e+01, 9.95720e-01, 3.58000e+00, 5.00000e-01],
                [8.80000e+00, 9.55000e-01, 5.00000e-02, 1.80000e+00, 7.50000e-02,
                 5.00000e+00, 1.90000e+01, 9.96160e-01, 3.30000e+00, 4.40000e-01],
                [8.60000e+00, 4.20000e-01, 3.90000e-01, 1.80000e+00, 6.80000e-02,
                 6.00000e+00, 1.20000e+01, 9.95160e-01, 3.35000e+00, 6.90000e-01],
                [1.02000e+01, 2.30000e-01, 3.70000e-01, 2.20000e+00, 5.70000e-02,
                 1.40000e+01, 3.60000e+01, 9.96140e-01, 3.23000e+00, 4.90000e-01],
                [6.00000e+00, 3.30000e-01, 3.20000e-01, 1.29000e+01, 5.40000e-02,
                 6.00000e+00, 1.13000e+02, 9.95720e-01, 3.30000e+00, 5.60000e-01],
                [8.10000e+00, 7.30000e-01, 0.00000e+00, 2.50000e+00, 8.10000e-02,
                 1.20000e+01, 2.40000e+01, 9.97980e-01, 3.38000e+00, 4.60000e-01],
                [6.50000e+00, 6.70000e-01, 0.00000e+00, 4.30000e+00, 5.70000e-02,
                 1.10000e+01, 2.00000e+01, 9.94880e-01, 3.45000e+00, 5.60000e-01],
                [6.30000e+00, 1.02000e+00, 0.00000e+00, 2.00000e+00, 8.30000e-02,
                 1.70000e+01, 2.40000e+01, 9.94370e-01, 3.59000e+00, 5.50000e-01],
                [8.20000e+00, 7.80000e-01, 0.00000e+00, 2.20000e+00, 8.90000e-02,
                 1.30000e+01, 2.60000e+01, 9.97800e-01, 3.37000e+00, 4.60000e-01],
                [5.50000e+00, 4.90000e-01, 3.00000e-02, 1.80000e+00, 4.40000e-02,
                 2.80000e+01, 8.70000e+01, 9.90800e-01, 3.50000e+00, 8.20000e-01],
                [8.50000e+00, 4.00000e-01, 4.00000e-01, 6.30000e+00, 5.00000e-02,
                 3.00000e+00, 1.00000e+01, 9.95660e-01, 3.28000e+00, 5.60000e-01],
                [7.50000e+00, 7.55000e-01, 0.00000e+00, 1.90000e+00, 8.40000e-02,
                 6.00000e+00, 1.20000e+01, 9.96720e-01, 3.34000e+00, 4.90000e-01],
                [6.80000e+00, 6.80000e-01, 9.00000e-02, 3.90000e+00, 6.80000e-02,
                 1.50000e+01, 2.90000e+01, 9.95240e-01, 3.41000e+00, 5.20000e-01],
                [8.00000e+00, 8.30000e-01, 2.70000e-01, 2.00000e+00, 8.00000e-02,
                 1.10000e+01, 6.30000e+01, 9.96520e-01, 3.29000e+00, 4.80000e-01],
                [6.60000e+00, 6.10000e-01, 0.00000e+00, 1.60000e+00, 6.90000e-02,
                 4.00000e+00, 8.00000e+00, 9.93960e-01, 3.33000e+00, 3.70000e-01],
                [7.20000e+00, 3.30000e-01, 3.30000e-01, 1.70000e+00, 6.10000e-02,
                 3.00000e+00, 1.30000e+01, 9.96000e-01, 3.23000e+00, 1.10000e+00],
                [6.40000e+00, 5.30000e-01, 9.00000e-02, 3.90000e+00, 1.23000e-01,
                 1.40000e+01, 3.10000e+01, 9.96800e-01, 3.50000e+00, 6.70000e-01],
                [7.20000e+00, 3.80000e-01, 3.10000e-01, 2.00000e+00, 5.60000e-02,
                 1.50000e+01, 2.90000e+01, 9.94720e-01, 3.23000e+00, 7.60000e-01],
                [6.20000e+00, 7.85000e-01, 0.00000e+00, 2.10000e+00, 6.00000e-02,
                 6.00000e+00, 1.30000e+01, 9.96640e-01, 3.59000e+00, 6.10000e-01],
                [6.70000e+00, 1.04000e+00, 8.00000e-02, 2.30000e+00, 6.70000e-02,
                 1.90000e+01, 3.20000e+01, 9.96480e-01, 3.52000e+00, 5.70000e-01],
                [5.60000e+00, 6.20000e-01, 3.00000e-02, 1.50000e+00, 8.00000e-02,
                 6.00000e+00, 1.30000e+01, 9.94980e-01, 3.66000e+00, 6.20000e-01],
                [7.20000e+00, 5.80000e-01, 5.40000e-01, 2.10000e+00, 1.14000e-01,
                 3.00000e+00, 9.00000e+00, 9.97190e-01, 3.33000e+00, 5.70000e-01],
                [6.80000e+00, 9.10000e-01, 6.00000e-02, 2.00000e+00, 6.00000e-02,
                 4.00000e+00, 1.10000e+01, 9.95920e-01, 3.53000e+00, 6.40000e-01],
                [6.90000e+00, 4.80000e-01, 2.00000e-01, 1.90000e+00, 8.20000e-02,
                 9.00000e+00, 2.30000e+01, 9.95850e-01, 3.39000e+00, 4.30000e-01],
                [7.40000e+00, 3.60000e-01, 3.00000e-01, 1.80000e+00, 7.40000e-02,
                 1.70000e+01, 2.40000e+01, 9.94190e-01, 3.24000e+00, 7.00000e-01]])
```

In [ ]:

```
def correlation_matrix(X):
    corr = None
    # YOUR CODE HERE

    data = X


# Compute the correlation matrix
    n = len(data)
    means = [sum(x) / n for x in data.T]
    stddevs = []
    for i in range(len(data[0])):
        std = [(x[i] - means[i])**2 for x in data]
        stddevs.append((sum(std) / n)**0.5)



    corr = [[0 for _ in range(len(data[0]))] for _ in range(len(data[0]))]


    for i in range(len(data[0])):
        for j in range(len(data[0])):
            if i == j:
                corr[i][j] = 1
            else:
                cov = sum((data[k][i] - means[i]) * (data[k][j] - means[j]) for k 
                corr[i][j] = cov / (stddevs[i] * stddevs[j])


    # YOUR CODE HERE
    return corr

X = data_np
Corr = correlation_matrix(X)
print(Corr)
plt.matshow(Corr)
plt.colorbar()
plt.show()
```

```
[[1, -0.32601657683847257, 0.6521302274742028, 0.03493438656618914, 0.127671224630
0788, 0.00721408143842016, -0.09335370290670614, 0.6518393555872984, -0.7348850846
057341, 0.15409619631135862, -0.20624830227625718, 0.19359054263958747], [-0.32601
657683847257, 1, -0.5964048351399217, -0.18276595476303115, 0.006695121004310826,
-0.05762256568603232, -0.012511707790229799, 0.030877132465994158, 0.4360812397422
024, -0.3030244745590644, -0.20033037584867222, -0.5048962791868324], [0.652130227
4742028, -0.5964048351399217, 1, 0.16350495835323509, 0.41314537114076105, 0.10893
33169736447, 0.09817644843151146, 0.3324457672160988, -0.7024543456464125, 0.50334
54599029221, 0.09151063635401442, 0.4306258751235469], [0.03493438656618914, -0.18
276595476303115, 0.16350495835323509, 1, 0.019655631306707767, -0.1017270517189379
8, 0.18164278438037024, 0.16776100303469146, -0.07800123824680011, 0.0010805504640
967426, 0.22429037784636513, -0.03058153816046265], [0.1276712246300788, 0.0066951
21004310826, 0.41314537114076105, 0.019655631306707767, 1, 0.22875324890174703, 0.
10687776900524952, 0.311387790025916, -0.37115019043438036, 0.6745559968200553, -
0.20769039837331688, -0.14618950059038924], [0.00721408143842016, -0.0576225656860
3232, 0.1089333169736447, -0.10172705171893798, 0.22875324890174703, 1, 0.62950266
44204577, -0.07375084312801586, 0.00020474676472789156, 0.2811419413969995, 0.0099
9211479453455, 0.04656926719626171], [-0.09335370290670614, -0.012511707790229799,
0.09817644843151146, 0.18164278438037024, 0.10687776900524952, 0.6295026644204577,
1, -0.008714019190938168, 0.011617768590369136, 0.24911573910795035, 0.00350924960
92082827, -0.045611697085719445], [0.6518393555872984, 0.030877132465994158, 0.332
4457672160988, 0.16776100303469146, 0.311387790025916, -0.07375084312801586, -0.00
8714019190938168, 1, -0.41979416914857043, 0.14347497445564528, -0.542134972586554
6, -0.3090397463833228], [-0.7348850846057341, 0.4360812397422024, -0.702454345646
4125, -0.07800123824680011, -0.37115019043438036, 0.00020474676472789156, 0.011617
768590369136, -0.41979416914857043, 1, -0.4284333278729068, 0.27277524793749824, -
0.2612334652019338], [0.15409619631135862, -0.3030244745590644, 0.503345459902922
1, 0.0010805504640967426, 0.6745559968200553, 0.2811419413969995, 0.24911573910795
035, 0.14347497445564528, -0.4284333278729068, 1, 0.10304130692486414, 0.331021210
00665546], [-0.20624830227625718, -0.20033037584867222, 0.09151063635401442, 0.224
29037784636513, -0.20769039837331688, 0.00999211479453455, 0.0035092496092082827,
-0.5421349725865546, 0.27277524793749824, 0.10304130692486414, 1, 0.62294561961833
82], [0.19359054263958747, -0.5048962791868324, 0.4306258751235469, -0.03058153816
046265, -0.14618950059038924, 0.04656926719626171, -0.045611697085719445, -0.30903
97463833228, -0.2612334652019338, 0.33102121000665546, 0.6229456196183382, 1]]
```



```
In [ ]: np.corrcoef(X, rowvar=False)
```

```
Out[ ]: array([[ 1.00000000e+00, -3.26016577e-01,  6.52130227e-01,
                 3.49343866e-02,  1.27671225e-01,  7.21408144e-03,
                -9.33537029e-02,  6.51839356e-01, -7.34885085e-01,
                 1.54096196e-01, -2.06248302e-01,  1.93590543e-01],
               [-3.26016577e-01,  1.00000000e+00, -5.96404835e-01,
                -1.82765955e-01,  6.69512100e-03, -5.76225657e-02,
                -1.25117078e-02,  3.08771325e-02,  4.36081240e-01,
                -3.03024475e-01, -2.00330376e-01, -5.04896279e-01],
               [ 6.52130227e-01, -5.96404835e-01,  1.00000000e+00,
                 1.63504958e-01,  4.13145371e-01,  1.08933317e-01,
                 9.81764484e-02,  3.32445767e-01, -7.02454346e-01,
                 5.03345460e-01,  9.15106364e-02,  4.30625875e-01],
               [ 3.49343866e-02, -1.82765955e-01,  1.63504958e-01,
                 1.00000000e+00,  1.96556313e-02, -1.01727052e-01,
                 1.81642784e-01,  1.67761003e-01, -7.80012382e-02,
                 1.08055046e-03,  2.24290378e-01, -3.05815382e-02],
               [ 1.27671225e-01,  6.69512100e-03,  4.13145371e-01,
                 1.96556313e-02,  1.00000000e+00,  2.28753249e-01,
                 1.06877769e-01,  3.11387790e-01, -3.71150190e-01,
                 6.74555997e-01, -2.07690398e-01, -1.46189501e-01],
               [ 7.21408144e-03, -5.76225657e-02,  1.08933317e-01,
                -1.01727052e-01,  2.28753249e-01,  1.00000000e+00,
                 6.29502664e-01, -7.37508431e-02,  2.04746765e-04,
                 2.81141941e-01,  9.99211479e-03,  4.65692672e-02],
               [-9.33537029e-02, -1.25117078e-02,  9.81764484e-02,
                 1.81642784e-01,  1.06877769e-01,  6.29502664e-01,
                 1.00000000e+00, -8.71401919e-03,  1.16177686e-02,
                 2.49115739e-01,  3.50924961e-03, -4.56116971e-02],
               [ 6.51839356e-01,  3.08771325e-02,  3.32445767e-01,
                 1.67761003e-01,  3.11387790e-01, -7.37508431e-02,
                -8.71401919e-03,  1.00000000e+00, -4.19794169e-01,
                 1.43474974e-01, -5.42134973e-01, -3.09039746e-01],
               [-7.34885085e-01,  4.36081240e-01, -7.02454346e-01,
                -7.80012382e-02, -3.71150190e-01,  2.04746765e-04,
                 1.16177686e-02, -4.19794169e-01,  1.00000000e+00,
                -4.28433328e-01,  2.72775248e-01, -2.61233465e-01],
               [ 1.54096196e-01, -3.03024475e-01,  5.03345460e-01,
                 1.08055046e-03,  6.74555997e-01,  2.81141941e-01,
                 2.49115739e-01,  1.43474974e-01, -4.28433328e-01,
                 1.00000000e+00,  1.03041307e-01,  3.31021210e-01],
               [-2.06248302e-01, -2.00330376e-01,  9.15106364e-02,
                 2.24290378e-01, -2.07690398e-01,  9.99211479e-03,
                 3.50924961e-03, -5.42134973e-01,  2.72775248e-01,
                 1.03041307e-01,  1.00000000e+00,  6.22945620e-01],
               [ 1.93590543e-01, -5.04896279e-01,  4.30625875e-01,
                -3.05815382e-02, -1.46189501e-01,  4.65692672e-02,
                -4.56116971e-02, -3.09039746e-01, -2.61233465e-01,
                 3.31021210e-01,  6.22945620e-01,  1.00000000e+00]])
```

[Motivate]

B) By observing the **correlation matrix** in point A), which pair of different attributes has the highest correlation?

By observing the correlation matrix, we can see that the pair of attributes with the highest positive correlation is chlorides and sulphates. The correlation coefficient between these two attributes is 0,6746, which is the highest among all the pairs of attributes in the dataset.

The pair of attributes with the highest negative correlation is fixed acidity and pH. The correlation coefficient between these two attributes is -0.7349.

In [ ]:
```
for i in range(len(Corr)):
    for j in range(len(Corr[i])):
        if Corr[i][j]==1:
            Corr[i][j]=0


print(np.where(Corr == np.amax(Corr)))
np.max(Corr)

print(Corr[5][6])
Corr[9][4]
```

```
(array([4, 9], dtype=int64), array([9, 4], dtype=int64))
0.6295026644204577
```

Out[ ]:
```
0.6745559968200553
```

[Motivate]

C) What does it mean that two attributs are highly correlated?

If two attributes are highly correlated, it means that changes in one variable are associated with changes in the other variable. In other words, when one attribute changes, there is a tendency for the other attribute to change as well. This indicates a relationship between the two attributes and they are probably influenced by similar underlying factors.

If the correlation coefficient between two variables is close to 1, it indicates a strong positive correlation, which means that an increase in one variable is associated with an increase in the other variable. On the other hand, if the correlation coefficient is close to -1, it indicates a strong negative correlation, which means that an increase in one variable is associated with a decrease in the other variable.

If the correlation coefficient is close to 0, it indicates that there is no linear relationship between the two variables.

[Motivate]

D) Based on the attributes of the data in Part 2 and your answer in C), did you expect the observation of B)?

The correlation between chlorides and sulphates does make somehow make sense since they are both type of salts which probably came from a common source. Therefore when the concentration of one of these two attributes increases, the concentration of the other attribute is also likely to increase.

The negative correlation between fixed acidity and pH can also make sense because these two attributes are related to the acidity of the wine. Fixed acidity refers to the amount of acid in the wine, while pH is a measure of the acidity level of the wine. Higher fixed acidity indicates more acid in the wine, which would result in a lower pH. Therefore, it is reasonable to expect that these two attributes would be negatively correlated.

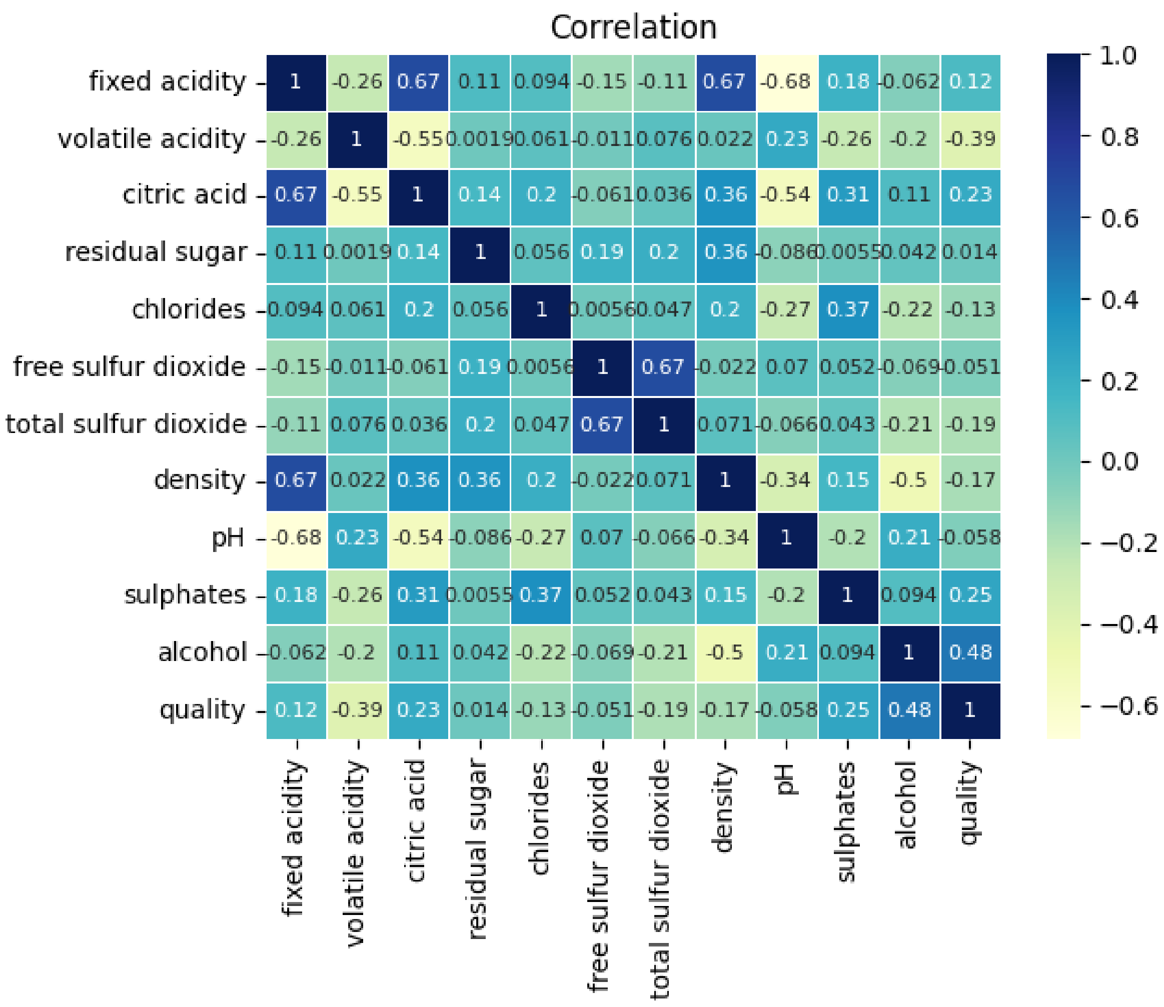
Task 2.1.2 (1 points)

[Motivate]

Plot the correlation matrix running the code below. What is the relationship between the correlation matrix and the covariance matrix? (1) Check the correct box below and (2) motivate your answer.

- [ ] The correlation matrix contains the unnormalized covariance values
- [✓] The correlation matrix contains the normalized covariance values
- [ ] The covariance matrix contains the variance of the correlation

```
In [ ]: sns.heatmap(wq.corr(),annot=True,linewidths=.5, cmap="YlGnBu", annot_kws={"fontsize
        plt.title('Correlation')
        plt.show()
```



The correlation matrix is obtained by normalizing the covariance matrix. This normalization involves dividing each element of the covariance matrix by the product of the standard deviations of the two variables corresponding to that element. The result is a matrix of correlation coefficients, where each coefficient is between -1 and 1, representing the linear relationship between the two variables.

Therefore, the correlation matrix provides a standardized way of measuring the linear relationship between variables, whereas the covariance matrix measures how much the two variables move in relation to each other. However, since the covariance value is not standardized, it can be difficult to interpret the strength of the relationship between two variables.

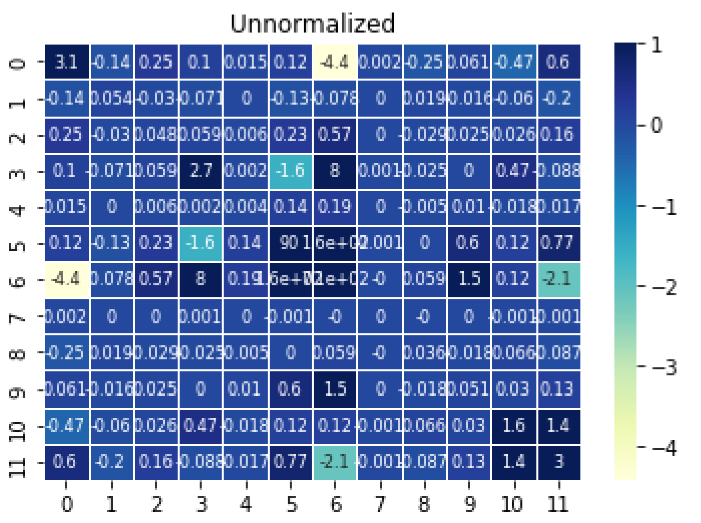
Task 2.1.3 (3 points)

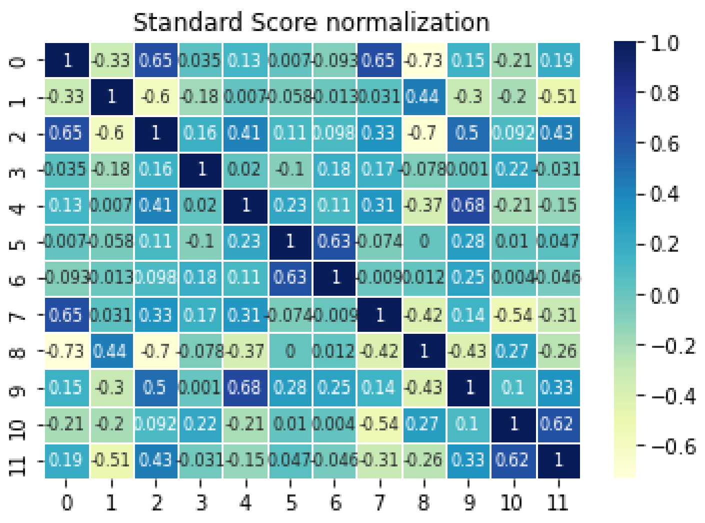
In this task, we reason about the covariance matrices.

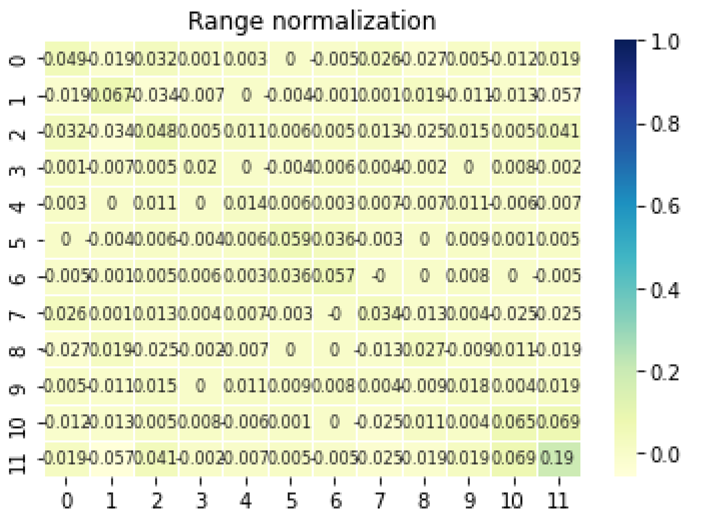
[Implement] code for normalizing the features of the wine dataset using (1) standard score normalization and (2) range normalization. Finally, (3) plot the **covariance** matrices for

1. The unnormalized data
2. The standard score normalized features
3. The range (min-max) normalized features

In [ ]:
```python
import math
# Data
X = data_np

# YOUR CODE HERE
# Standard score normalization

def standard_score(X):
    X_mean = []
    X_std = []

    X_new = [[0 for _ in range(len(X[0]))] for _ in range(len(X))]

    for i in range(len(X[0])):
        column_i = [row[i] for row in X]
        mean_i = sum(column_i) / len(column_i)
        std_i = np.sqrt(sum([(x - mean_i)**2 for x in column_i]) / len(column_i))
        X_mean.append(mean_i)
        X_std.append(std_i)
        for j in range(len(X)):
            X_new[j][i] = (X[j][i] - mean_i) / std_i

    return np.array(X_new)




# Range normalization
def range_normalization(X):
    X_min = []
    X_max = []
    X_new = [[0 for _ in range(len(X[0]))] for _ in range(len(X))]
    for i in range(len(X[0])):
        column_i = [row[i] for row in X]
        min_i = min(column_i)
        max_i = max(column_i)
        X_min.append(min_i)
        X_max.append(max_i)
        for j in range(len(X)):
            X_new[j][i] = (X[j][i] - min_i) / (max_i - min_i)
    return np.array(X_new)



def mean(X):
    return [sum(x) / n for x in X.T]



# Covariance matrices
n = len(X)
X_stand_score = standard_score(X)
X_range_norm = range_normalization(X)


mu = mean(X)
mu_ss = mean(X_stand_score)
mu_rn = mean(X_range_norm)



covariance_unnormalized = ((X - mu).T @ (X - mu)) /n
covariance_std = ((X_stand_score - mu_ss).T @ (X_stand_score - mu_ss)) /n
covariance_minmax = ((X_range_norm - mu_rn).T @ (X_range_norm - mu_rn)) /n



# Plotting covariance matrices
import matplotlib.pyplot as plt

sns.heatmap(np.round(covariance_unnormalized,3) ,annot=True,linewidths=.5, cmap="Y
plt.title('Unnormalized')
plt.show()


sns.heatmap(np.round(covariance_std,3) ,annot=True,linewidths=.5, cmap="YlGnBu", ar
plt.title('Standard Score normalization')
plt.show()


sns.heatmap(np.round(covariance_minmax,3) ,annot=True,linewidths=.5, cmap="YlGnBu",
plt.title('Range normalization')
plt.show()

# YOUR CODE HERE
```







Task 2.1.4 (3 points)

[Describe] how the covariance matrix changes with different normalization schemes and reason on why such behaviour appears. You should notice some differences. (1) Check the correct box below and (2) motivate your answer.

- ☐ Range normalization preserves the variance. Therefore, features are directly comparable.
- ☐ Standard score normalization preserves the variance. Therefore, features are directly comparable.

- [✓] Both methods normalize in such a way, that it makes sense to compare the different covariance values to each other.
- [ ] None of the methods normalize in such a way that it makes sense to compare the different covariance values to each other.

IMPORTANT: Do NOT just choose one answer. Please clarify WHY this is the correct answer.

Normalization schemes alter the scale and distribution of the data by shifting and rescaling values to a specific range. Range normalization scales the data to fit within the range [0,1] based on the difference between the maximum and minimum values, while standard score normalization transforms the data to have a mean of 0 and a standard deviation of 1.

Normalization can make it easier to compare covariances across variables with different scales. For instance, normalization can enable comparing variables measured in kilometers with variables measured in centimeters, making it easier to compare their covariances.

Task 2.2 Normal distribution

Task 2.2.1 (6 points)

Sometimes it is convenient to know whether a variable is close to a normal distribution.

[Implement] a method norm_dist that:

1. **Inputs**:
   - the number of buckets b
   - a vector x of values
2. First, compute the histogram of a Gaussian variable with mean μ corresponding to the sample mean of x and σ^2 corresponding to the sample variance of x. Second, calculate the histogram of x using b buckets.
3. **Output**: the sum of the absolute differences of the buckets between the two histograms computed in 2). The sum of the differences is computed as

$$\sum_{i=1}^{b} |H_X(i) - H_{\mathcal{N}}(i)|$$

where $H_X(i)$ is the i-th bucket of the histogram of x and $H_{\mathcal{N}}(i)$ is the i-th bucket of the hisotgram obtained from the normal distribution $\mathcal{N}(\mu, \sigma^2)$.

IMPORTANT: You can use the norm function from Scipy to get the normal distribution to subtract from.

In [ ]:
```
from scipy.stats import norm
import matplotlib.pyplot as plt

## Our data comes from the variable X
X = data_np
```

```
def norm_dist(x, b):

    # Compute sample mean and variance of the variable
    mu = np.mean(x)
    var = np.var(x)

    # Simulate normal distributed data with the same mean and variance as data obse
    x_norm = norm.rvs(loc=mu, scale=np.sqrt(var), size=len(x), random_state = RANDO

    # Compute histogram of normal distribution and sample
    hist_norm, _ = np.histogram(x_norm, bins=b)
    hist_x, _ = np.histogram(x, bins=b)

    # Plot histogram

    plt.hist(x_norm, bins=b, alpha=0.5, label='Normal distribution')
    plt.hist(x, bins=b, alpha=0.5, label='Sample')
    plt.legend(loc='upper right')
    plt.show()

    # Compute absolute differences and sum them up
    dist = np.sum(np.abs(hist_x - hist_norm))
    return dist
```

Task 2.2.2 (6 point)

A) **[Motivate]** which drawbacks the method in Task 2.2.1 has.

Answer

The method in Task 2.2.1 has the following drawbacks:

1. The method is not robust to outliers. If the data contains outliers, the histogram of the data will be very different from the histogram of the normal distribution. Therefore, the method will not be able to detect that the data is not normally distributed.

2. The method is not robust to the number of buckets. If the number of buckets is too small, the histogram of the data will be very different from the histogram of the normal distribution. Therefore, the method will not be able to detect that the data is not normally distributed.

3. The method is not robust to the shape of the distribution. If the distribution is not symmetric, the histogram of the data will be very different from the histogram of the normal distribution. Therefore, the method will not be able to detect that the data is not normally distributed.

B) **[Motivate]** whether the method in Task 2.2.1 is robust to outliers.

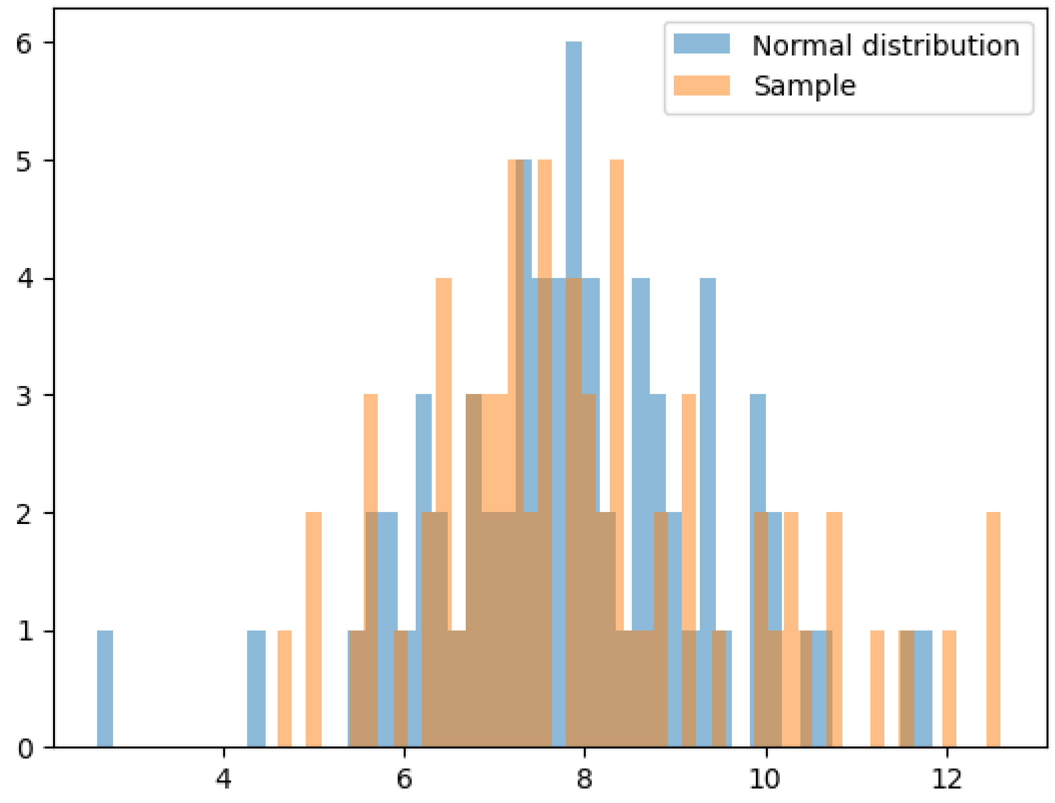
Answer

This is stated in the previous question.

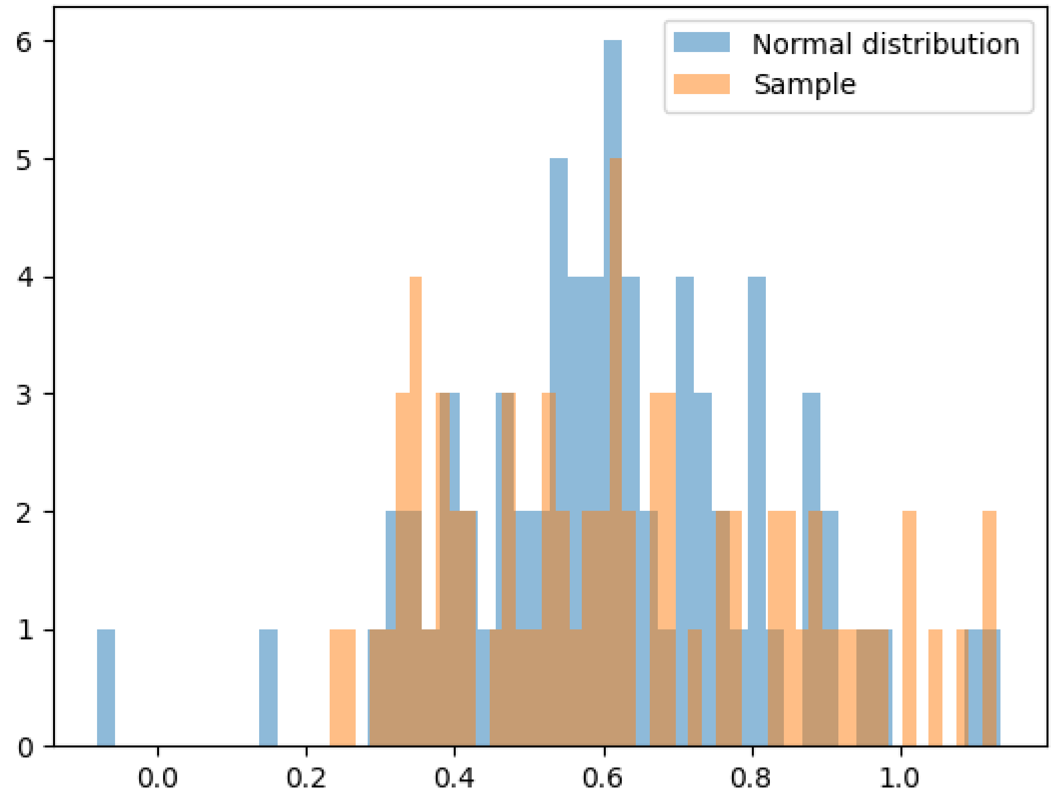
[Implement]

C) Run your code on each columns of the dataset.

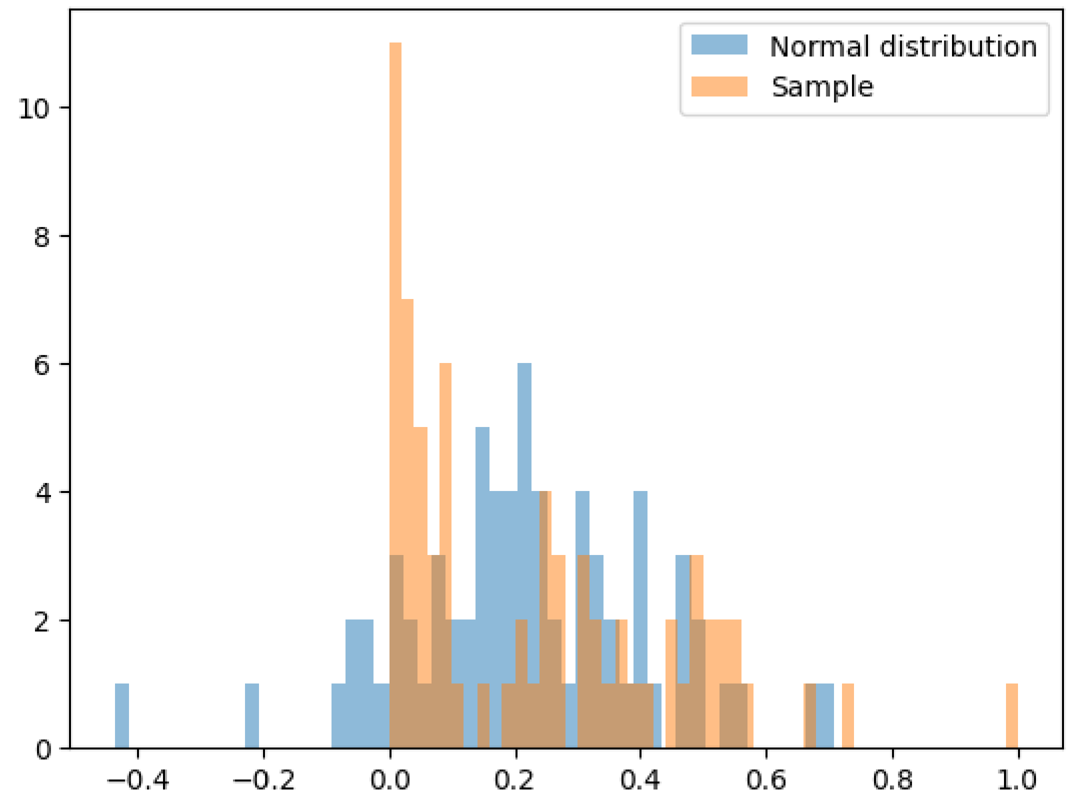
```
In [ ]: for variable in X.T:
            print(norm_dist(variable, 50))
```



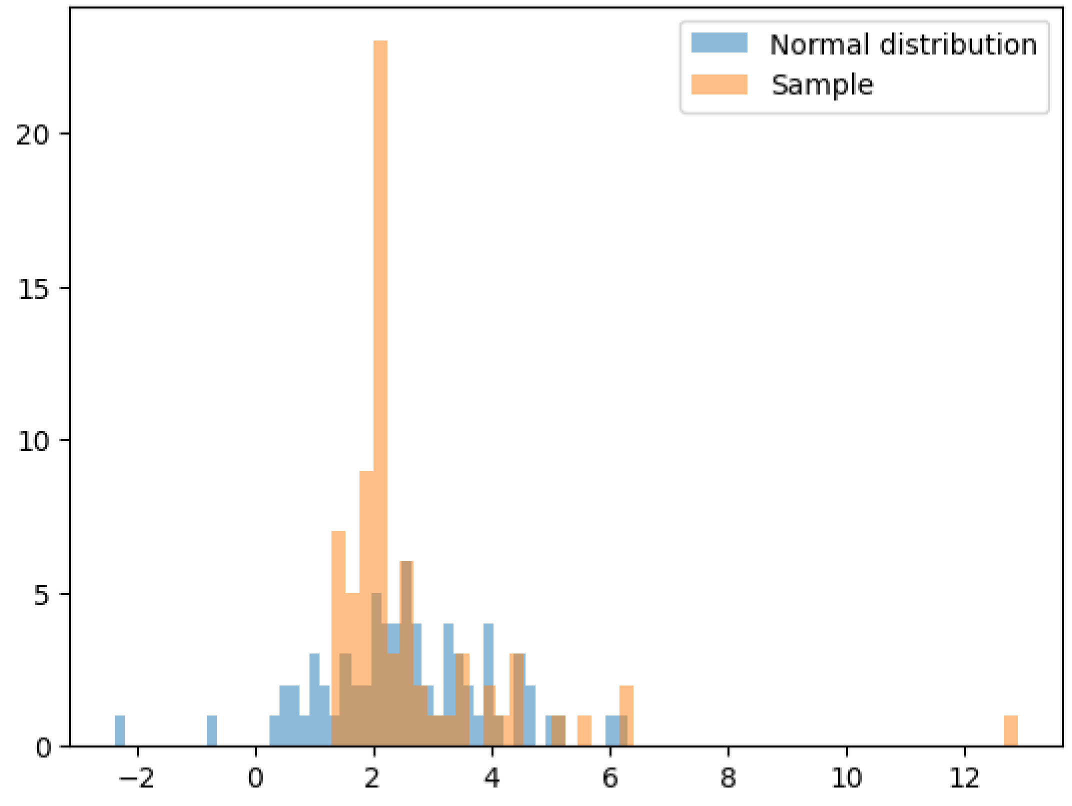
```
84
```



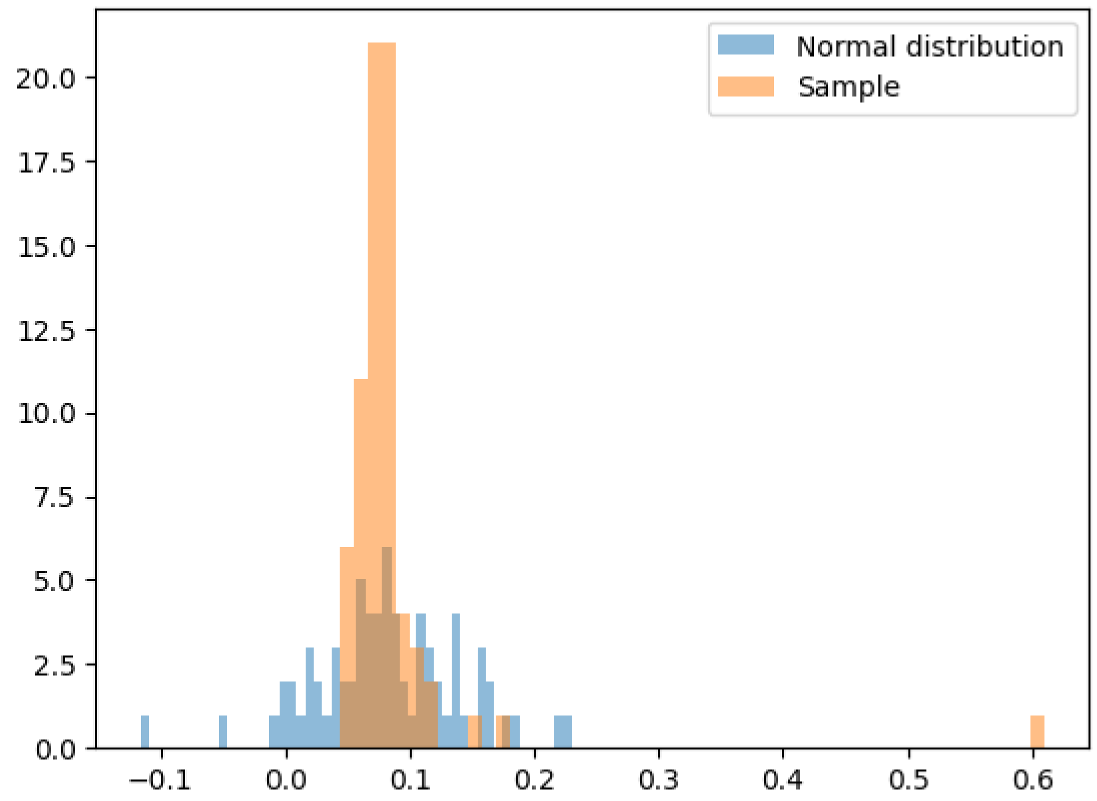
```
68
```



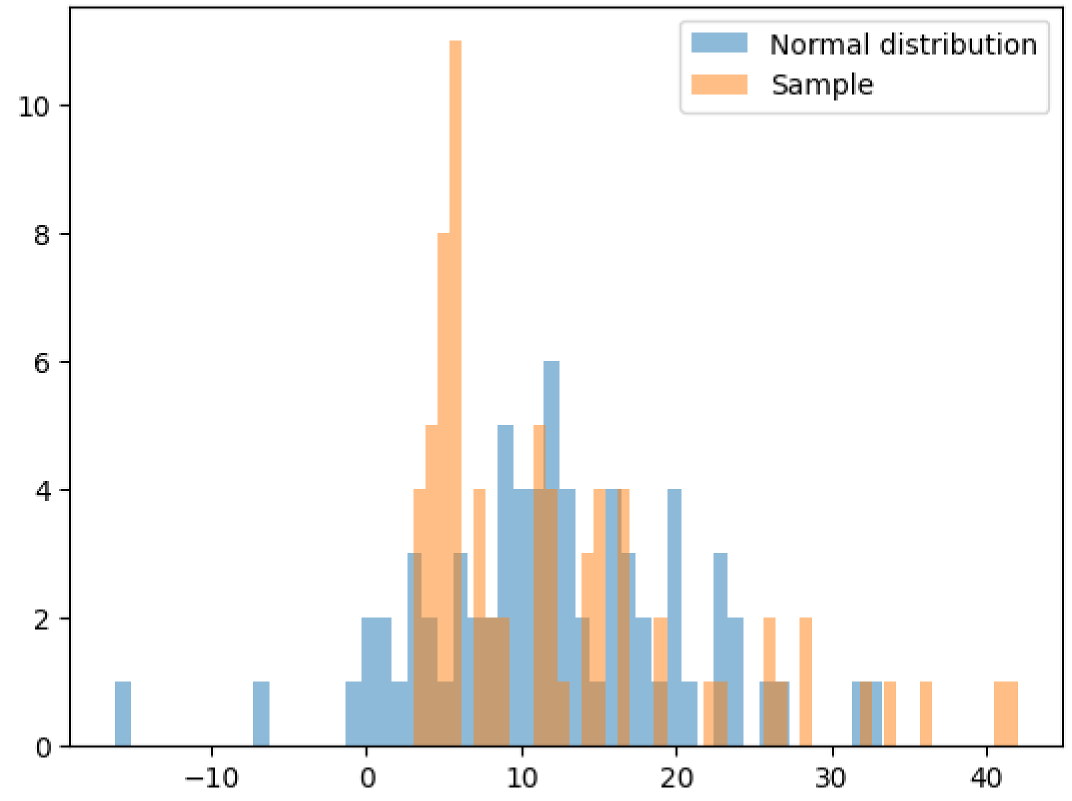
```
94
```



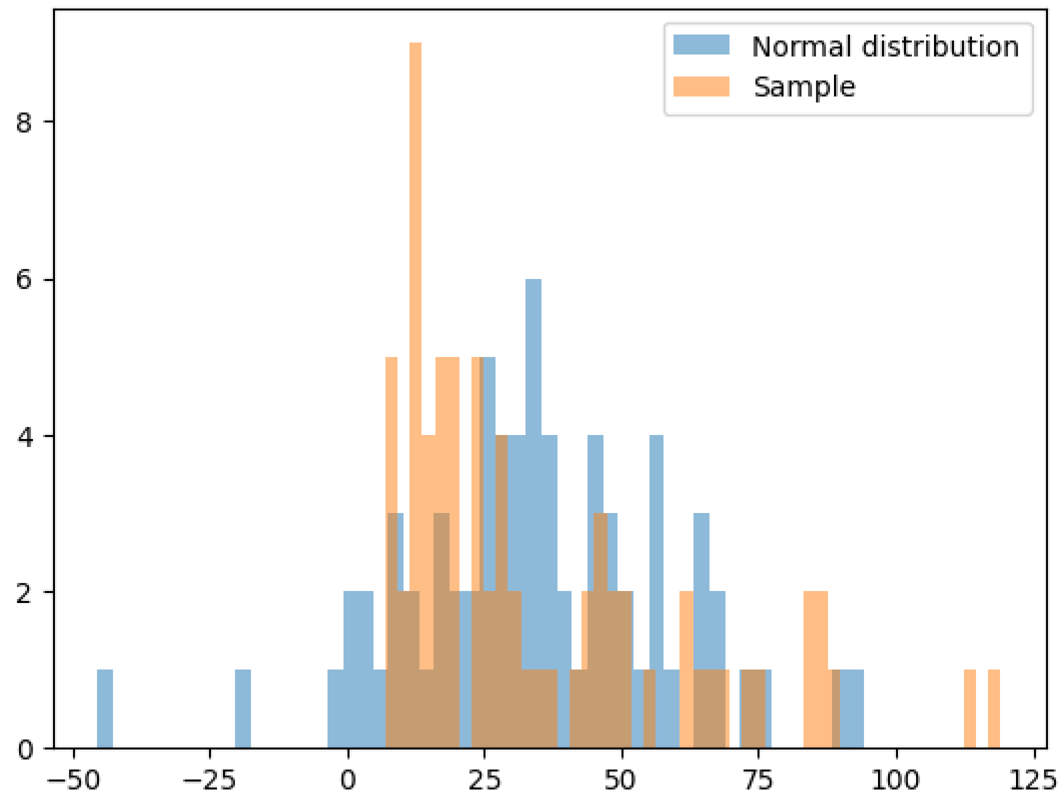
```
130
```


Normal distribution
Sample
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
−0.1
0.0
0.1
0.2
0.3
0.4
0.5
0.6

```
136
```

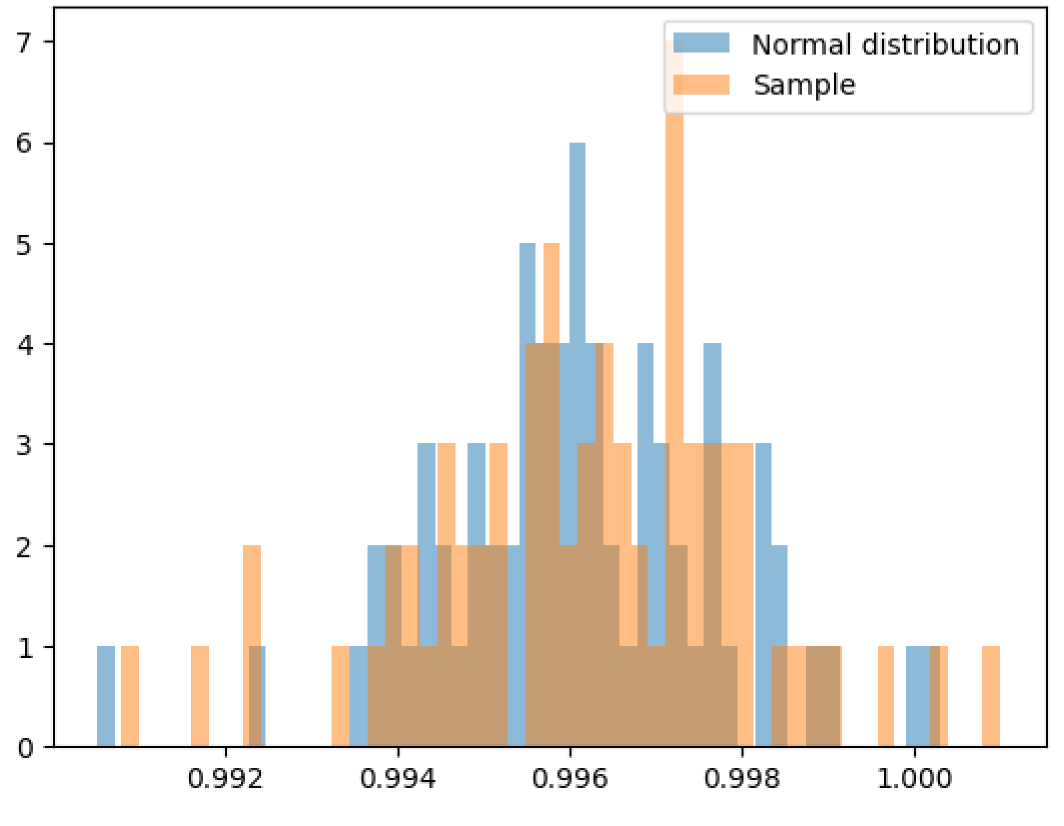

Normal distribution
Sample
10
8
6
4
2
0
−10
0
10
20
30
40

```
106
```



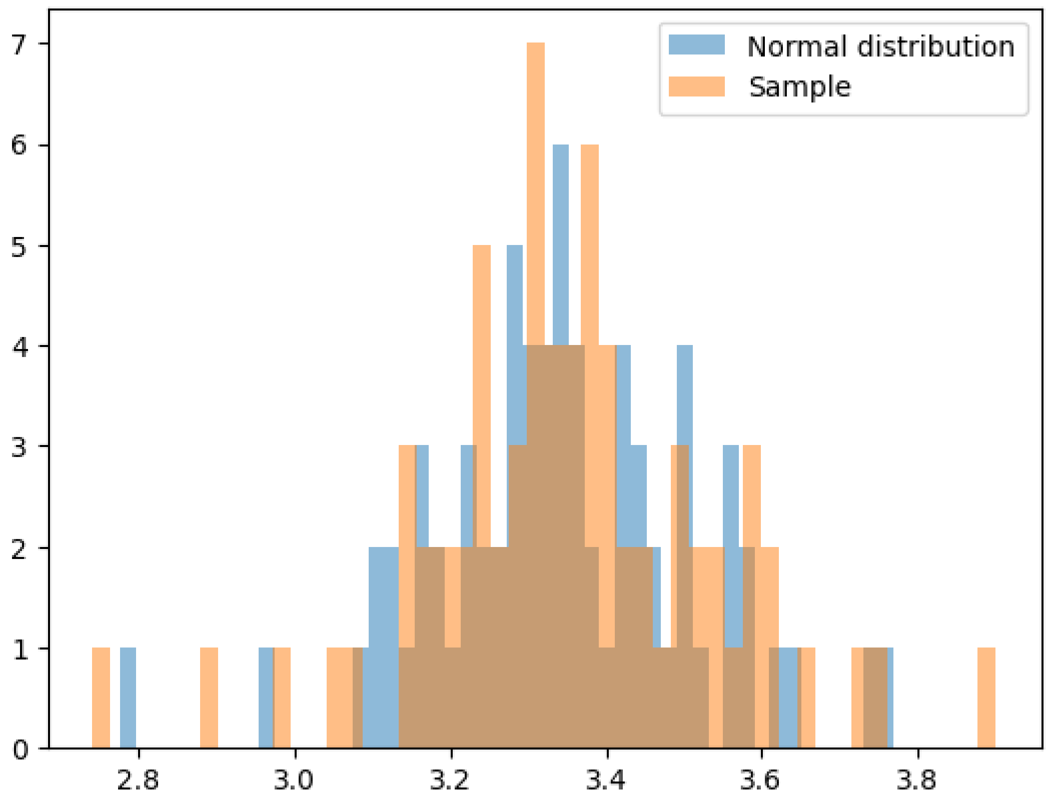
Normal distribution
Sample
8
6
4
2
0
−50
−25
0
25
50
75
100
125

96

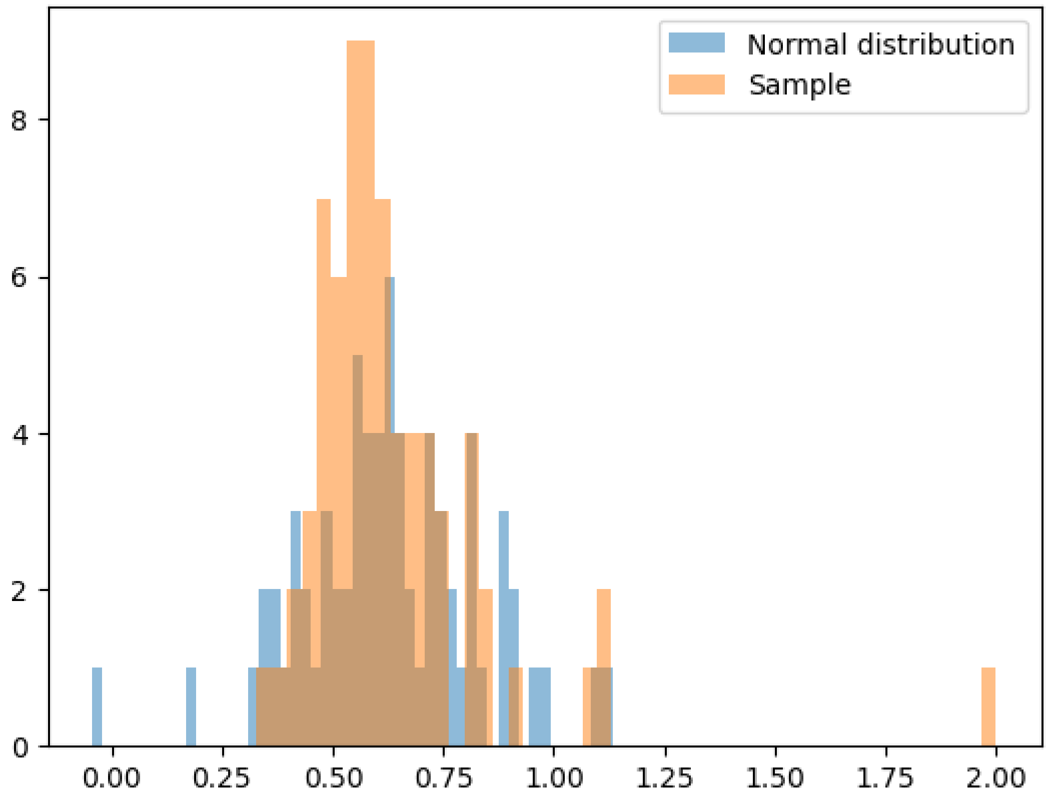


Normal distribution
Sample
7
6
5
4
3
2
1
0
0.992
0.994
0.996
0.998
1.000

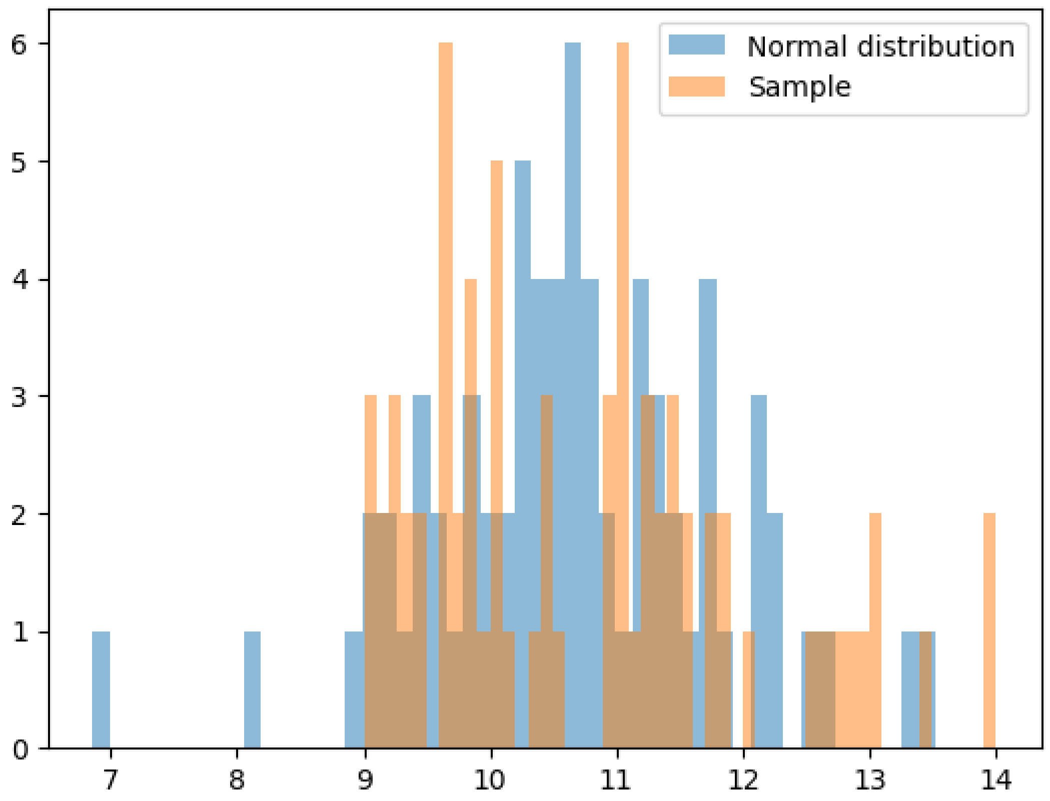
52


Normal distribution
Sample
7
6
5
4
3
2
1
0
2.8
3.0
3.2
3.4
3.6
3.8

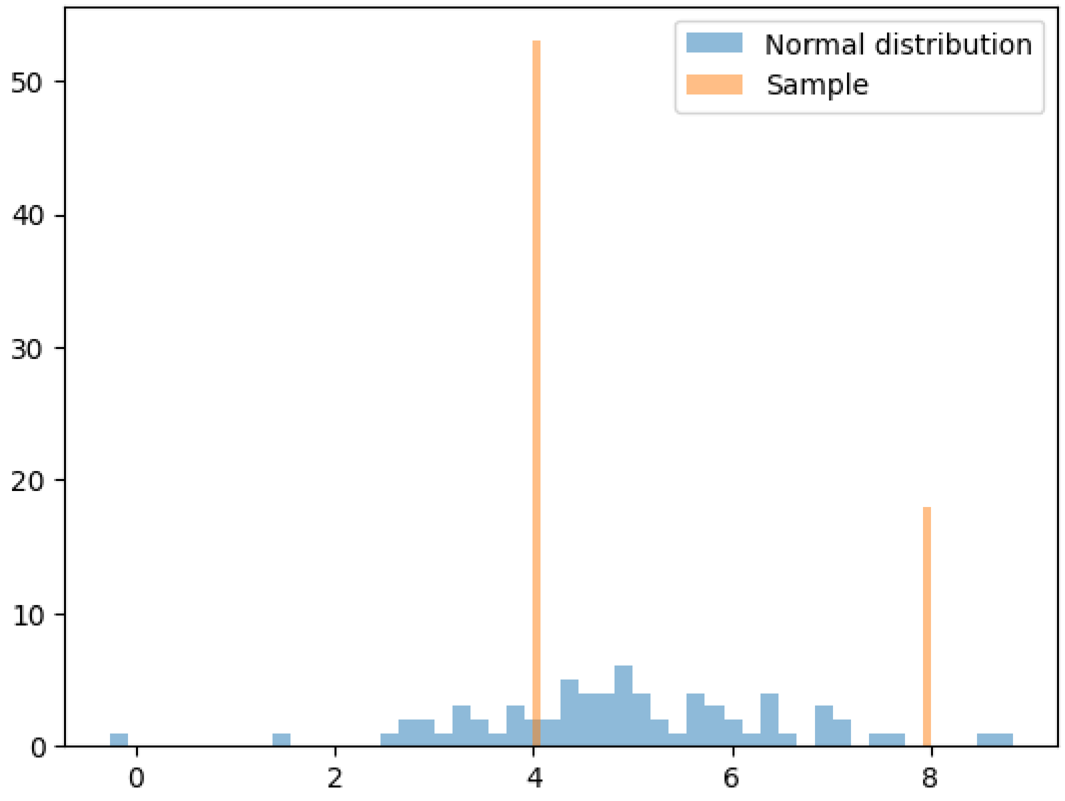
38


Normal distribution
Sample
8
6
4
2
0
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00

126



82



138

[Motivate]

D) What is the column with the largest distance?

Answer

It is the last column, which is the quality of the wine.

[Motivate]

E) Do the attribute features follow a normal distribution?

Answer

As seen from the histograms above, some of the attribute features might follow a normal distribution. While others, such as the last column, which is the quality of the wine, does not follow a normal distribution, as it only has two values, namely 4 and 8.

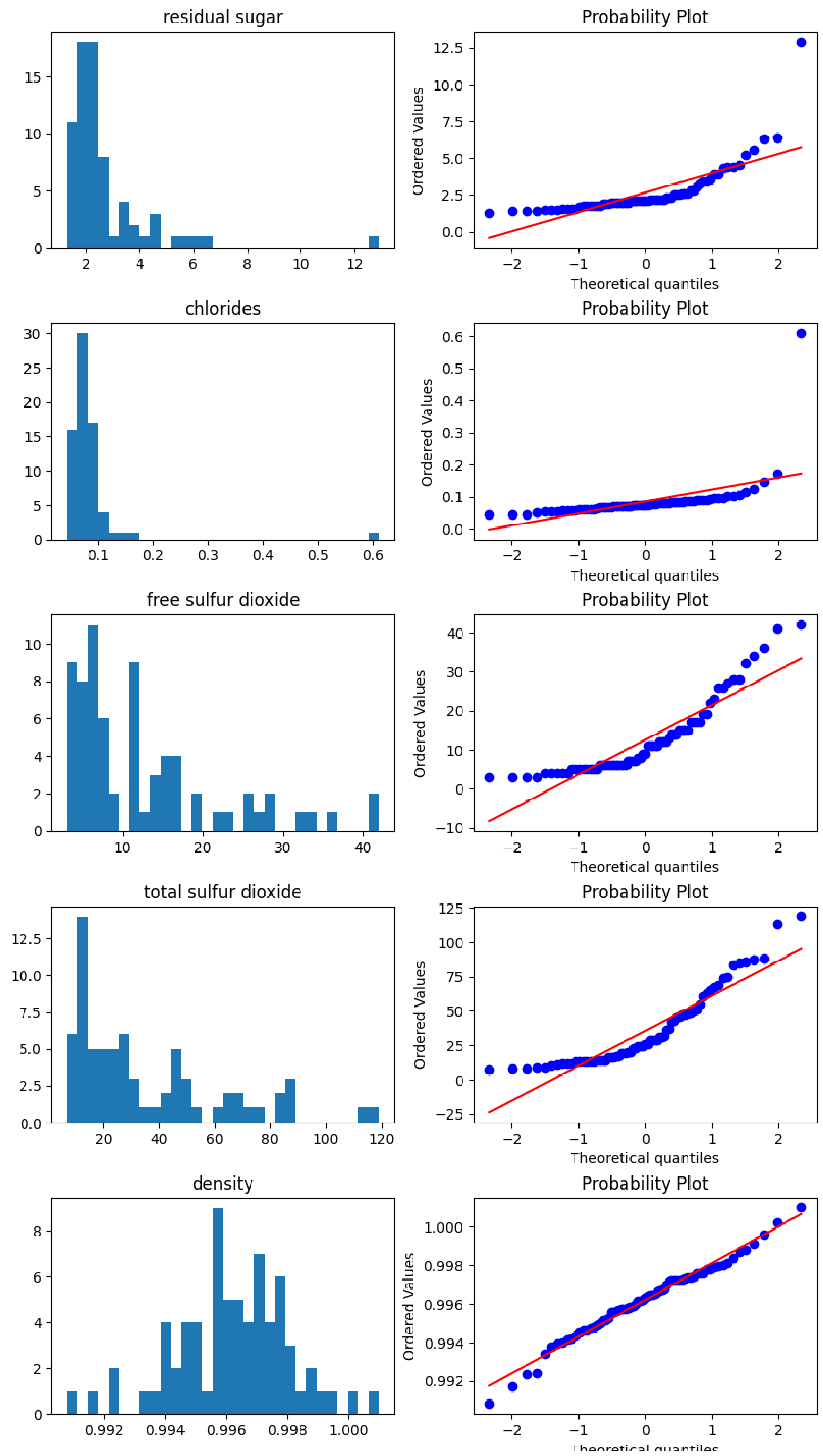
Task 2.2.3 (1 points)

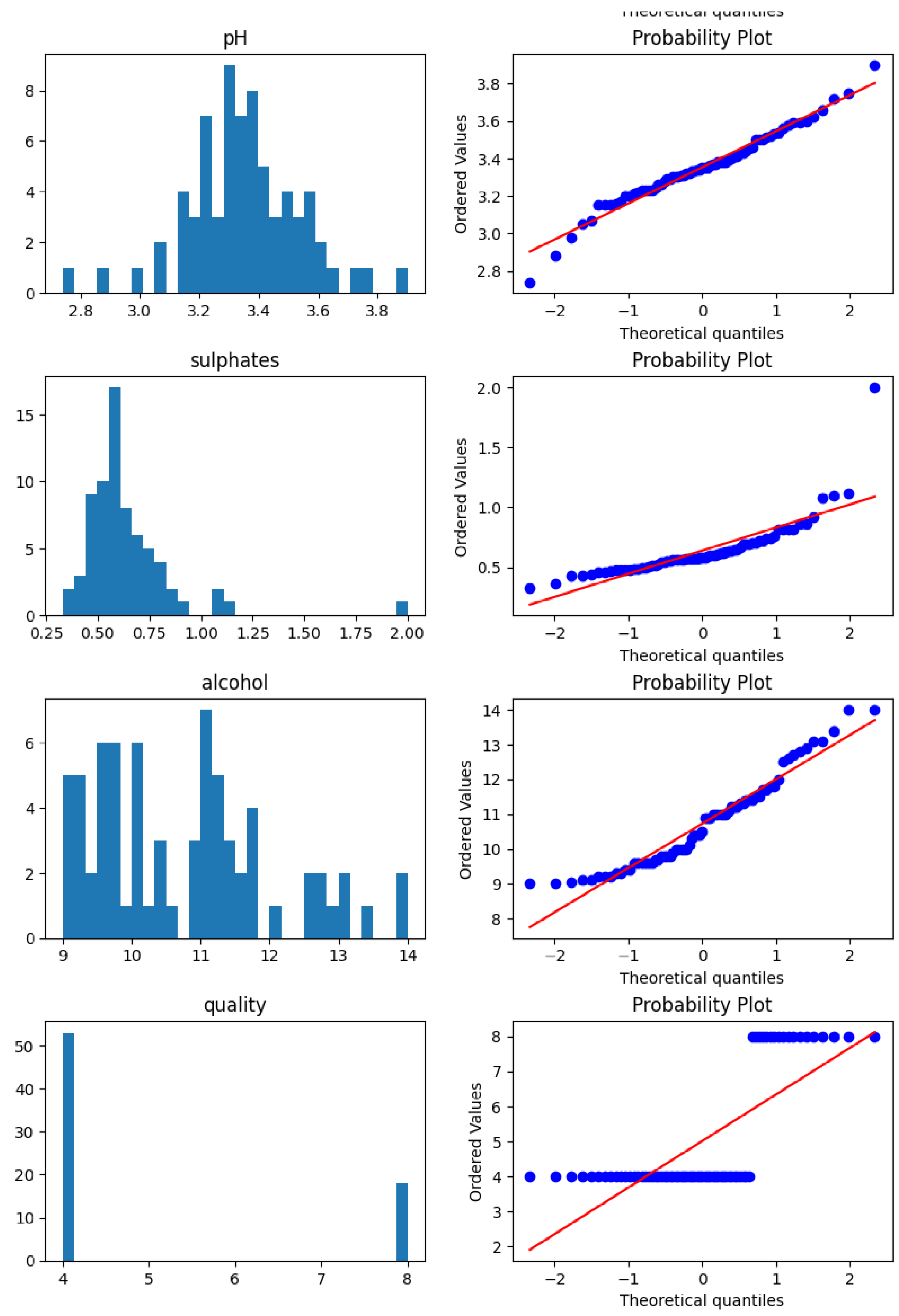
Now look at the method below. This is called a Quantile-Quantile Q-Q plot.

[Describe] why this method is more robust than the one we proposed in Task 2.2.1.

In [ ]:
```
from scipy import stats
from matplotlib import gridspec

plt.tight_layout()
_, n = X.shape

fig = plt.figure(constrained_layout=True, figsize=(8, 30))
spec = gridspec.GridSpec(ncols=2, nrows=(n-1), figure=fig)
for i in np.arange(3,n):
    x = toy[headers[i]]
    r = i-1
    qq = fig.add_subplot(spec[r, 1])
    stats.probplot(x, plot=qq)
    h = fig.add_subplot(spec[r, 0])
    h.set_title(headers[i])
    h.hist(x, bins = 30)
```

```
<Figure size 640x480 with 0 Axes>
```


residual sugar
Probability Plot
Ordered Values
Theoretical quantiles
chlorides
Probability Plot
Ordered Values
Theoretical quantiles
free sulfur dioxide
Probability Plot
Ordered Values
Theoretical quantiles
total sulfur dioxide
Probability Plot
Ordered Values
Theoretical quantiles
density
Probability Plot
Ordered Values



Answer

Using the Q-Q plot is a more robust method than using the histograms, because the Q-Q plot is not sensitive to the number of buckets. The Q-Q plot is also not sensitive to outliers, as it only compares the quantiles of the data to the quantiles of the normal distribution. The Q-Q plot is also not sensitive to the shape of the distribution, as it only compares the quantiles of the data to the quantiles of the normal distribution.

Part 3 Cluster Analysis

In this section, you will perform cluster analysis of the dataset in Part 2 and modify clustering algorithms to achieve better results.

Task 3.1

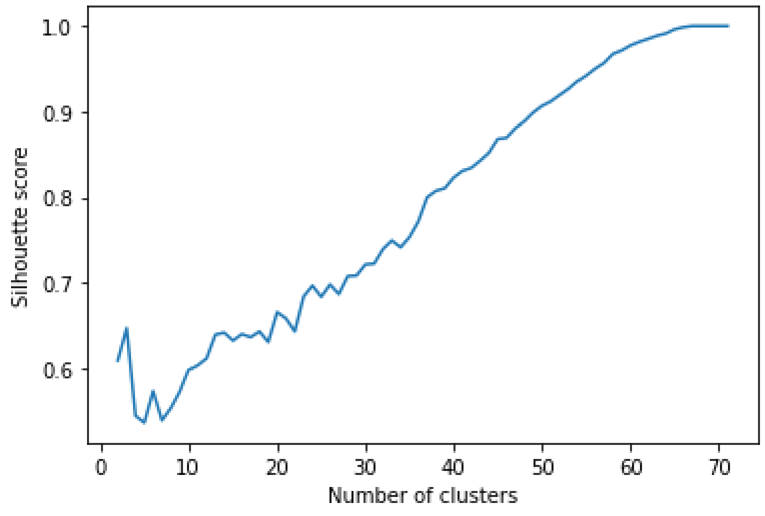
Task 3.1.1 (6 points)

A) **[Implement]** and plot the **silhouette coefficient** to detect the number of clusters k.

Use the "sulphates" and "alcohol" features of the data set.

In [ ]:
```
# Data
X = toy[["sulphates", "alcohol"]].to_numpy()

# Silhouette score

def euclidean_distance(x1, x2):
    return np.sqrt(np.sum((x1 - x2) ** 2))


def silhouette_coefficient(X, labels):
    n_samples, n_features = X.shape
    n_clusters = len(set(labels))

    # Compute the cluster centers
    cluster_centers = np.zeros((n_clusters, n_features))
    for k in range(n_clusters):
        cluster_centers[k] = np.mean(X[labels == k], axis=0)

    # Compute the average distance between each sample and all other points in the
    a = np.zeros(n_samples)
    for i in range(n_samples):
        k = labels[i]
        a[i] = np.mean([euclidean_distance(X[i], X[j]) for j in range(n_samples) i

    # Compute the average distance between each sample and all points in the neares
    b = np.zeros(n_samples)
    for i in range(n_samples):
        k = labels[i]
        b[i] = np.min([np.mean([euclidean_distance(X[i], X[j]) for j in range(n_sar

    # Compute the silhouette coefficient for each sample
    s = np.zeros(n_samples)
    for i in range(n_samples):
        s[i] = (b[i] - a[i]) / max(a[i], b[i])

    # Compute the overall average silhouette coefficient
    silhouette_coef = np.mean(s)

    return silhouette_coef

# Set range of clusters to evalute
```

```
range_n_clusters = range(2, 72)

# Compute the silhouette score for each number of clusters
silhouette_scores = []
for n_clusters in range_n_clusters:
    # Fit the KMeans model with kmeans++ initialization
    kmeans = KMeans(n_clusters=n_clusters, init='k-means++', random_state=RANDOM_SE

    # Predict the cluster labels for each data point
    labels = kmeans.labels_

    # Compute the silhouette coefficient for each number of clusters
    silhouette_coef = silhouette_coefficient(X, labels)

    silhouette_scores.append(silhouette_coef)

# Plot the silhouette score against the number of clusters
plt.plot(range_n_clusters, silhouette_scores)
plt.xlabel('Number of clusters')
plt.ylabel('Silhouette score')
plt.show()

# Identify the optimal number of clusters based on the highest silhouette score
optimal_n_clusters = range_n_clusters[np.argmax(silhouette_scores)]
print(f"The optimal number of clusters is {optimal_n_clusters}")
```



```
The optimal number of clusters is 67
```

B) **[Motivate]** your choice of clusters k.

Answer

As seen above, the optimal choice of clusters k, is 67 or greater. However, this is because it will create a cluster for each point, which surely makes no sense to do. Because this would overfit the data, and would not generelize well to new data. As well as added complexity, interpretations and so on.

Thus, the number of clusters we would choose, is $k = 3$, as this gives us a very general fitting of the data and could generelize better to new data. It has a score of around 0.65, thus giving us a medium structure.

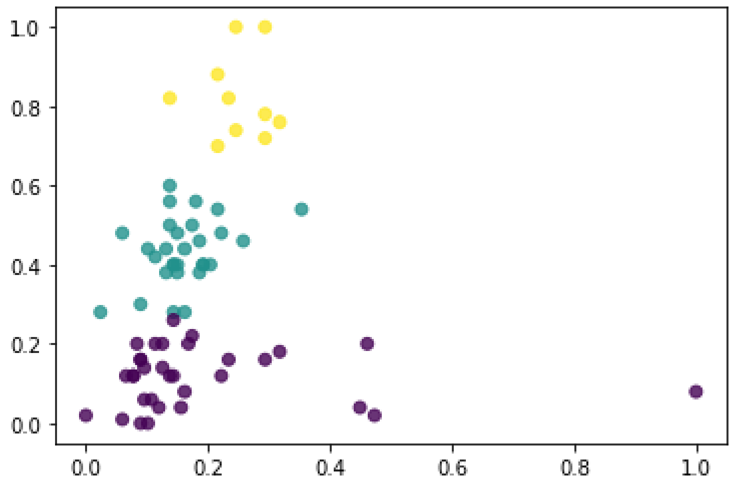
Task 3.1.2 (1 points)

[Implement]

Run k-means on the dataset X, with the number of clusters detected in the previous exercise.

IMPORTANT: You can use the KMeans implementation from scikit-learn.

```
In [ ]:
# Data
X = toy[["sulphates", "alcohol"]].to_numpy()
# Necessary Data normalization!
X_norm = (X - X.min(0)) / X.ptp(0)

# Fit the KMeans model with kmeans++ initialization
kmeans = KMeans(n_clusters=3, init='k-means++', random_state=RANDOM_SEED).fit(X_nor

# Assign the cluster labels to each data point
clusters = kmeans.labels_

plt.scatter(X_norm[:, 0], X_norm[:, 1], alpha=0.8, c=clusters)
plt.show()
```



Task 3.1.3 (6 points)

[Implement]

Kernel K-means and the Gaussian Kernel.

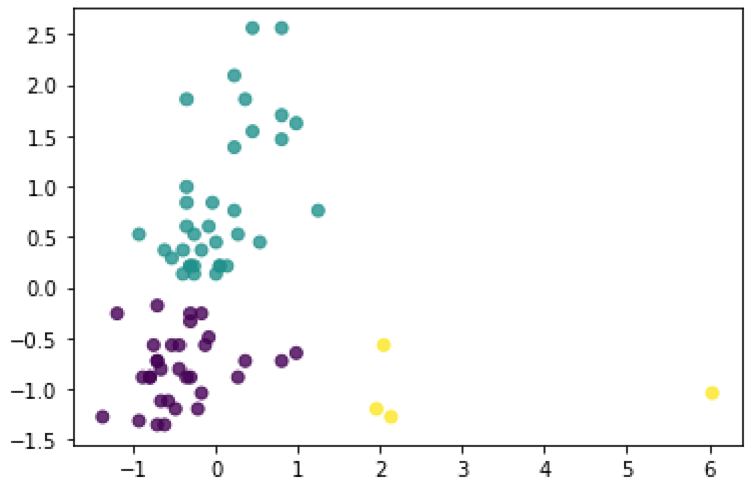
The Gaussian kernel is defined as in the following equation:

$$K\left(\mathbf{x}_i, \mathbf{x}_j\right) = \exp\left(-\frac{\left\|\mathbf{x}_i - \mathbf{x}_j\right\|^2}{2\sigma^2}\right)$$

```
In [ ]:
### YOUR CODE HERE
X = toy[["sulphates", "alcohol"]].to_numpy()

# Necessary Data normalization!
X_norm = (X - X.min(0)) / X.ptp(0)
```

```python
def gaussian_kernel(x, y, sigma=0.8):
    k = 0

    # Calculate the squared Euclidean distance between x and y
    dist_sq = np.linalg.norm(x-y)**2

    # Calculate the value for the Gaussian kernel
    k = np.exp(-dist_sq / (2 * (sigma ** 2)))

    return k



def kernel_kmeans(X, n_clusters, kernel=gaussian_kernel, iters=100, error=.01):
    n_samples, n_features = X.shape

    np.random.seed(RANDOM_SEED)

    # Initialize the cluster centers randomly
    random_sample_idxs = np.random.choice(n_samples, n_clusters)
    centers = X[random_sample_idxs]

    # Iterate until convergence or max_iters reached
    for iteration in range(iters):

        # Set old centers to current centers
        old_centers = centers.copy()


        # Compute pairwise kernel matrix
        K = np.zeros((n_samples, n_clusters))
        for i in range(n_samples):
            for j in range(n_clusters):
                K[i, j] = gaussian_kernel(X[i], centers[j])

        # Assign points to nearest cluster
        # As the values in K are between 0 and 1, the cluster with the highest Kerr
        clusters = np.argmax(K, axis=1)

        # Update cluster centers
        for j in range(n_clusters):
            points = X[clusters == j]
            if len(points) > 0:
                centers[j] = np.mean(points, axis=0)

        # Check for convergence
        if np.linalg.norm(centers - old_centers) < error:
            break

    return clusters

NUMBER_OF_CLUSTERS = 3
SOME_AMOUNT_OF_CLUSTERS = 3

clusters = kernel_kmeans(X_norm, NUMBER_OF_CLUSTERS)

scaler = StandardScaler().fit(X_norm)
X_scaled = scaler.transform(X_norm)
clusters = kernel_kmeans(X_scaled, SOME_AMOUNT_OF_CLUSTERS)
```

```
plt.scatter(X_scaled[:, 0], X_scaled[:, 1], alpha=0.8, c=clusters)
plt.show()
```



Task 3.2 Clustering quality

Task 3.2.1 (6 points)

[Implement] Conditional Entropy (CE) as a measure for clustering quality.

Entropy for a clustering is $H(C) = -\sum_{i=1}^{k} p_{C_i} \log p_{C_i}$.

The **Conditional Entropy** of C **given** T is given by:

$$\mathrm{CE}(C|T) = -\sum_{i=1}^{|C|} \sum_{j=1}^{|T|} \frac{n_{ij}}{n_i} \log \frac{n_{ij}}{n_i}$$

where n_i is the total number of points in cluster C_i and n_{ij} is the number of common points between clusters C_i and T_j

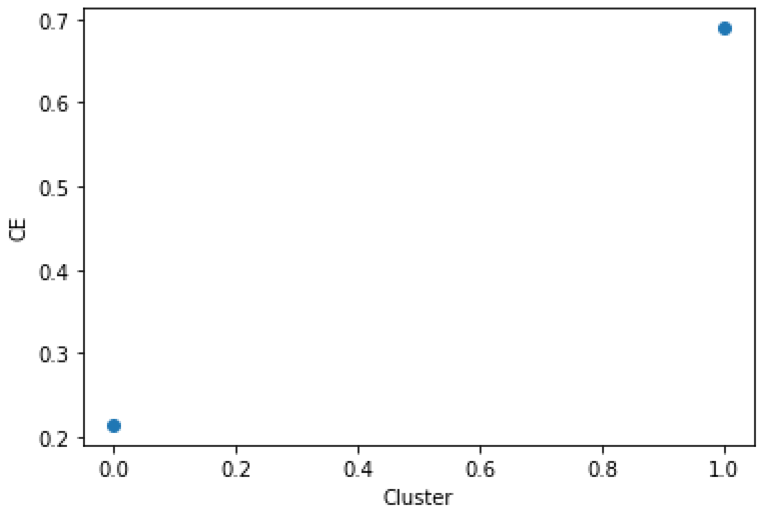
Hint: First implement **Entropy** and then **Conditional Entropy**.

In [ ]:
```
def entropy(C):
    # Let C be a list of clusters
    entropy = 0

    # Total number of data points
    n = sum([len(c) for c in C])

    entropy = - sum([len(c) / n * np.log(len(c) / n) for c in C])

    return entropy


def CE(C, T):
    ce = 0

    for i in range(len(C)):
        # Number of data points in each cluster in C

        n_1 = len(C[i])
        # Number of data points in common between the cluster and T
        for j in range(len(T)):
            n_12 = len(np.intersect1d(C[i], T[j]))


            # Calculate the entropy of C2 given C1
            ce += n_12 / n_1 * np.log(n_12 / n_1)

    return -ce



X = toy[["sulphates", "alcohol"]].to_numpy()

# Necessary Data normalization!
X_norm = (X - X.min(0)) / X.ptp(0)
```

Task 3.2.2 (3 points)

[Implement] Plot the Conditional Entropy (implementation from the Task 3.2.1) among the class labels y and the clusters you found with k-means in Task 3.1.1. Make sure that the number of clusters and the number of class labels is the same.

In [ ]:
```
# Data
X = toy[["sulphates", "alcohol"]].to_numpy()
# Necessary Data normalization!
X_norm = (X - X.min(0)) / X.ptp(0)

# Fit the KMeans model with kmeans++ initialization
kmeans = KMeans(n_clusters=2, init='k-means++', random_state=RANDOM_SEED).fit(X_nor

# Assign the cluster labels to each data point
clusters = kmeans.labels_

# Assign ground truth labels to each data point
class_labels = np.array(toy["quality"])
class_labels = np.array([1 if x == 8 else 0 for x in class_labels])


# Get indicies of points in each cluster
T = np.array([np.where(class_labels == i)[0] for i in range(len(set(class_labels))
C = np.array([np.where(clusters == i)[0] for i in range(len(set(clusters)))]) # Clu


ce = []
for k in range(len(set(clusters))):
    # Calculate the conditional entropy for the current label
    score = CE([C[k]], T)  #added list around C[k]because CE() expects a list of cl

    ce.append(score)


plt.plot(range(len(set(clusters))), ce, 'o')
plt.xlabel('Cluster')
plt.ylabel('CE')
plt.show()
```



[Motivate]

A) Reason about the measure, is the measure influenced by the size of the clusters?

Answer

Yes it is. If the cluster is large, then n_i will be a large number, and the intersection with the ground truth will probably be smaller, thus giving us a higher conditional entropy (CE). While a smaller cluster may be more likely to share all of its points with the ground truth cluster, this would make the fraction go closer to 1, thus pushing the CE towards 0. However, if the cluster only consists of two points, it only needs to misclassify one point, for the fraction to be one half, thus resulting to a higher CE.

[Describe]

B) What does the measure capture?

Answer

The conditional entropy captures the degree to which the clustering reflects the ground truth. If the clustering perfectly matches the ground truth, then the conditional entropy will be zero, indicating that no additional information is gained by knowing the ground truth. On the other hand, if the conditional entropy is very high, this indicates that our clustering is completely random or has a very high uncertainty with respect to our ground truth.

Task 3.2.3 (4 points)

[Implement]

Provide an implementation of purity. Recall that purity is the weighted sum of the individual $purity_i = \frac{1}{|C_i|} \max_{j=1..k}\{n_{ij}\}$ values where n_{ij} is the number of common points in cluster C_i and ground-truth cluster j obtained from the labels y.

In [ ]:
```
### YOUR CODE HERE
class_labels = np.array(toy["quality"])
class_labels = np.array([1 if x == 8 else 0 for x in class_labels])

# Fit the KMeans model with kmeans++ initialization
kmeans = KMeans(n_clusters=2, init='k-means++', random_state=RANDOM_SEED).fit(X_nor

# Assign the cluster labels to each data point
clusters = kmeans.labels_


#we calculate the ground truth clusters and the clusters obtained by k-means:
T = np.array([np.where(class_labels == i)[0] for i in range(2)]) # Ground-truth clu
C = np.array([np.where(clusters == i)[0] for i in range(2)]) # Clusters obtained by



### YOUR CODE HERE

## C is the clustering from k-means and T is the ground truth cluster assignments.
def purity(C, T):
    purity = 0
    n = sum([len(c) for c in C])
    ### YOUR CODE HERE
    for i in range(len(C)):

        purity += np.amax([len(np.intersect1d(C[i], T[j])) for j in range(len(T))])

    ### YOUR CODE HERE
    return purity/n



def individual_purity(C, T):
    purity = []
    ### YOUR CODE HERE
    for i in range(len(C)):
        purity.append(np.amax([len(np.intersect1d(C[i], T[j])) for j in range(len(

    ### YOUR CODE HERE
    return purity


print('Purity: {}, CE: {}'.format(purity(C,T), CE(C,T)))
```

```
Purity: 0.7464788732394366, CE: 0.9040283549515653
```

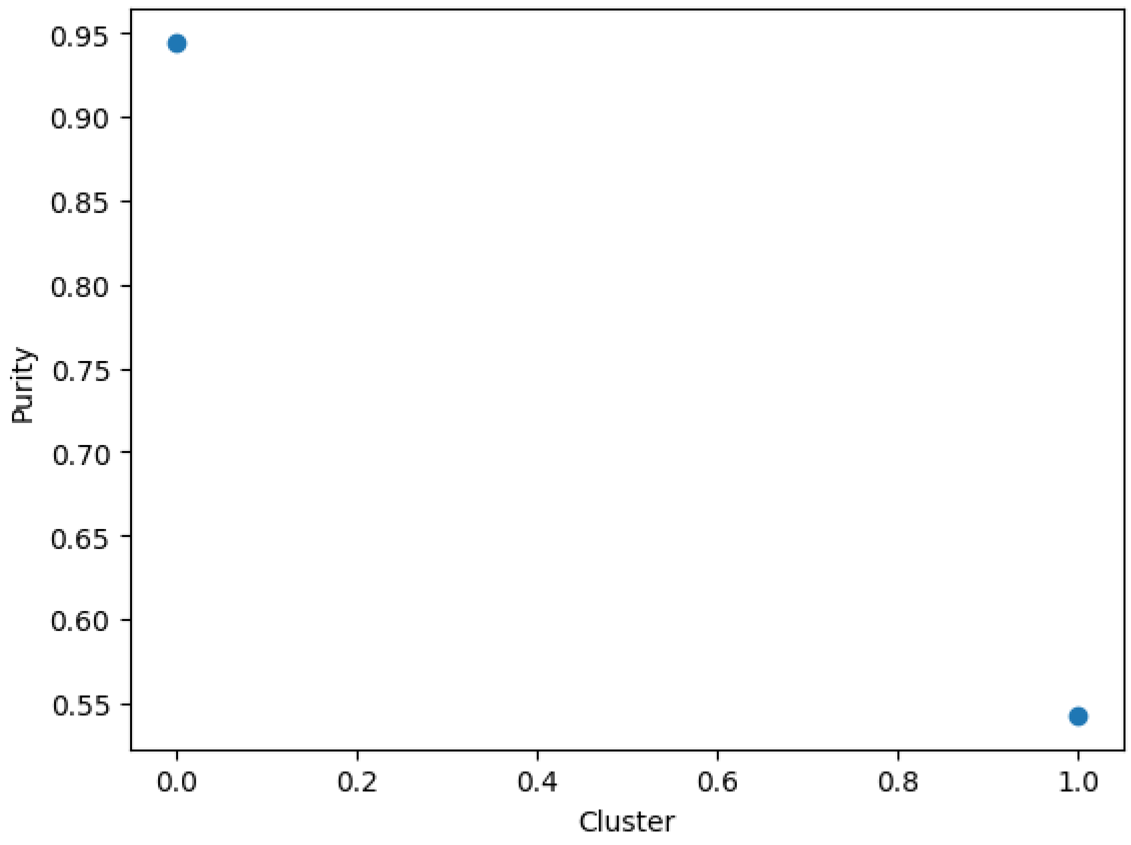
Task 3.2.4 (2 points)

A) **[Implement]**

Plot the purity of the clusters obtained by k-means in Task 3.1.1.

In [ ]:
```
### YOUR CODE HERE

# Calculate the conditional entropy for the current label
purities = individual_purity(C, T)  #added list around C[k] and T[k] because CE() e


plt.plot(range(2), purities, 'o')
plt.xlabel('Cluster')
plt.ylabel('Purity')
```

```python
plt.show()
### YOUR CODE HERE
```



B) **[Motivate]**

Compare purity with **Conditional Entropy (CE)**. Which measure is preferable? (1) Check the correct box below and (2) motivate your answer.

- ☐ **CE** is preferable because it uses all the points
- ☐ Purity is preferable because it is less computational demanding
- ☑ **CE** is preferrable because it does not favor small clusters
- ☐ Purity is preferrable because it tends to favor balanced clusters.

Answer Conditional Entropy is preferrable because it does not favor small clusters. The reason is that purity tends to be swayed by a large number of small clusters which may lead to favoring imbalanced clustering solutions. However, CE, on the other hand, does not have this bias since it takes all data points into account when the quality of a clustering solution is evaluated. The drawback of CE is that it is typically more computationally demanding than purity, since it involves computing the joint and conditional probabilities of the data.

Task 3.3 OPTICS

Task 3.3.1 (7 point)

[Implement] the OPTICS algorithm

If you do not remember OPTICS, check the slides or the lecture notes.

In [ ]:
```python
# Defince function to calculate the distances
def dist(X):
    n = len(X)

    # Find the distance matrix
    dist_matrix = np.zeros((n, n))
    for i in range(n):
        for j in range(i+1, n):
            dist_matrix[i,j] = np.linalg.norm(X[i] - X[j])
            dist_matrix[j,i] = dist_matrix[i,j]

    return dist_matrix

# Define function to calculate the core distances
def cdist(X, eps, min_samples):
        n = len(X)

        # Get the distance matrix
        dist_matrix = dist(X)

        # Find the core distances
        core_distances = np.zeros(n)
        for i in range(n):

            # Check if the point is a core point
            if np.sum(dist_matrix[i,:] < eps) < min_samples:
                core_distances[i] = np.inf # Use inf as ?
            else:
                # If it is a core point, calculate the core distance
                sorted_distances = np.sort(dist_matrix[i,:])
                # Return the distance to the point making it a coredistance
                core_distances[i] = sorted_distances[min_samples-1]

        return core_distances


def rdist(X,eps,min_samples):
    ### YOUR CODE HERE
    n = len(X)

    # Get the distance matrix
    dist_matrix = dist(X)

    # Get the core distances
    core_dist = cdist(X, eps, min_samples)

    # Find the reachability distances
    reach_dist = np.ones((n,n)) * np.inf
    for i in range(len(X)):

        # Find the indices of all neighbors within eps distance
        neighbors = np.where(dist_matrix[i] <= eps)[0]

        # If the point is not a core point, skip it
        if len(neighbors) < min_samples:
            continue

        for j in neighbors:
            if j != i:
                reach_dist[i,j] = max(core_dist[i], dist_matrix[i,j])
                reach_dist[j,i] = reach_dist[i,j].copy()

    return reach_dist


def optics(X, eps, min_samples):
    ### YOUR CODE HERE
    n = len(X)

    # Get the distance matrix
    dist_matrix = dist(X)

    # Get the core distances
    core_dist = cdist(X, eps, min_samples)

    # Get the reachability distances
    reach_dist = rdist(X, eps, min_samples)

    # Define the outliers
    outliers = np.where(core_dist == np.inf)[0]

    # Find the order of the points
    rdis = np.ones(n) * np.inf
    used_indecies = []

    # Initizilsation
    i = 0
    kluster = 0

    cluster = np.zeros(n)

    while len(used_indecies) < n:
        used_indecies.append(i)
        cluster[i] = kluster

        # Pull out the  reachability distances from the current point
        reach_dist_temp = reach_dist[i,:].copy()

        # Ignore distances to points that have already been used
        reach_dist_temp[used_indecies] = np.inf

        # If there are no more points to use, find the closest point outside of the
        if min(reach_dist_temp) == np.inf:

            # Pull out the distances to all points from the current point
            dist_matrix_temp = dist_matrix[i,:]
            dist_matrix_temp[used_indecies] = np.inf

            # Find the closest point outside of the current neighborhood
            j = np.argmin(dist_matrix_temp)

            # Set the reachability distance to the closest point outside of the cur
            rdis[i] = reach_dist[i,j]

            # Continue from the new point
            i = j

            # Start new cluster
            kluster += 1

            continue
        # Get the index of the point with the smallest reachability distance
        j = np.argmin(reach_dist_temp)

        # Set the reachability distance to the smallest reachability distance
        rdis[i] = reach_dist[i, j].copy()

        # Continue from the new point
        i = j

    # Set the reachability distance of the first point to infinity
    rdis[0] = np.inf

    # Set the cluster of the outliers to -1
    cluster[outliers] = -1

    ### YOUR CODE HERE
    return cluster
```

Task 3.3.2 (1 points)

[Implement]

A) Run OPTICS with parameters $\varepsilon = 0.07, minPts = 3$.

In [ ]:
```
### YOUR CODE HERE

optics(X_norm, 0.07, 3)

### YOUR CODE HERE
```

Out[ ]:
```
array([ 0.,  1.,  1., -1.,  1., -1.,  2., -1., -1.,  0., -1., -1.,  1.,
        1., 15.,  7.,  7.,  7.,  1., 15., -1., -1.,  4.,  4.,  1.,  1.,
       -1.,  1.,  3.,  3.,  3., -1.,  1.,  3.,  3.,  7.,  3., -1.,  1.,
        3.,  1.,  1., -1.,  3., -1.,  3.,  3., 16.,  3.,  1.,  3., 16.,
        3.,  3., 16., -1.,  6.,  1.,  3.,  1., -1., -1.,  4., -1.,  1.,
        3.,  1.,  1.,  4.,  0.,  3.])
```

B) **[Motivate]**

Compare the results of OPTICS with those of k-means. Which of the two methods two achieve a better **CE**?

Answer

K-means tends to perform well when the clusters in the data are well-defined, spherical, and non-overlapping. This is because k-means attempts to minimize the sum of squared distances between data points and their assigned cluster center. This approach is effective for data that has a clear separation between the clusters.

OPTICS, on the other hand, tends to perform well when the data contains clusters with different shapes and densities. OPTICS can identify clusters of varying shapes and densities because it uses a hierarchical clustering approach based on density-reachability. It can identify the core points, border points, and noise points, which can help improve the CE.

Task 3.3.3 (6 points)

[Implement] a simple subspace clustering algorithm.

1. Take all subsets of 2,3 attributes. Beware that you should only use the numerical attributes.
2. Run OPTICS on each subset.
3. Compute **CE** for each subset.
4. Keep the k subsets with the largest **CE**.

IMPORTANT: You may have to experiment a lot with eps and MinPts to get reasonable clusters. You are allowed to use **itertools** library to iterate over all subsets of size 2 and 3.

In [ ]:
```
# Necessary Data normalization!
X_pt = toy.to_numpy()
X_norm_pt = (X_pt - X_pt.min(0)) / X_pt.ptp(0)

### YOUR CODE HERE


### YOUR CODE HERE
```

```
Top 3 Subsets: []
```

Part 4 Outlier detection

In this exercise we will work with outlier detection techniques and analyze their performance on the small dataset. Before starting the exercise, run the code below.

In [ ]:
```
X_small = toy[["sulphates", "alcohol"]].to_numpy()

X_norm = (X_small - X_small.min(0)) / X_small.ptp(0)
```

Task 4.1 (DBoutliers)

We will now compare two outlier detection techniques.

Task 4.1.1 (6 points)

[Implement] a simple distance-based outlier detector. This is the distance-based outlier detection from the lectures, where a point is considered an outlier if at most a fraction pi of the other points have a distance less of than eps to it.

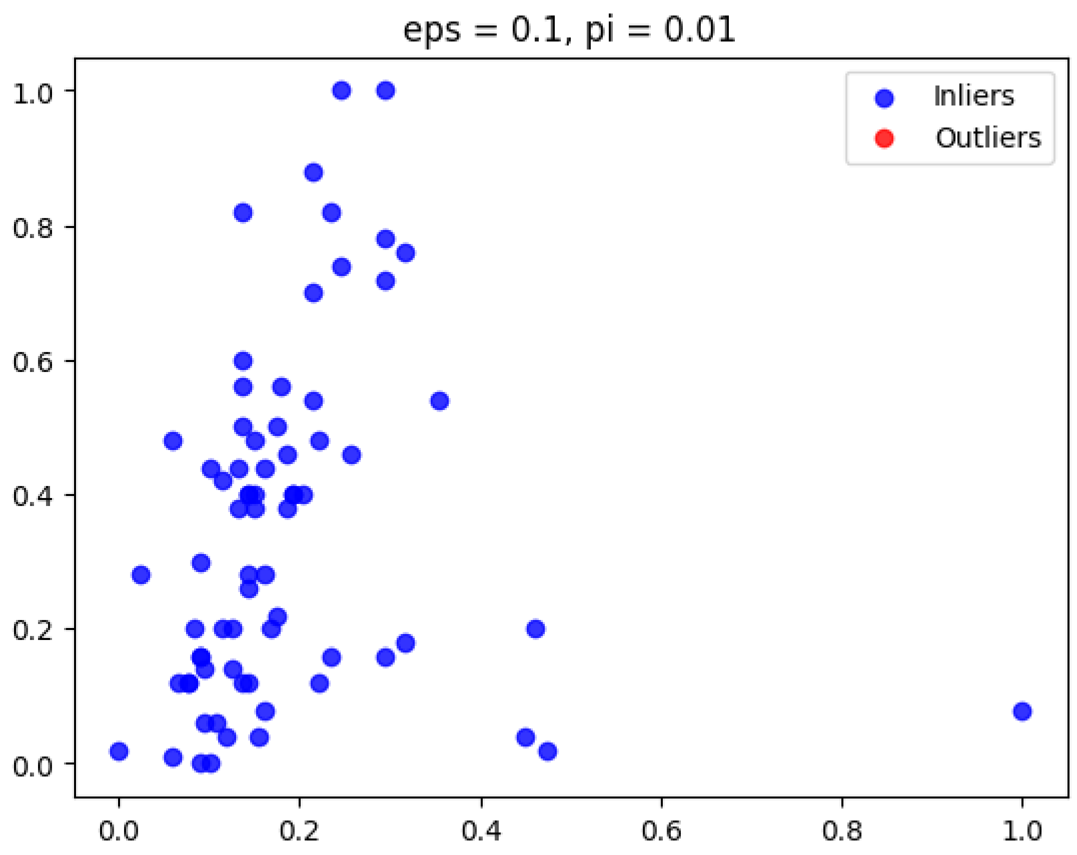
In [ ]:
```
def DBOutliers(X, eps, pi):
    outliers = None

    # Total number of data points
    n_samples = X.shape[0]

    # Initialize the matrix of distances
    Dist = np.zeros((n_samples, n_samples))
    # Compute the pairwise distances
    for i in range(n_samples):
        for j in range(n_samples):
            Dist[i, j] = np.linalg.norm(X[i] - X[j])
            Dist[j, i] = Dist[i, j] # Symmetric matrix

    # Number of neighbors within eps
    n_neighbors = np.sum(Dist < eps, axis=1)

    # Extract the indices of the outliers
    outliers = np.where(n_neighbors / n_samples <= pi)[0]

    return outliers
```

Task 4.1.2 (2 points)

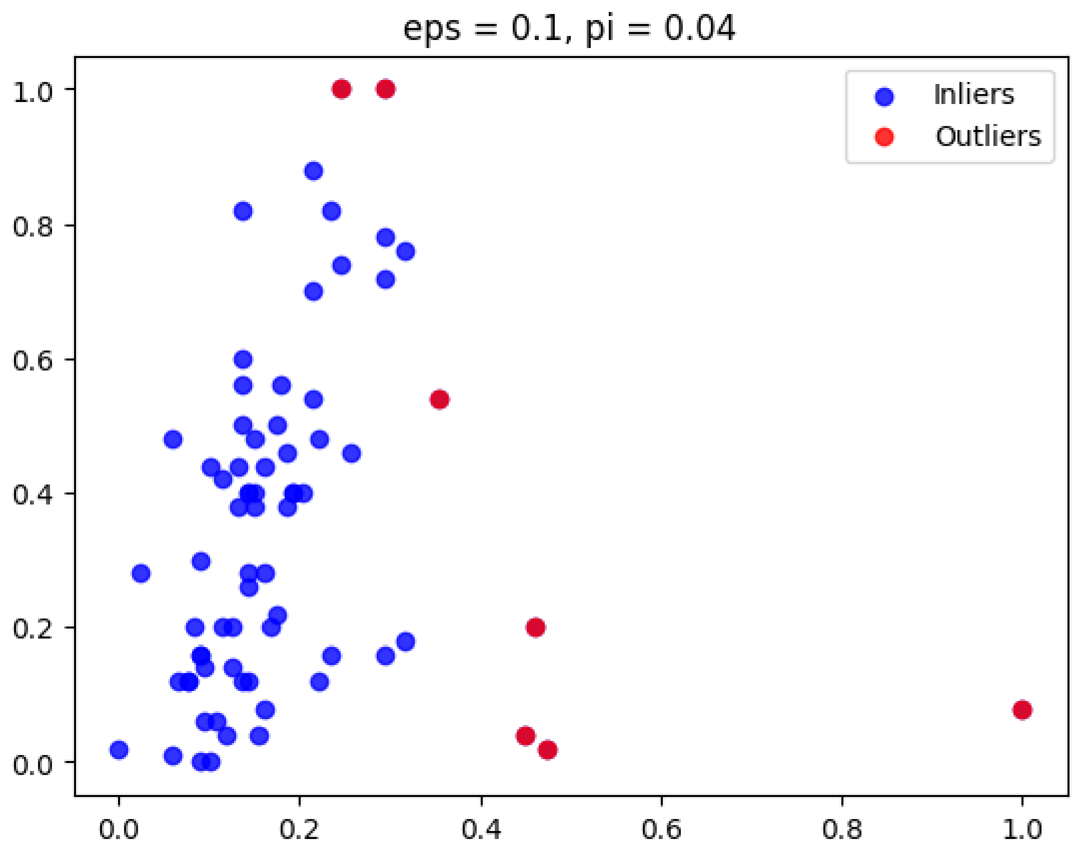
A) **[Implement]** DBOutliers requires tuning the parameters eps, pi. Run the code from Task 4.1.1 with different choices of eps, pi

Note that the data is normalized. Choose two ranges with **at least** 4 values each.

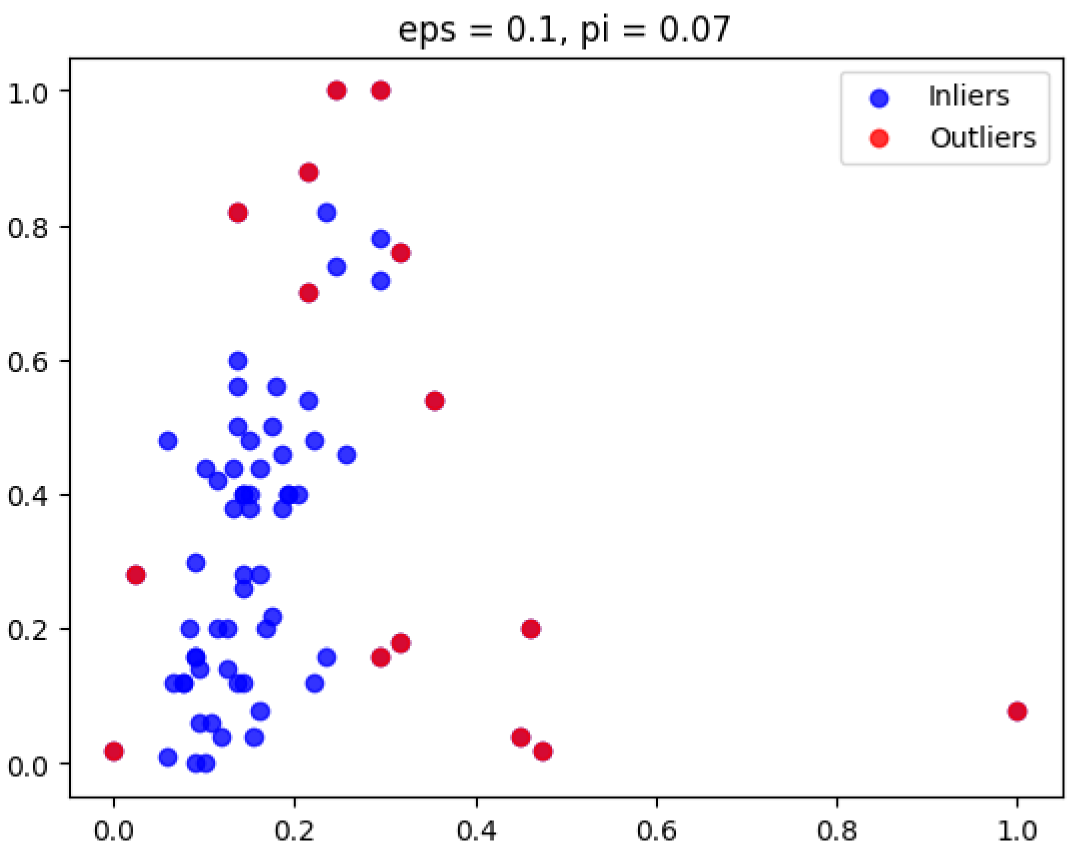
In [ ]:
```
# Range of eps values to test
eps_range = np.linspace(0.1, 0.5, 4)

# Range of pi values to test
pi_range = np.linspace(0.01, 0.1, 4)

# Go through all the combinations of eps and pi and plot the results
for eps in eps_range:
    for pi in pi_range:
        outliers = DBOutliers(X_norm, eps, pi)
        plt.scatter(X_norm[:, 0], X_norm[:, 1], alpha=0.8, c='b')
        plt.scatter(X_norm[outliers, 0], X_norm[outliers, 1], alpha=0.8, c='r')
        plt.title('eps = {}, pi = {}'.format(eps, pi))
        plt.legend(['Inliers', 'Outliers'])
        plt.show()
```

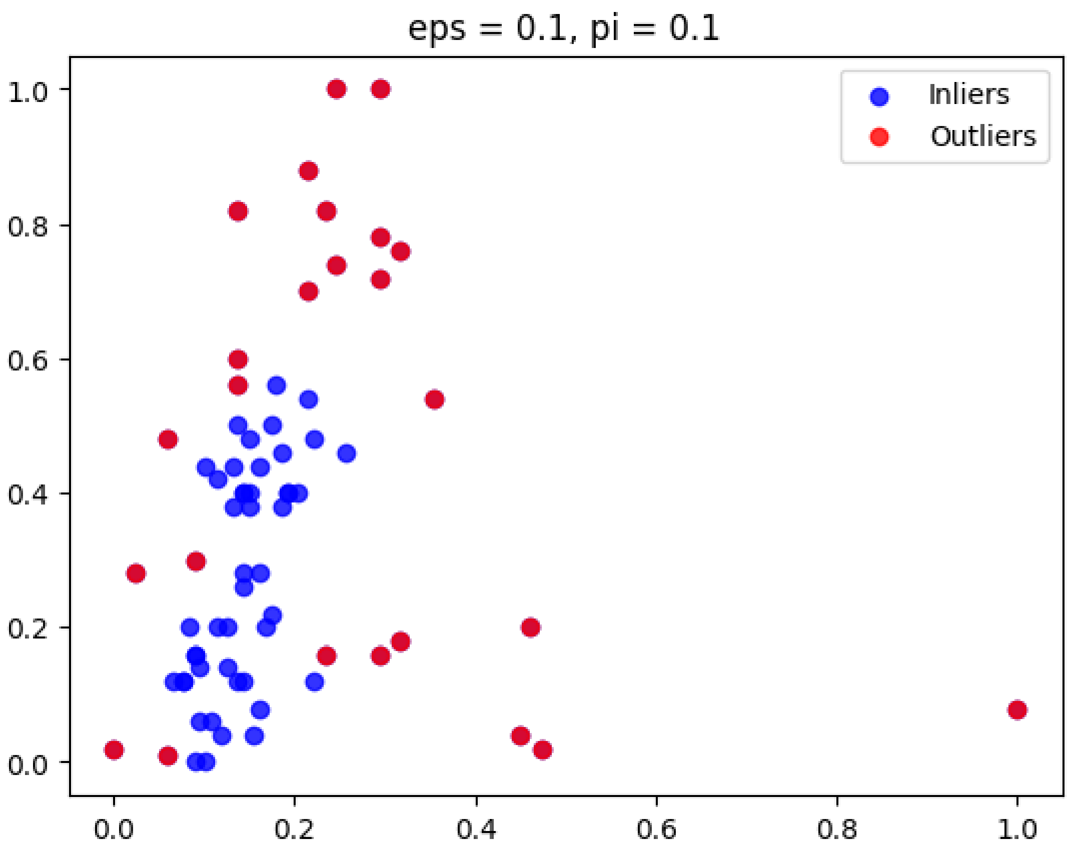


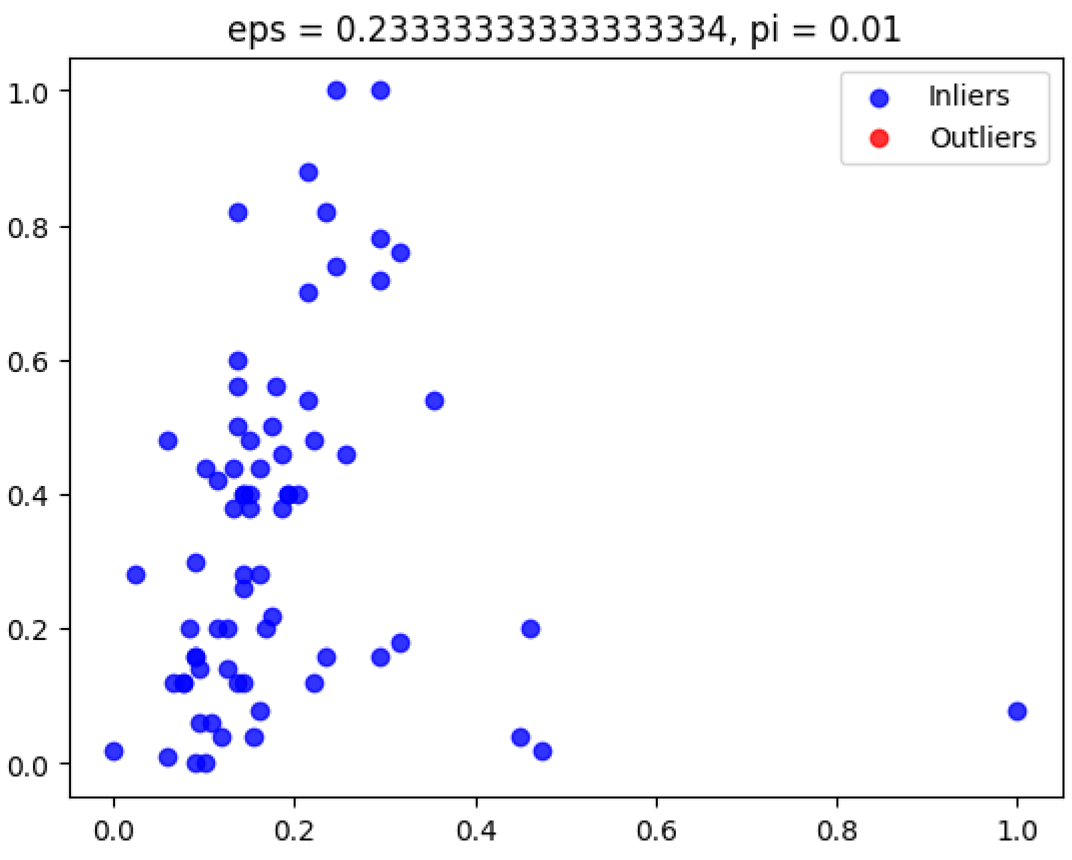
eps = 0.1, pi = 0.01
Inliers
Outliers
0.0
0.2
0.4
0.6
0.8
1.0

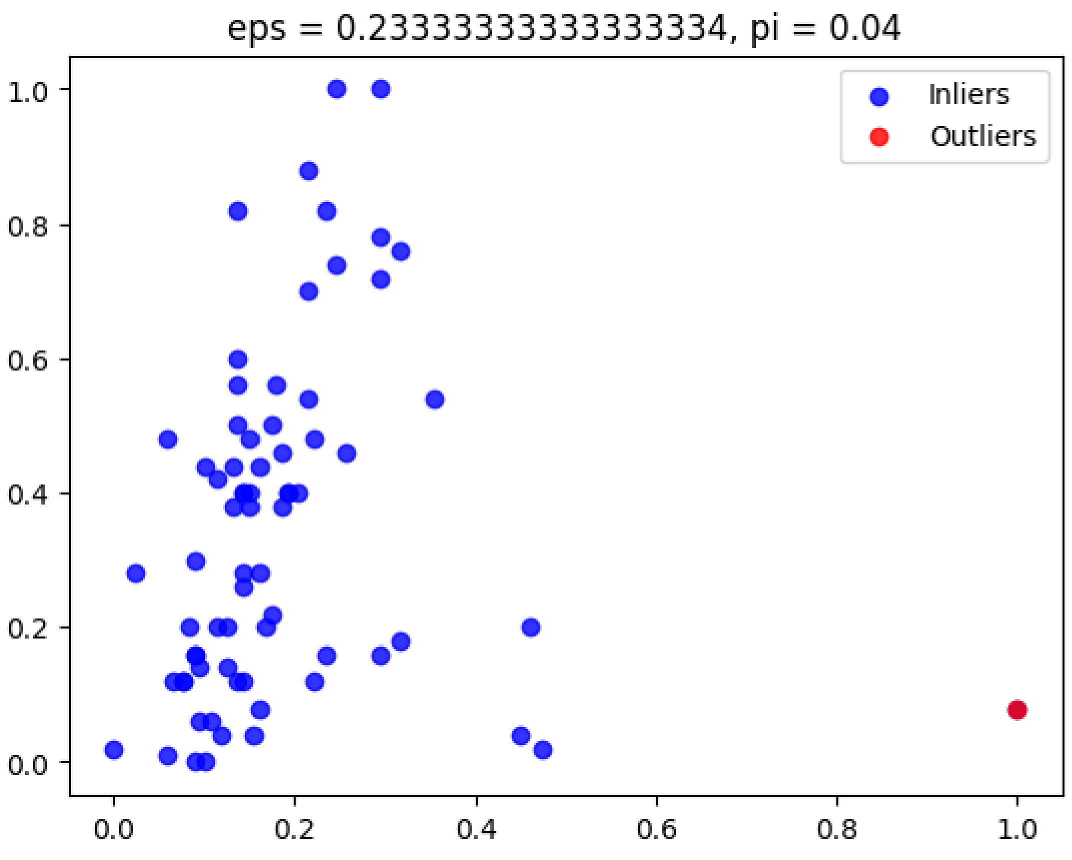


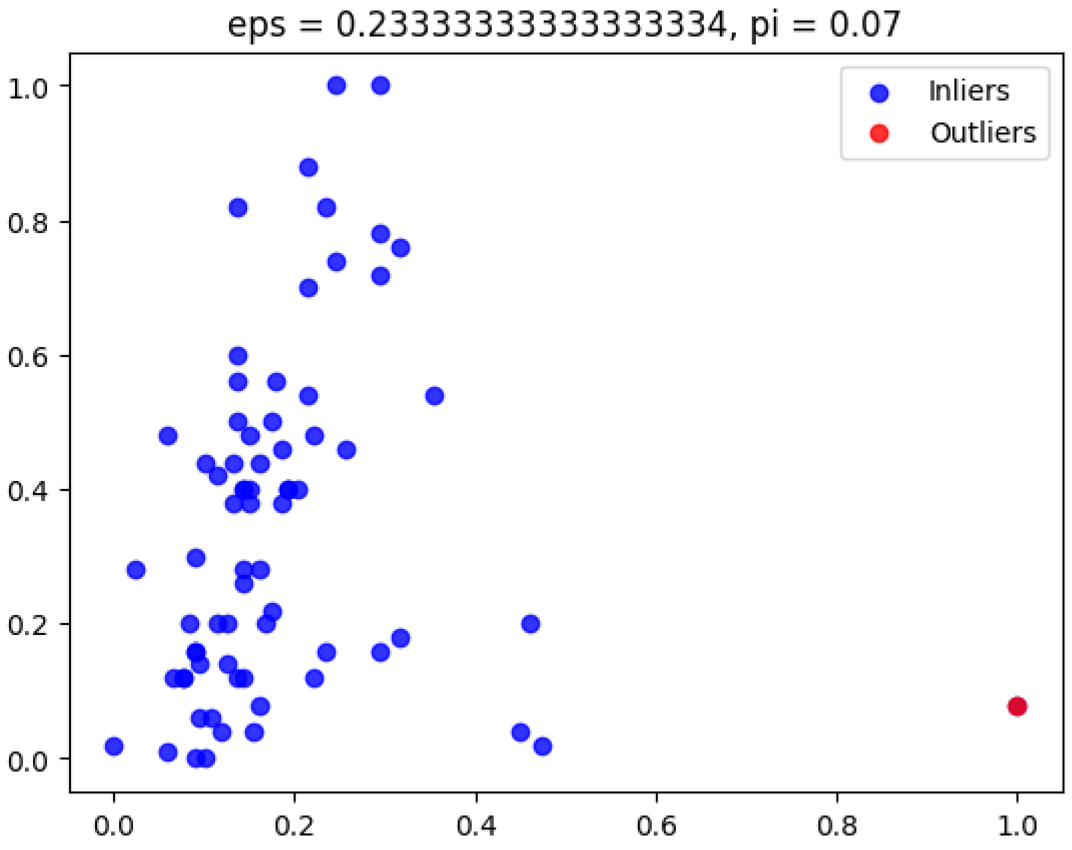
eps = 0.1, pi = 0.04
Inliers
Outliers
0.0
0.2
0.4
0.6
0.8
1.0

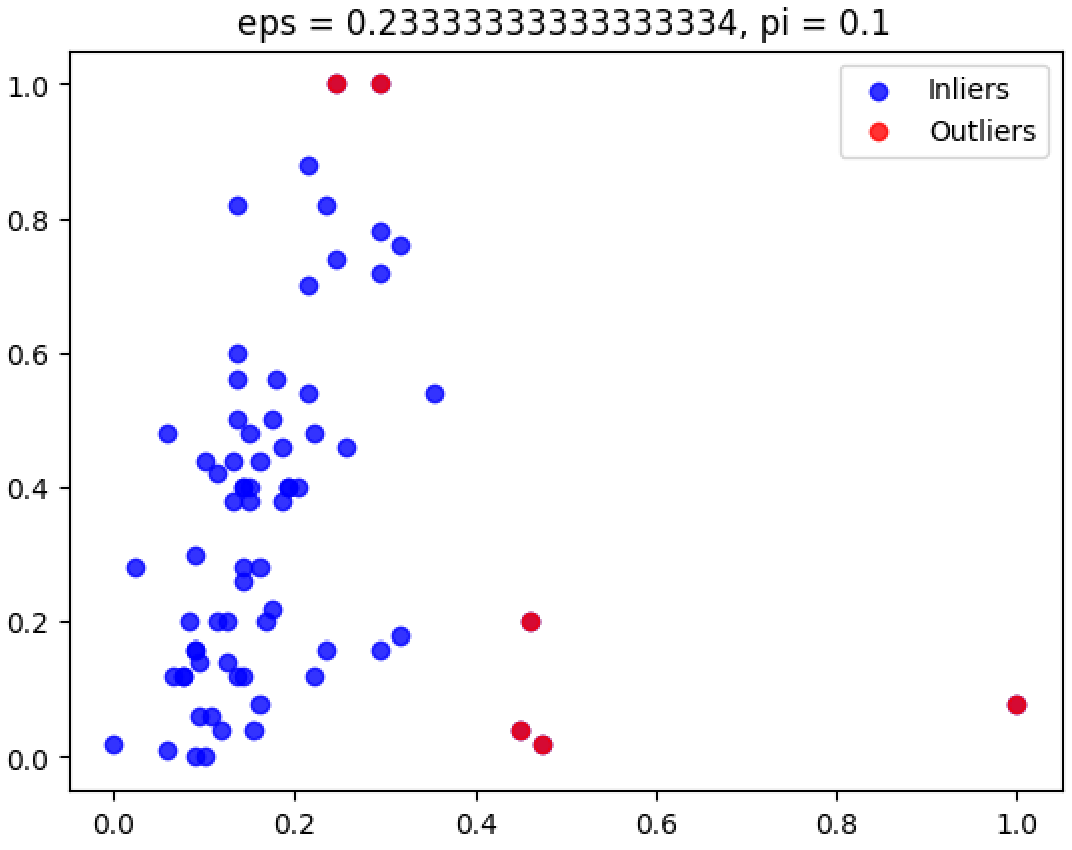

eps = 0.1, pi = 0.07
Inliers
Outliers
0.0
0.2
0.4
0.6
0.8
1.0

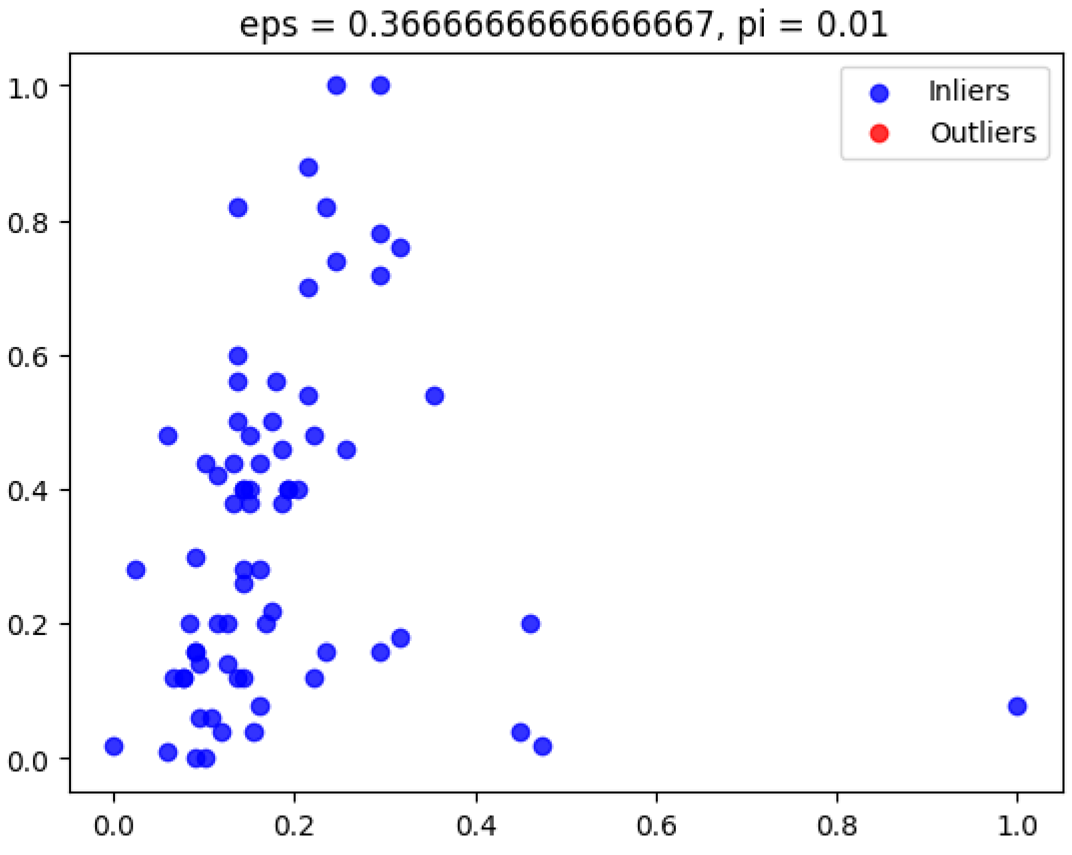

eps = 0.1, pi = 0.1
Inliers
Outliers
0.0
0.2
0.4
0.6
0.8
1.0


eps = 0.23333333333333334, pi = 0.01
Inliers
Outliers
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

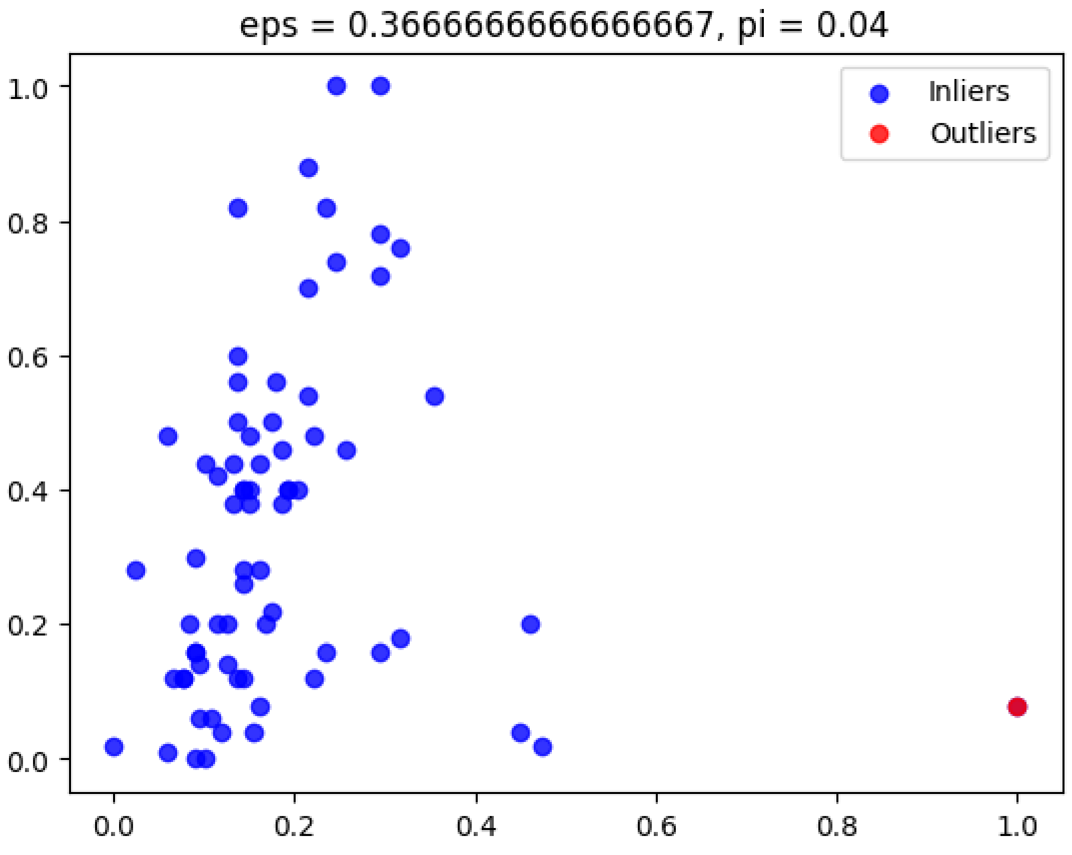

eps = 0.23333333333333334, pi = 0.04
Inliers
Outliers
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0


eps = 0.23333333333333334, pi = 0.07
Inliers
Outliers
0.0
0.2
0.4
0.6
0.8
1.0

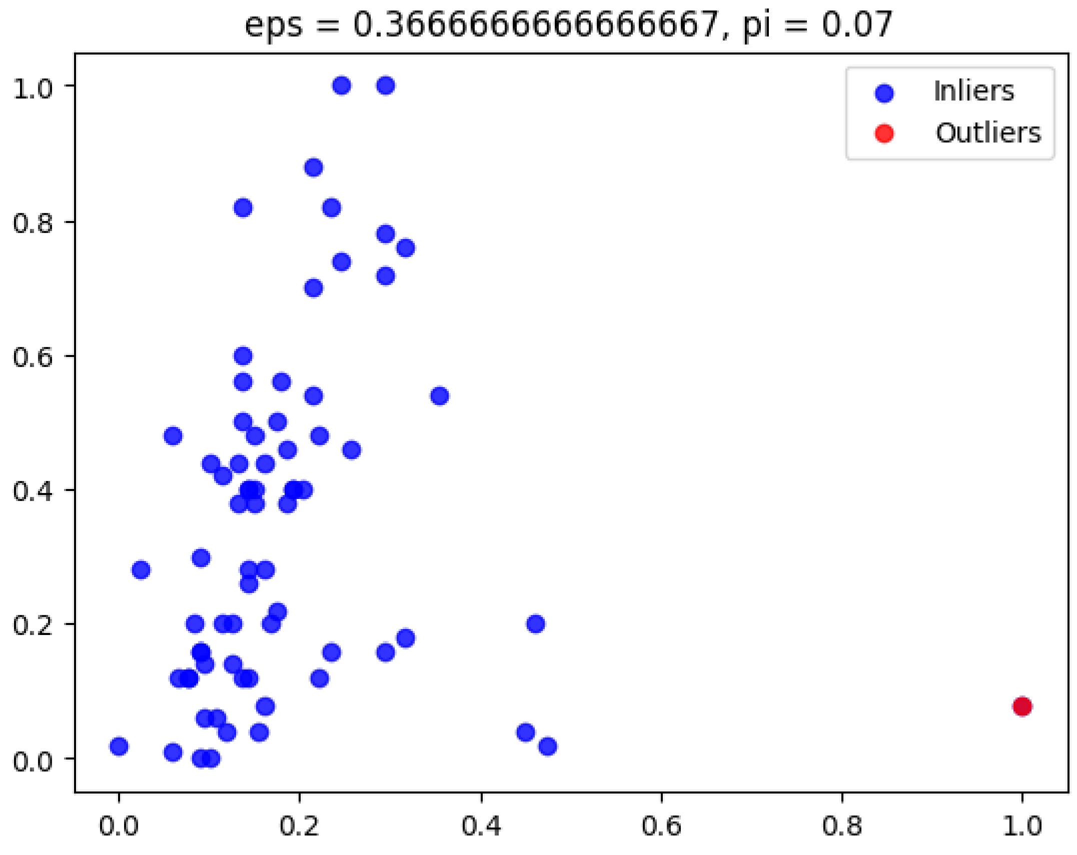

eps = 0.23333333333333334, pi = 0.1
Inliers
Outliers
0.0
0.2
0.4
0.6
0.8
1.0

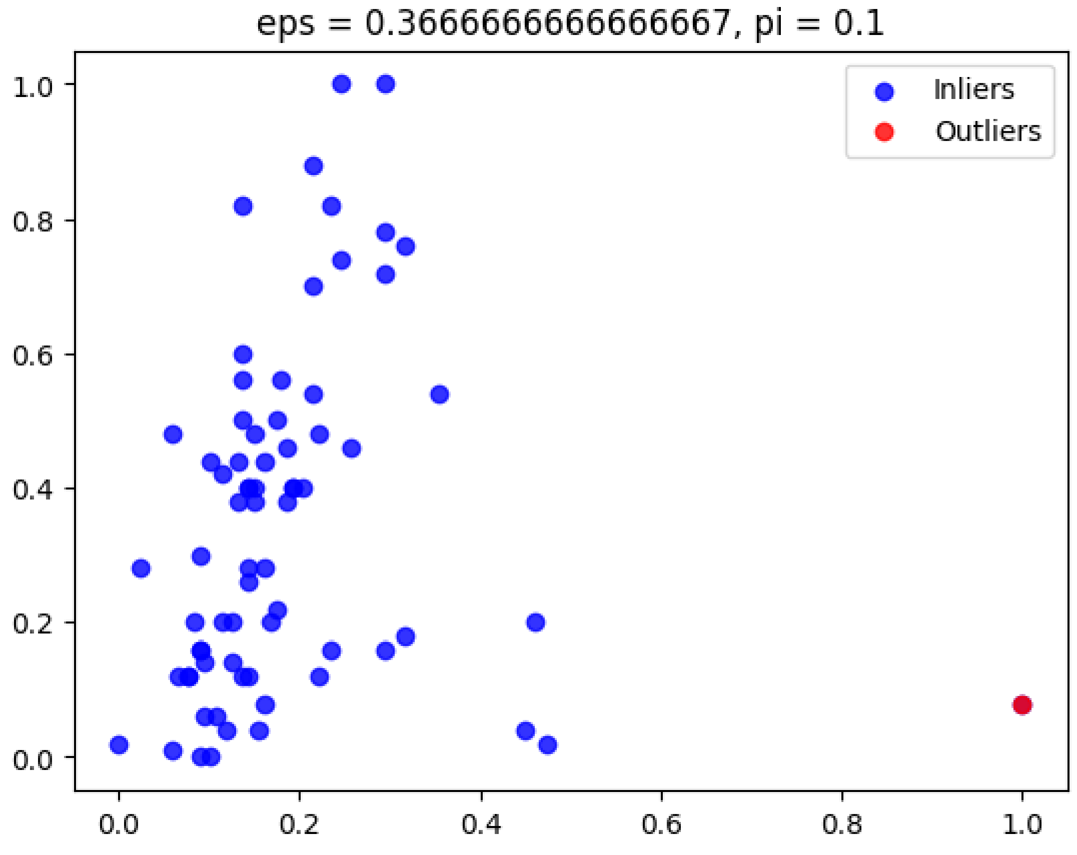


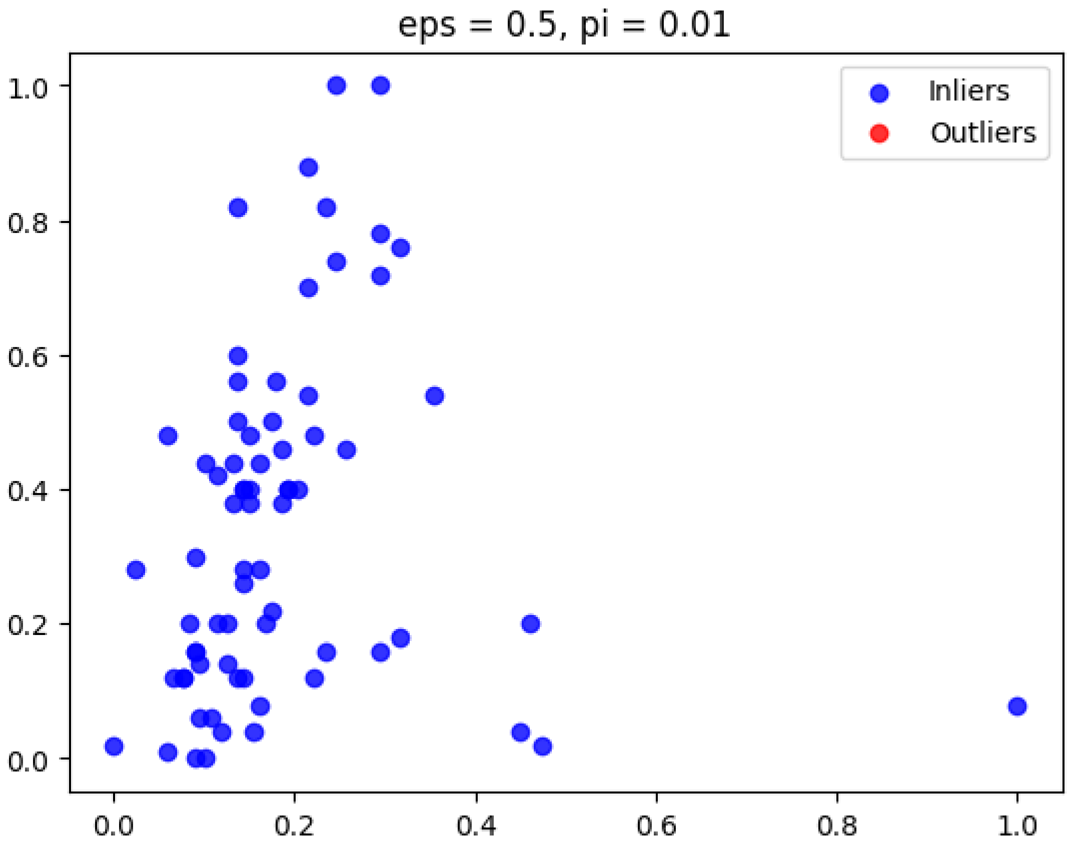
eps = 0.3666666666666667, pi = 0.01
Inliers
Outliers
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

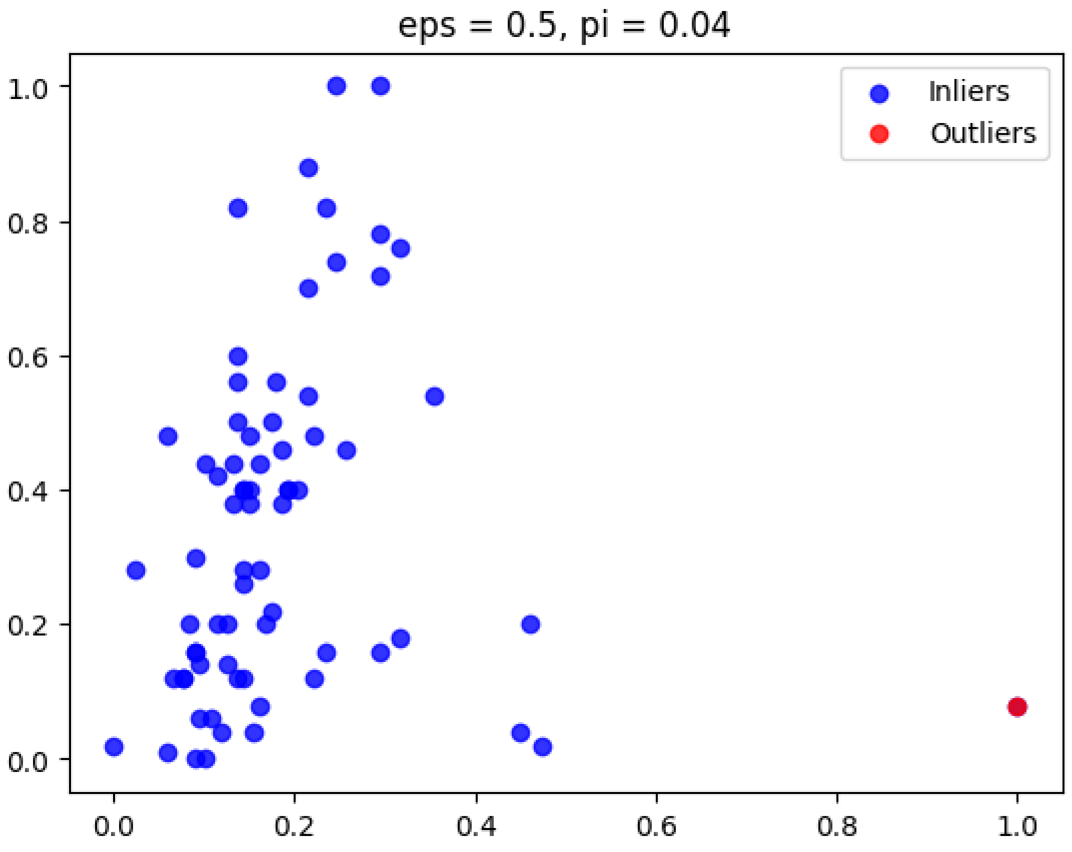


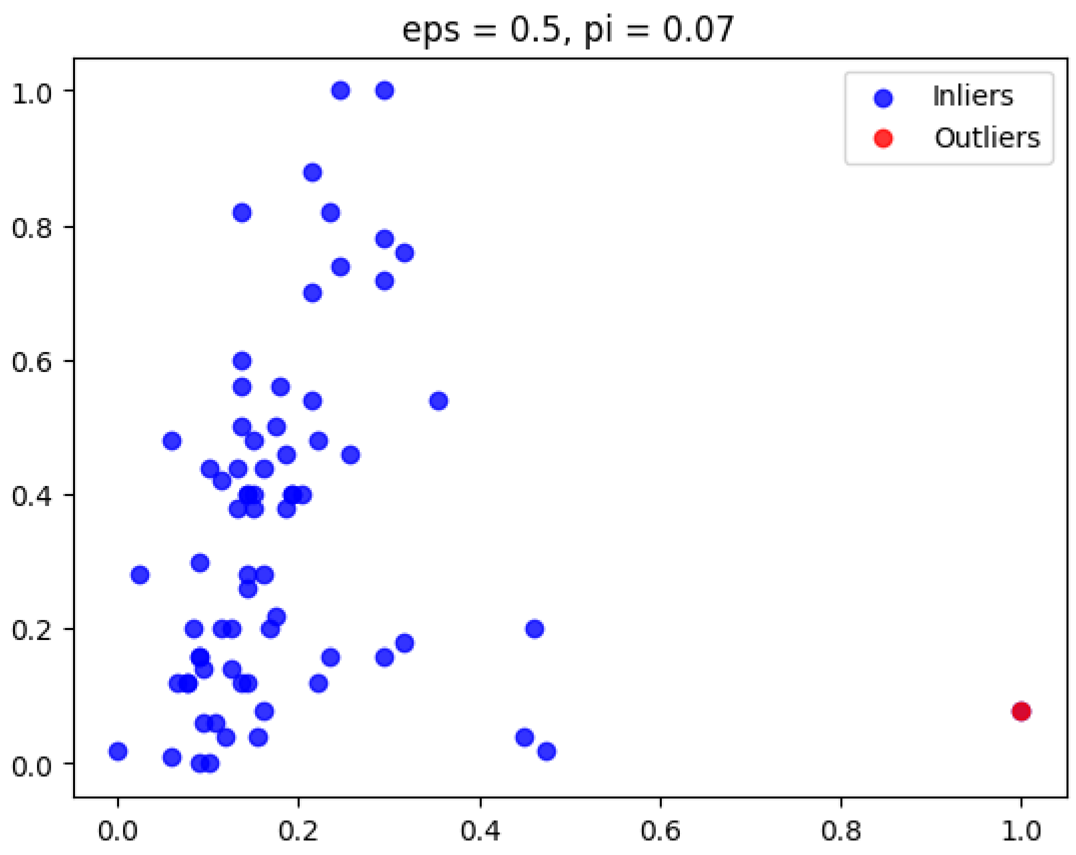
eps = 0.3666666666666667, pi = 0.04
Inliers
Outliers
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

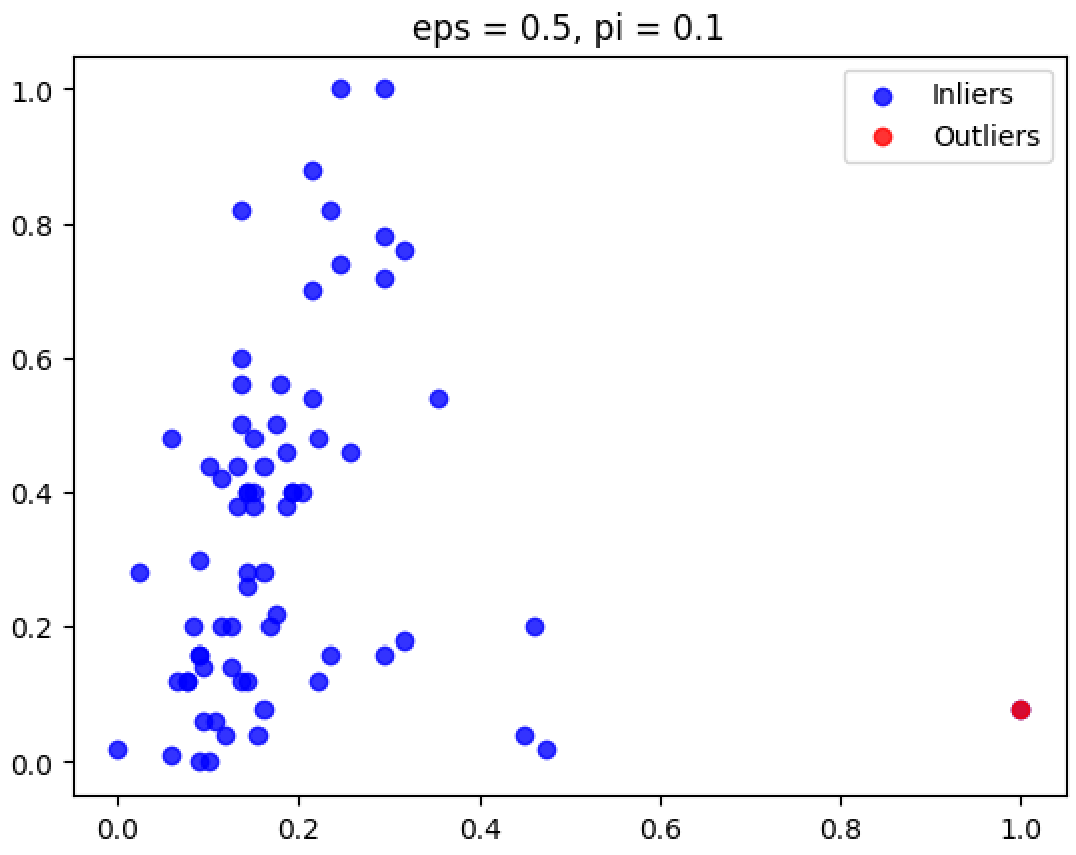

eps = 0.3666666666666667, pi = 0.07
Inliers
Outliers
0.0
0.2
0.4
0.6
0.8
1.0


eps = 0.3666666666666667, pi = 0.1
Inliers
Outliers
0.0
0.2
0.4
0.6
0.8
1.0


eps = 0.5, pi = 0.01
Inliers
Outliers
0.0
0.2
0.4
0.6
0.8
1.0


eps = 0.5, pi = 0.04
Inliers
Outliers
0.0
0.2
0.4
0.6
0.8
1.0





B) **[Motivate]**

Present the results and **discuss** how the results vary with respect to (1) eps and (2) pi.

Answer

eps is a scalar representing the maximum distance for a point to be considered close to another point; and pi is a scalar representing the maximum fraction of other points that can be close to a point for it to be considered an outlier.

From the plots above we see that, the lower the epsilon, then we will have less points that are considered to be neighboors of a point p. Thus, the fraction of points x considered as neighboors to a point p, will be lower, and thus we might detect an higher amount of outliers in our dataset.

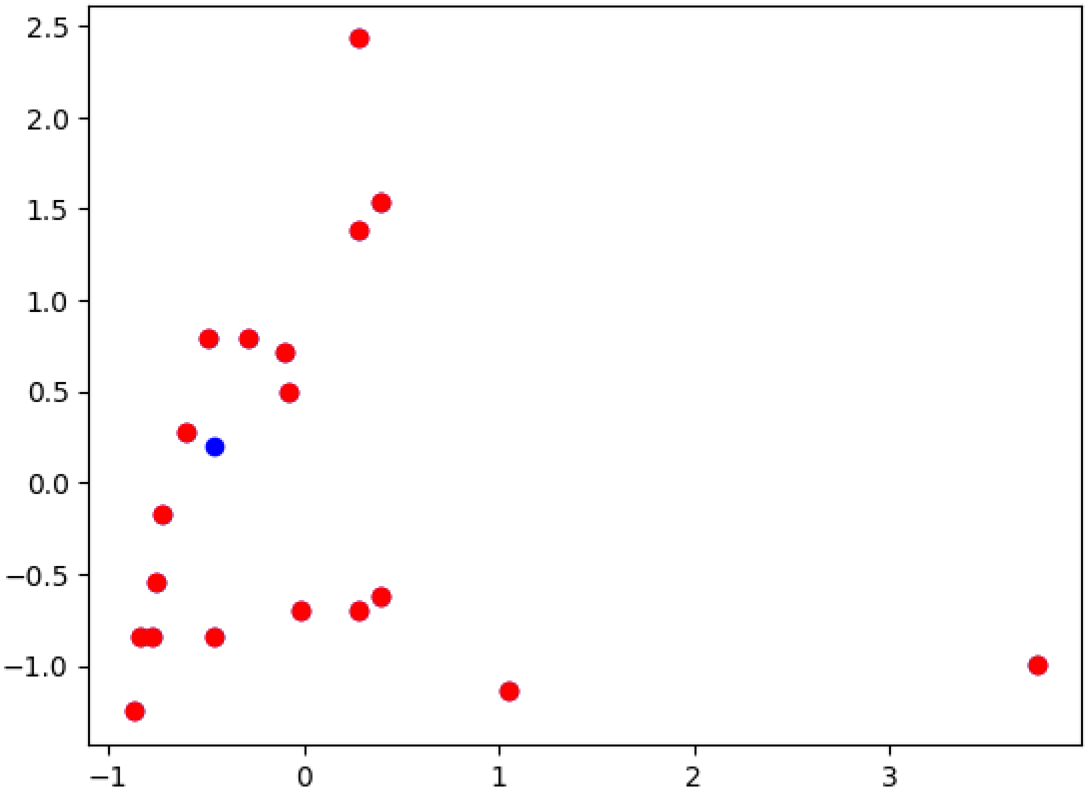
Further, the higher the value of pi, the easier it is for a point to be considered an outlier.

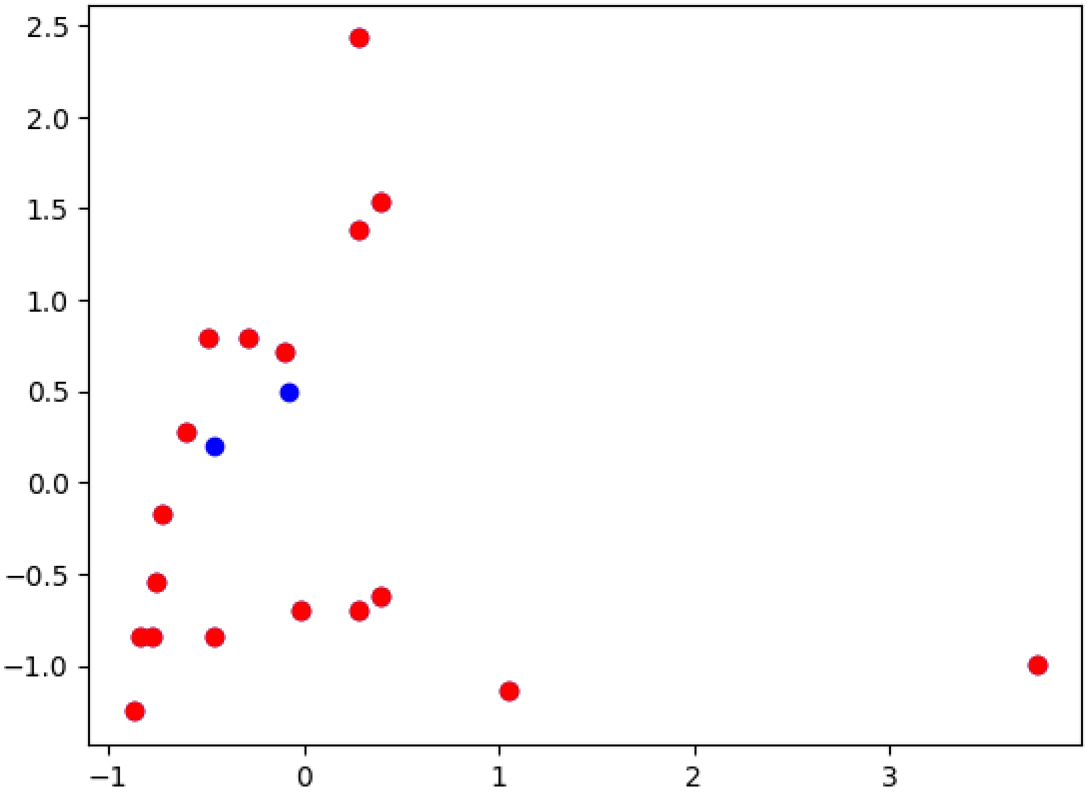
Task 4.1.3 (3 points)

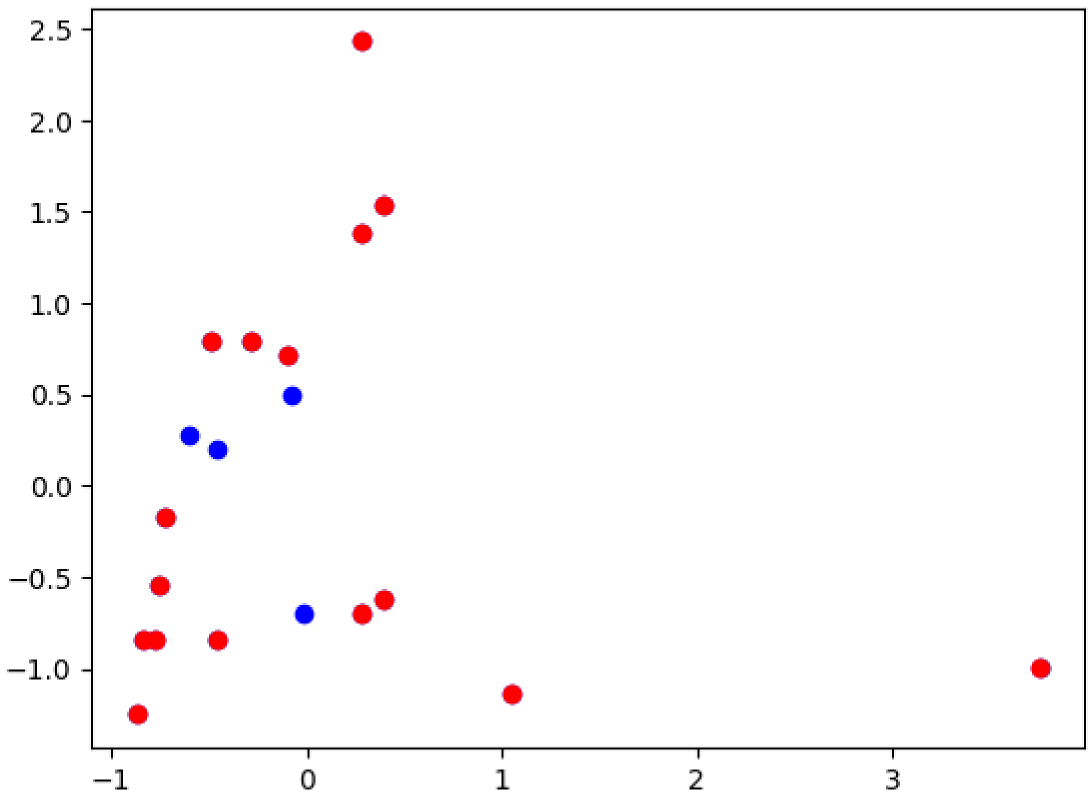
NOTE This is hard but also fun. Since it is not impacting the grade too much, you can choose to invent something new.

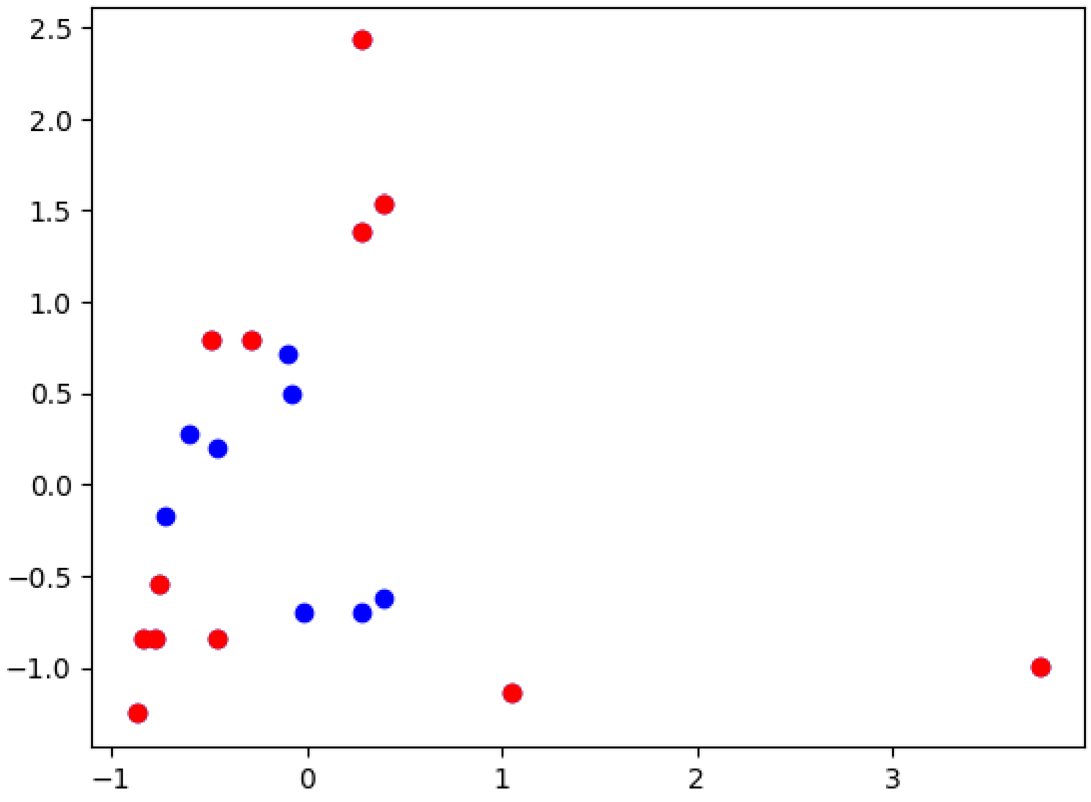
A) Propose and **[Implement]** a heuristic method to tune parameters eps, pi.

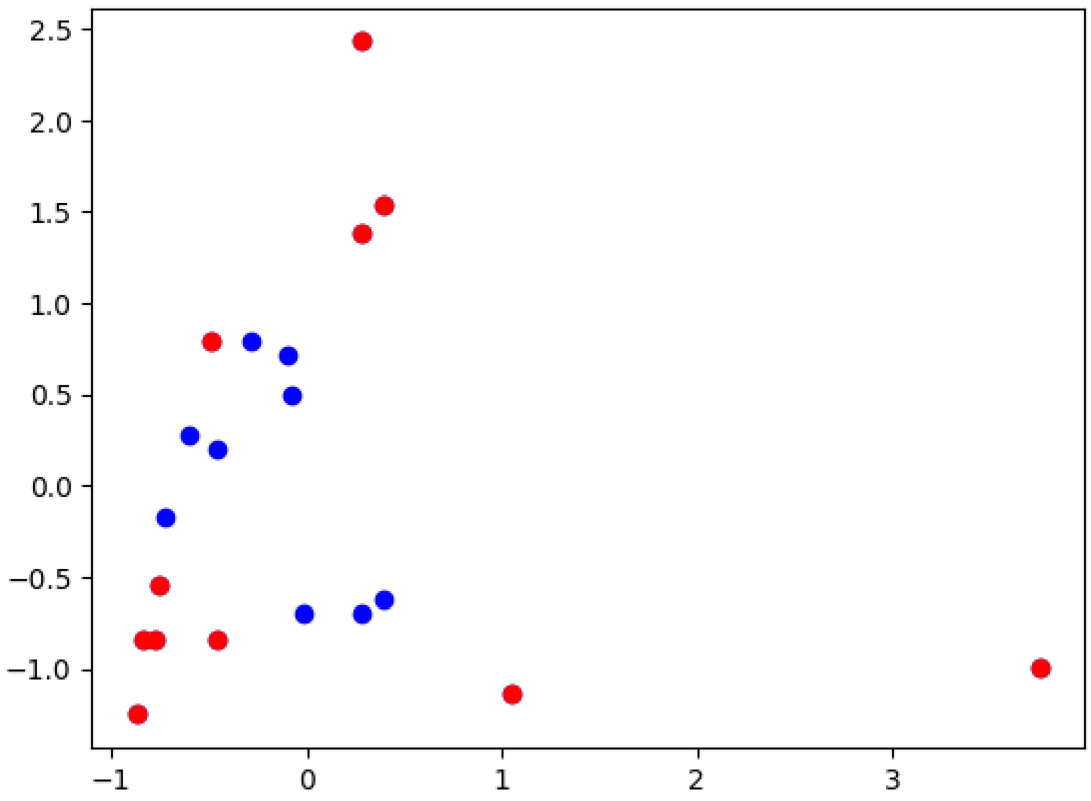
In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt

def tune_dboutliers(X, eps=0.5, pi=2, max_iter=10, plot=True):
    # Define threshold for outlier detection
    threshold = pi * np.median(X)

    # Iteratively adjust eps and pi
    for i in range(max_iter):
        # Detect outliers
        outliers = np.where(np.linalg.norm(X, axis=1) > eps)

        # Plot the data and outliers
        if plot:
            plt.scatter(X[:, 0], X[:, 1], color='b')
            plt.scatter(X[outliers, 0], X[outliers, 1], color='r')
            plt.show()

        # Compute the fraction of outliers and adjust eps and pi accordingly
        frac_outliers = len(outliers[0]) / len(X)
        if frac_outliers > 0.05:
            eps += 0.1
        elif frac_outliers < 0.01:
            eps -= 0.1
        else:
            break
        pi = threshold / np.median(X)


    # Return optimized parameters
    eps_opt = eps
    pi_opt = pi
    return eps_opt, pi_opt

print(tune_dboutliers(X_scaled, eps=0.5, pi=2, max_iter=10, plot=True))
```


2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1
0
1
2
3

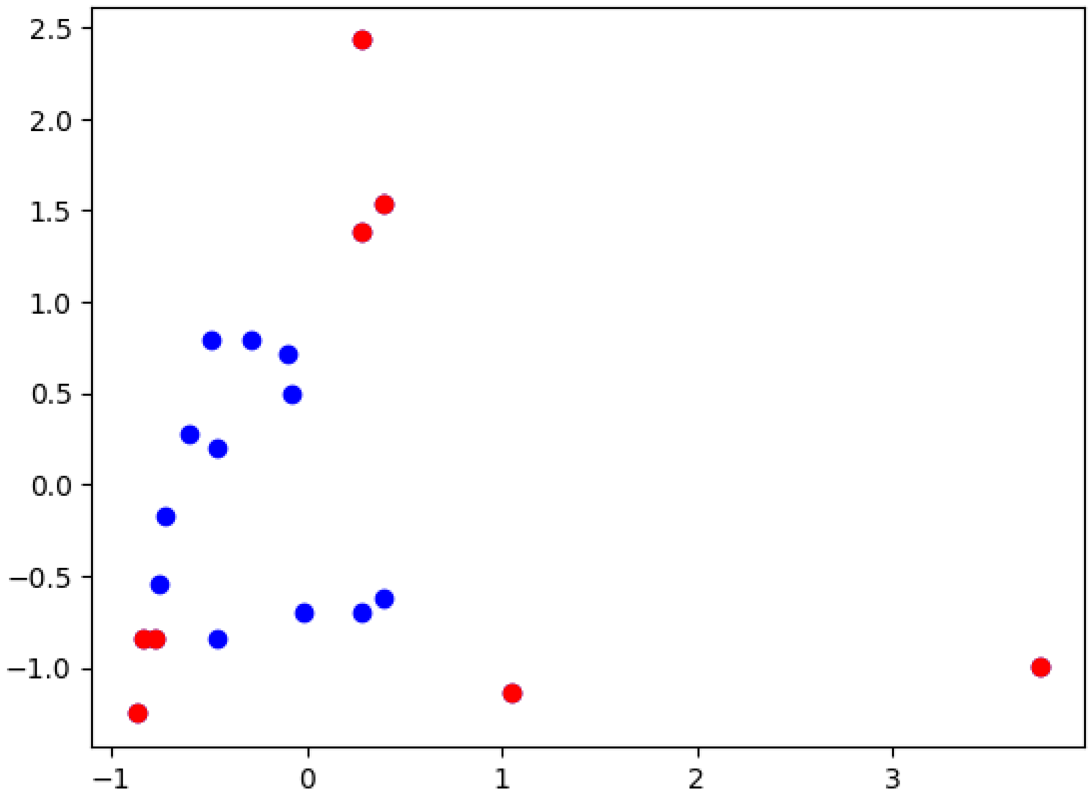

2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1
0
1
2
3


2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1
0
1
2
3

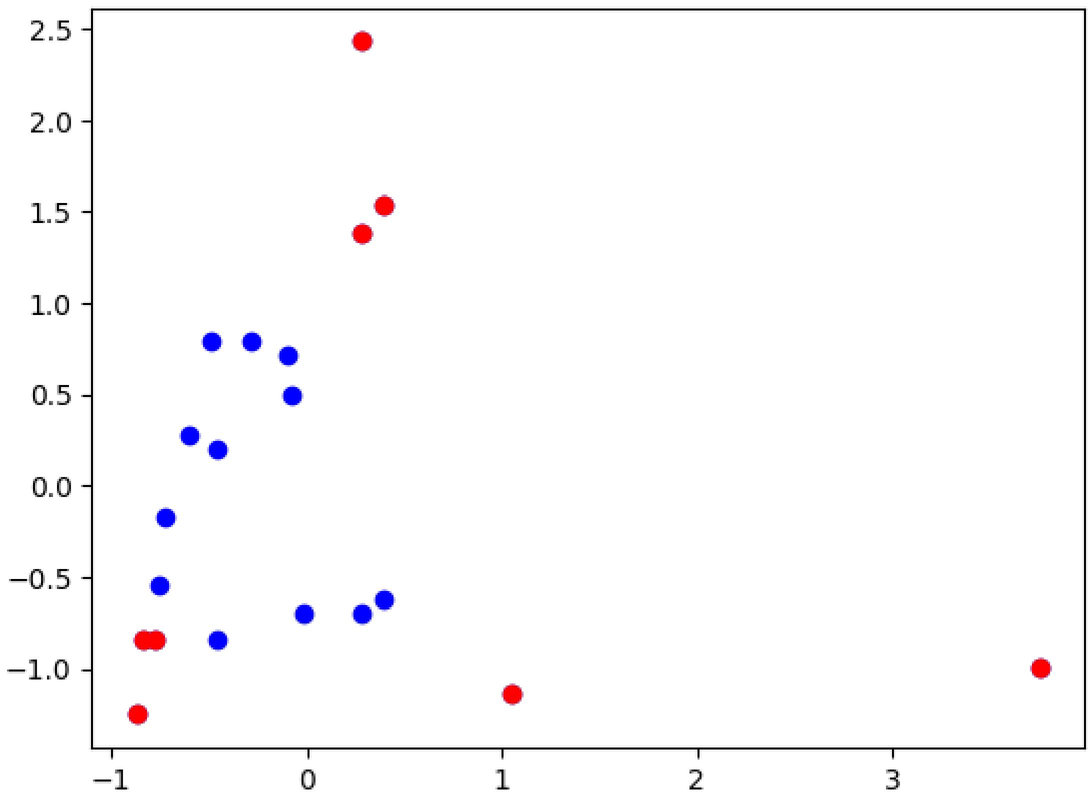

2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1
0
1
2
3



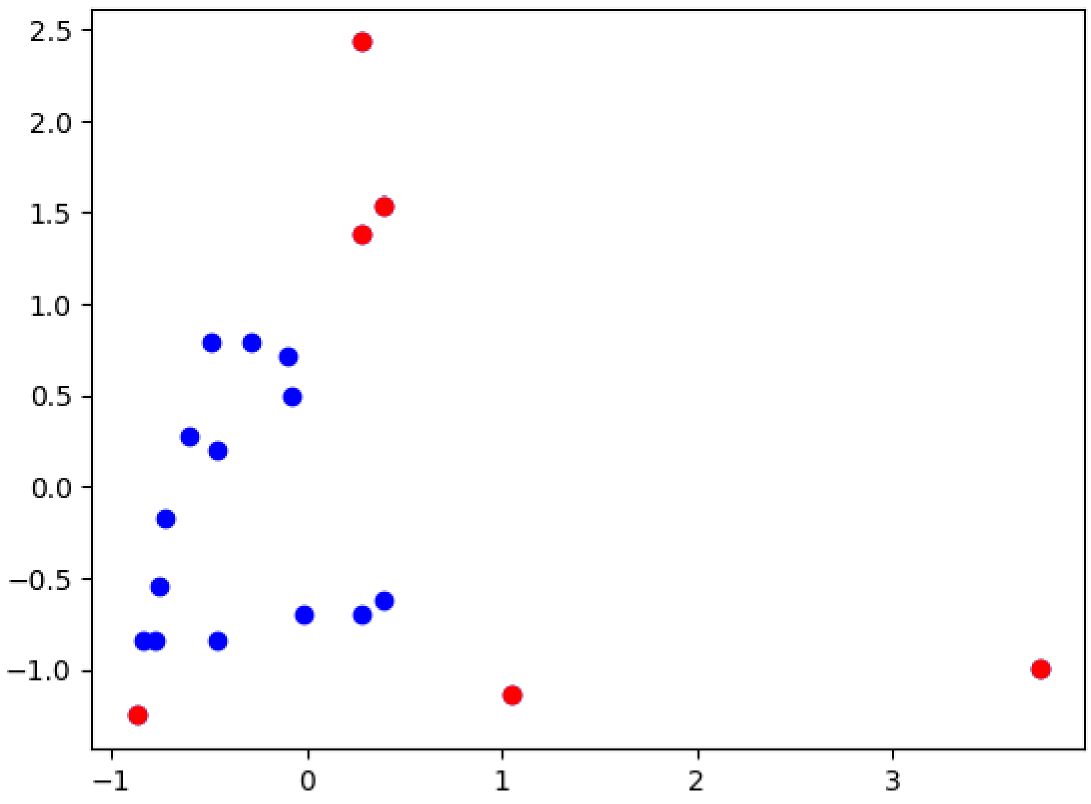
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1
0
1
2
3



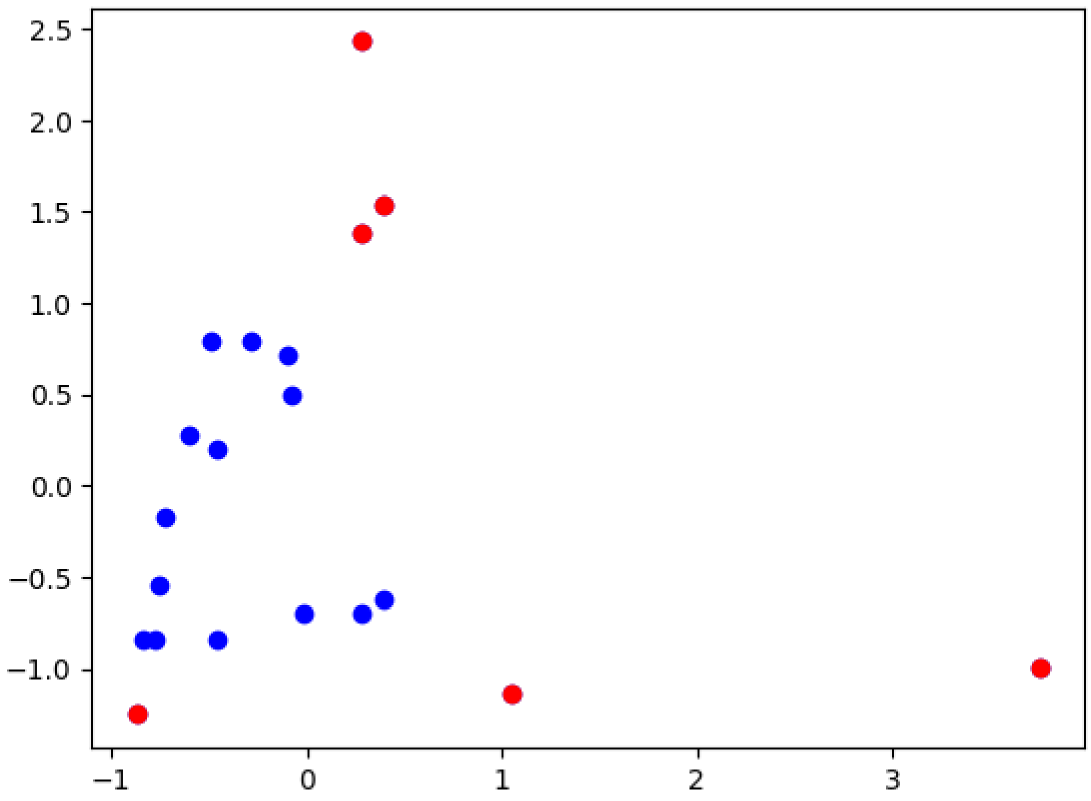
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1
0
1
2
3

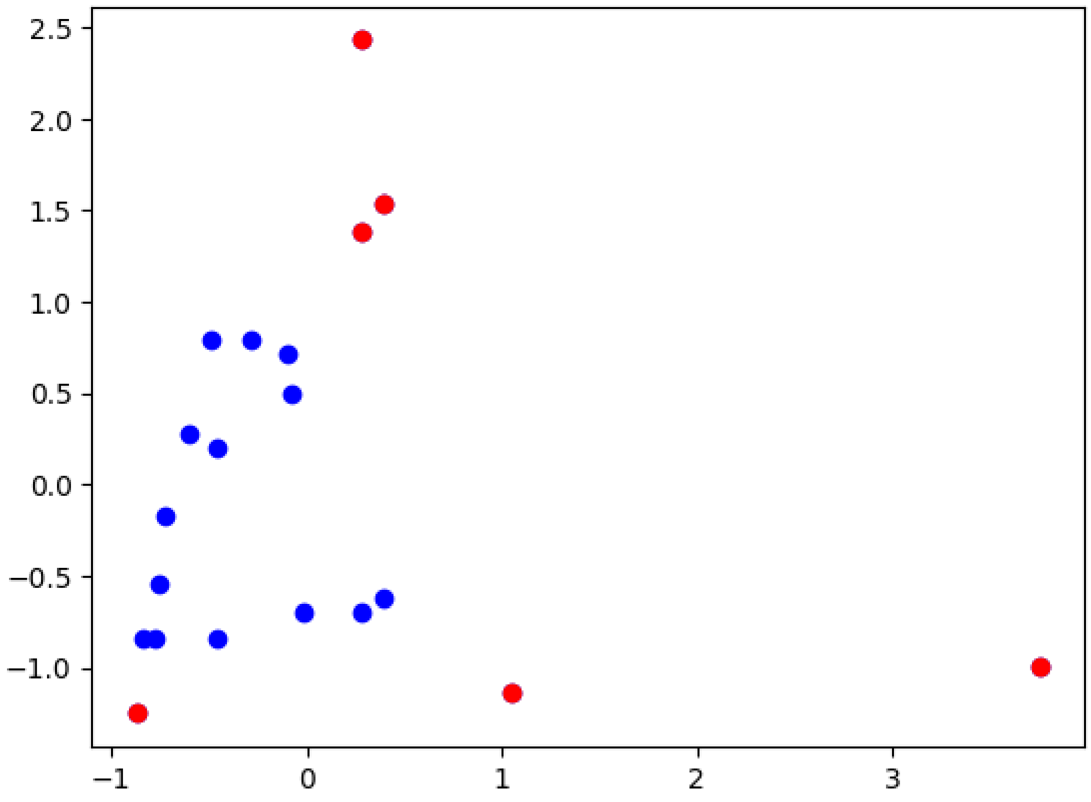


2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1
0
1
2
3



2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1
0
1
2
3





```
(1.5000000000000002, 2.0)
```

B) **[Describe]** your algorithm, its main idea, its strengths and its weaknesses

---

Answer

The function iteratively adjusts eps and pi based on the fraction of detected outliers, until the fraction is between 1% and 5% or the maximum number of iterations is reached. We adjust eps by a fixed amount of 0.1, and set pi based on the updated value of eps and the median of the data.

One of the strengths is that it is simple to implement, and can give us better optimal parameters. However, one of its weaknesses, is that it increases and decreases by a fixed value **0.1**, which could be too high or low, depending on the data.

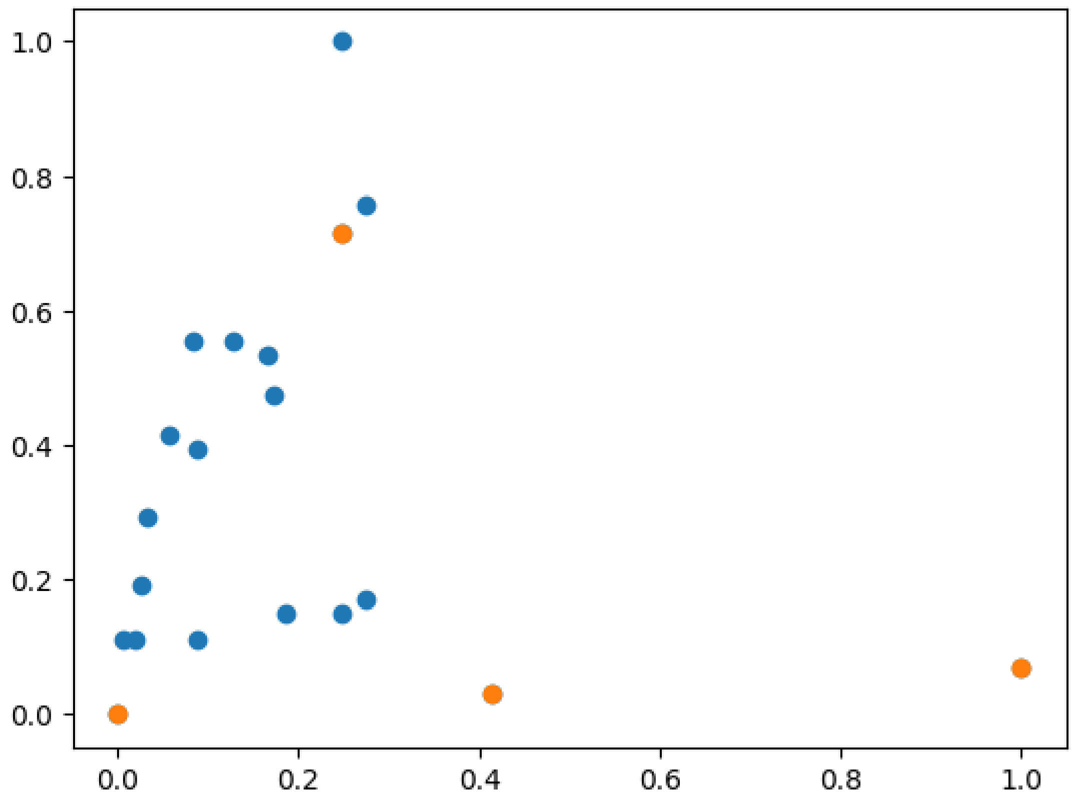
Task 4.2 LOF (2 points)

[Describe]
Using the parameters eps=0.18, pi=0.2 compare the results of DBOutliers with those obtained by LOF implemented in Week 9. What outliers do you find?

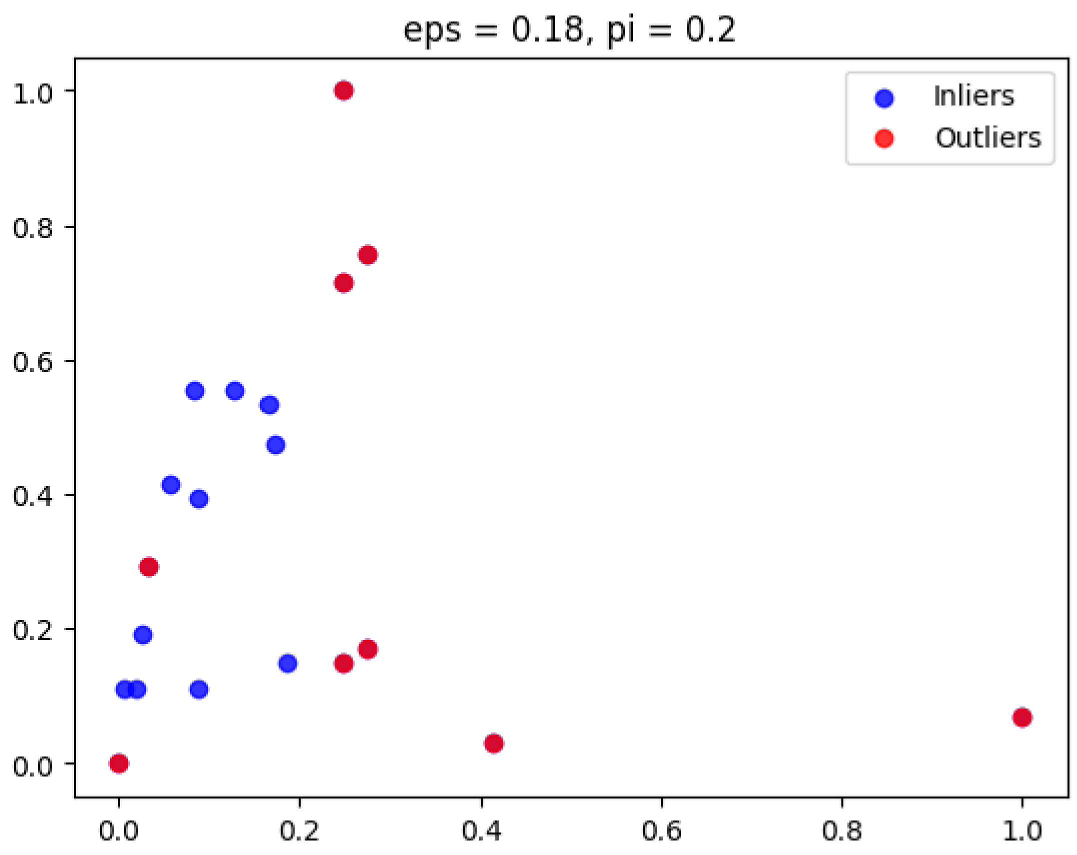
In [ ]:
```python
X = X_norm

neighborhoods  = []
kdists         = []

# Find neighbors and kdists
for i, x in enumerate(X):
    n = np.arange(X.shape[0])
    d = np.zeros_like(n, dtype=float)
    for j, x_ in enumerate(X):

        if i == j: continue # skip i it self
        d[j] = np.linalg.norm((x-x_), ord=1)
    args = np.argsort(d)
    ns = n[args]
    ds = d[args]

    kdist =  ds[k-1]
    #print(f'd:{d}\nns:{ns}\nds: {ds}\nkdist:{kdist}')
    kdists.append(kdist)
    neighborhoods.append(ns[ds <= kdist])

# Compute lrds
reach_dist = lambda i, j: max(kdists[j], np.linalg.norm(X[i]-X[j], ord=1))
lrds = []
for i, x in enumerate(X):
    lrd = 0
    for j in neighborhoods[i]:
        lrd += reach_dist(i, j)
    lrd /= len(neighborhoods[i])-1
    lrd = 1 / lrd
    lrds.append(lrd)

def LOF(i):
    lof = np.sum([lrds[j] for j in neighborhoods[i]])/len(neighborhoods[i])
    return lof / lrds[i]

lofs = [LOF(i)> 1.30 for i in range(len(X))]

plt.scatter(X[:,0],X[:,1])
plt.scatter(X[lofs,0],X[lofs,1])
plt.show()
```

```
for i in range(len(lofs)):
    if lofs[i]: print(LOF(i))
```

```
outliers = DBOutliers(X_norm, 0.18, 0.2)
plt.scatter(X_norm[:, 0], X_norm[:, 1], alpha=0.8, c='b')
plt.scatter(X_norm[outliers, 0], X_norm[outliers, 1], alpha=0.8, c='r')
plt.title('eps = {}, pi = {}'.format(0.18, 0.2))
plt.legend(['Inliers', 'Outliers'])
plt.show()
```



```
1.7187407214542385
3.9138608501845473
2.3518602885345525
1.4024024024024015
```



Answer

All the outliers found in LOF are also contained in the set of outliers found by DBOutlier, however DBOutlier finds more.
