CS245 Homework 2

July 31, 2024

Name: Kelly Furuya

1 Problem 1: KMeans

1.1 Predict the centroids of the clustesr and classify each point into its respective cluster.

Using the sklearn package in Python, we can calculate the centroids and label each data point using the code below that is based on the code provided in the sklearn documentation.

```
from sklearn.cluster import KMeans
import numpy as np

data_points = np.array([[9,5], [5,10], [10,11],
                        [4, 9], [14, 2], [4, 1],
                        [9, 9], [3, 12], [19, 6],
                        [17, 13], [13, 15]])

kmeans = KMeans(n_clusters = 2, init=([[10,13], [7,10]]), n_init = 'auto').fit(
                                        data_points)

kmeans.cluster_centers_
kmeans.labels_
```

We get [14.75, 11.25] and [6.85714286, 6.85714286] as the centroids. One cluster contains [9,5], [5, 10], [4, 9], [14, 2], [4, 1], [9, 9], and [3, 12] while the other cluster contains [10, 11], [19, 6], [17, 13], and [13, 15].

1.2 The KMeans algorithm aims to minimize the Within-Cluster Sum of Squares (WCSS). Explain the reasoning behind this objective function. What are the potential limitations or drawbacks of this approach?

Minimizing the WCSS means that the distance between each data point within a cluster is as small as possible. This means that the points grouped into a cluster are more likely to be related to those in the same cluster than the data points grouped into another cluster.

One of the potential drawbacks is that KMeans tends to be a poor choice for handling outliers. By their very nature, outliers will have a much larger distance from any given data point, greatly increasing the cluster's WCSS. The algorithm also struggles to deal with noisy data and very large datasets. The vast number of calculations needed at each iteration and the increasing number of

passes needed to noisy/large data makes it fairly resource intensive. It also has the drawback that it can not handle non-numeric data without it first being encoded into a numeric value in some way. Lasly, using WCSS means that the algorithm is not technically deterministic, so each initialization of the algorithm can produce different results depending on the initial centroid selection.

1.3 Investigate the impact of initial centroid selection in the KMeans algorithm

1.3.1 Analyze at least two common strategies, discussing their pros and cons

One of the most common centroid selection strategies is random selection. This requires the algorithm to randomly select k centroids as starting points. It is very simple and easy to understand and not resource intensive. However, it does have an issue where different starting centroids can produce poor results.

Another common approach is KMeans++. A single initial centroid is randomly selected. Subsequent initial centroids are then selected based on a probability that uses their distance from the randomly selected one. This is an improvement over the first approach as it aims to cover as much ground as possible with the starting centroids to produce better results. It does have a higher intialization cost, especially with larger datasets, as all the initial centroids other than one must be determined by running calculations. However, this up-front cost can lead to a reduced number of iterations and produce better clusterings.

1.3.2 Use of the two common strategies identified in part (a) on a dataset of your choice: (1) Perform KMeans on two dimensions of this dataset and show the outcome of each cluster center intialization, and (2) discuss the observed effect of cluster center initialization and what information the clustering provides.

By default, the KMeans function in the sklearn library uses the random method by default, so, to use the KMeans++ method, we just need to specify it. In this case, we will use a dataset created with one distinct cluster and an outlier in an attempt to specifically show how the two different initialization methods produce different results. The code used is shown below.

```
from sklearn.cluster import KMeans
import numpy as np

data_points2 = np.array([[0, 0], [1, 0], [-1, 0], [0, -1], [2, 2], [-1, -1], [1, -
                                        1], [0,1]])

kmeans_rand = KMeans(n_clusters = 3, init='random', n_init='auto').fit(
                                        data_points2)
kmeans_plus = KMeans(n_clusters = 3, init='k-means++', n_init='auto').fit(
                                        data_points2)

print("random centers ",kmeans_rand.cluster_centers_)
print(kmeans_rand.labels_)
print("plus centers ", kmeans_plus.cluster_centers_)
print(kmeans_plus.labels_)
```

The random initialization ended up with [1, 1] and [-0.2, -0.6] as the centroids, one cluster with [0, 0], [-1, 0], [0, -1], [-1, -1], and [1, -1], and the other cluster with [1, 0], [2, 2], and [0, 1]. The KMeans++ initialization ended up with [0, -0.286] and [2, 2] as the centroids, one cluster with [2, 2], and the rest of the data points in another cluster. The images below show the two different clusters for each method.

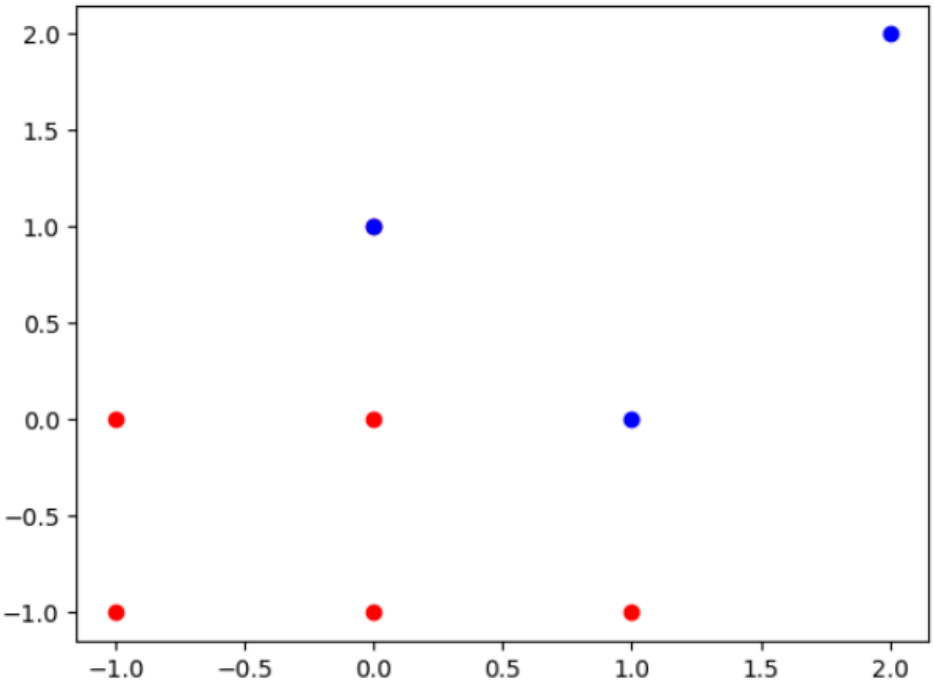

Figure 1: Clusters generated from randomly selected initial centroids

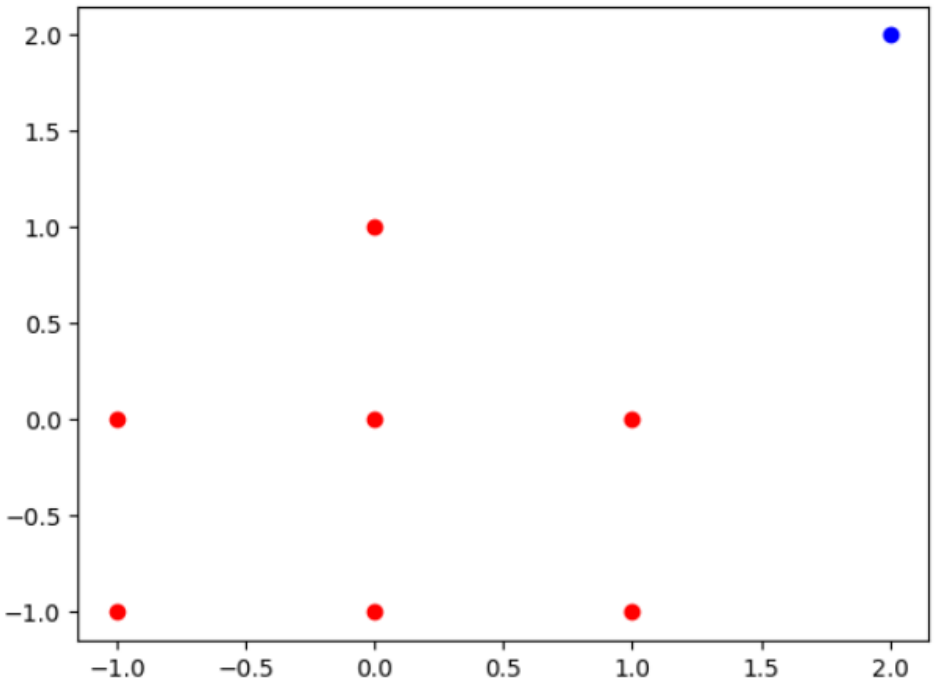

Figure 2: Clusters generated from initial centroids selected by the KMeans++ method

The figures more clearly show the different groupings created by each method in a very basic example. Although neither grouping is necessarilty better or worse than the other, given certain contexts, one grouping may be preferable over the other. The two methods can produce the same

results, especially for sparse data and a small k, but will still usually result in differnet answers. In this case, the random method tried to combine the outlier into a cluster while the KMeans++ method attempted to make it its own cluster. The random method's clustering indicates that it perceived the outlier as more related to the two other data points than those points were to the rest of the data set, while the KMeans++ method determined that the outlier was unrelated enough to the rest of the dataset to be its own cluster.

2 Problem 2: Hierarchical Clustering

Run agglomerative hierarchical clustering on the following 1D dataset: {10, 2, 8, 12, 13, 15}.

2.1 If we use single-linkage as the cluster distance, perform the hierarchical clustering step-by-step and show the resulting dendrogram.

The first step is to calculate the distance between each cluster or data point. For this dataset, we will define the distance as the difference between the two points. So, for example, the distance between 10 and 2 will be 8. Since we are using single-linkage, the distance between a cluster [10, 12] to the point 8, for example, will be the shortest distance between the individual data point and the closest data points in the cluster. The distance between two clusters will be the distance between the two closest data points from each cluster.

	10	**2**	**8**	**12**	**13**	**15**
10	0					
2	8	0				
8	2	6	0			
12	2	10	4	0		
13	3	11	5	1	0	
15	5	13	7	3	2	0

Figure 3: Step 1: Distance matrix showing the linear distance between each data point

We can see that the shortest distance between any two of the given data points is 1 between 12 and 13. These will be the first two objects we merge together to form a cluster. We combine them together in the distance matrix into one entry and recalculate the distance between the new cluster and all other data points.

	12,13	**10**	**2**	**8**	**15**
12,13	0				
10	2	0			
2	10	8	0		
8	4	2	6	0	
15	2	5	13	7	0

Figure 4: Step 2: Updated distance matrix with 12 and 13 merged

Next, we see that there are multiple shortest distances of 2. Since we can only merge them two at a time, we will start by combining the individual items first before adding to an existing cluster.

	12,13	**10,8**	**2**	**15**
12,13	0			
10,8	2	0		
2	10	6	0	
15	2	5	13	0

Figure 5: Step 3: Updated distance matrix with two clusters

Now, we see that we once again have mutliple shortest distance. We will combine the two existing clusters to form a larger cluster.

	12,13, 10,8	**2**	**15**
12,13	0		
2	6	0	
15	2	13	0

Figure 6: Step 3: Updated distance matrix with both clusters merged

Between 2 and 15, 15 is closer to the large cluster, so we will merge it.

	12,13,10,8,15	**2**
12,13,15,10,8	0	
2	6	0

Figure 7: Step 3: Updated distance matrix with15 merged into the large clusterr

Lastly, we are only left with 2 and the large cluster consisting of the rest of the data points. We can now merge 2 into this cluster and create the final dendogram.

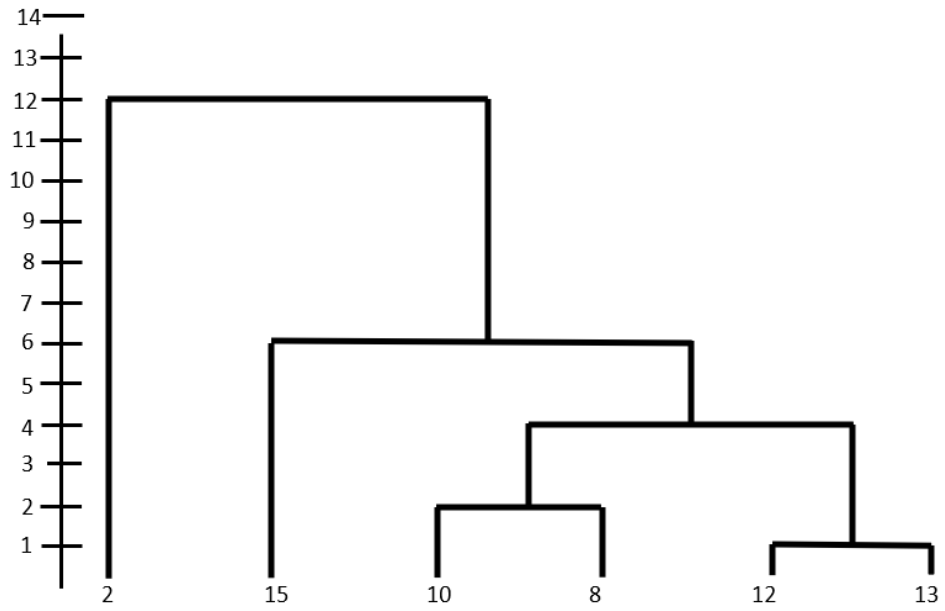

Figure 8: Step 6: Dendrogram showing the final merge order and the distance between each step.

2.2 What clusters will we get if we cut the dendrogram at a height of 3?

If we cut the dendrogram at a height of 3, based on the order we grouped the nodes, we would end up with one cluster containing 10 and 8, another cluster containing 12 and 13, and 2 and 15 still separate.

2.3 Discuss the differences between single-linkage and complete-linkage methods. How might the choice of linkage method impact the final clusters?

Single-linkage uses the shortest distance between the two closest clusters to determine what to group. This means it ends up producing better results for local clustering. Complete-linkage, on the other hand, uses the largest distance between the two closest clusters to determine grouping. This results in better global clustering instead.

The diagram below from the recommended textbook for this course, "Data Mining. Concepts and techniques" by Han, Kamber and Pei, Morgan Kaufmann, 2011, shows a good example of how the two different methods can result in different groupings.

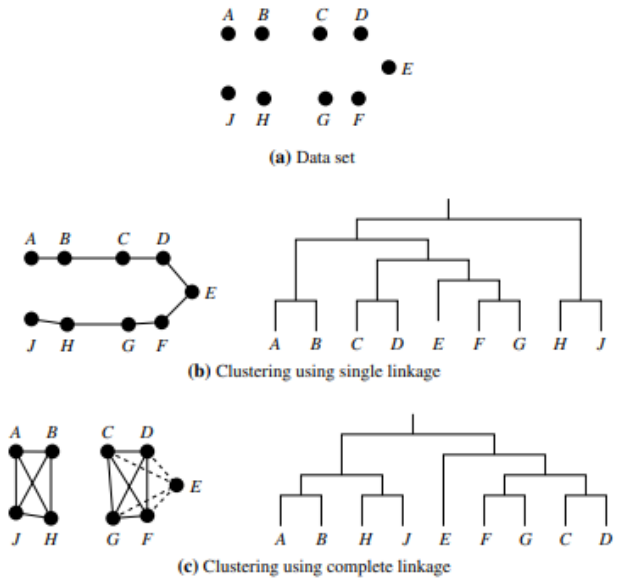

Figure 9: Two different dendrograms showing the difference between single and complete linkage

The example more clearly shows that single-linkage was better at creating a dendrogram that grouped data points together into local clusters based on what was immediately close. Complete-linkage created a dendrogram that split the data into two distinct clusters as oppposed to the multiple smaller clusters that single created.

3 Problem 3: DBSCAN

3.1 Apply the DBSCAN algorithm to the given dataset using Euclidean distance with the following parameters: $\epsilon = 1$ and MinPts $= 2$. Identify the core points, border points, and noise points in the final clustering result

The first step is to normalize the data before applying the DBSCAN algorithm. We then find the Euclidean distance between each data point to determine which, if any points, have 2 or more points in its neighborhood. The code below was used to accomplish this.

```
from sklearn.cluster import DBSCAN
from sklearn.preprocessing import StandardScaler

data3 = np.array([[10,-20], [5, 12],
                  [24, 22], [-2, -7],
                  [-3, -10], [12, -20],
                   [8, 17], [9, 11],
                    [18, 20], [8, 2],
                     [20, -10]])

data3 = StandardScaler().fit_transform(data3)


db_clust = DBSCAN(eps = 1, min_samples = 2).fit(data3)

from sklearn.metrics.pairwise import euclidean_distances
temp = euclidean_distances(data3, data3)
print(temp)
```

The matrix below shows these distances using the original data points as labels for simplicity.

	[10, -20]	[5, 12]	[24, 22]	[-2, -7]	[-3, -10]	[12, -20]	[8, 17]	[9, 11]	[18, 20]	[8, 2]	[20, -10]
[10, -20]	0.	2.22641153	3.30233446	1.72695531	1.74964524	0.2487822	2.48428991	.07470138	2.85149624	1.49062715	1.41195305
[5, 12]	2.22641153	0.	2.45603389	1.53925818	1.7749248	2.30830323	0.50083183	0.50202918	1.70311233	0.76521576	2.37519147
[24, 22]	3.30233446	2.45603389	0.	3.77004018	3.98120622	3.17817704	2.01809304	2.00536206	0.75821179	2.39714678	2.19491458
[-2, -7]	1.72695531	1.53925818	3.77004018	0.	0.23588103	1.94601599	2.02928174	1.8216071	3.07290779	1.38159872	2.74393318
[-3, -10]	1.74964524	1.7749248	3.98120622	0.23588103	0.	1.98185636	2.26401295	2.04848775	3.29246547	1.58584933	2.8609953
[12, -20]	0.2487822	2.30830323	3.17817704	1.94601599	1.98185636	0.	2.52138337	2.10432197	2.77448783	1.55165945	1.19857339
[8, 17]	2.48428991	0.50083183	2.01809304	2.02928174	2.26401295	2.52138337	0.	0.41969037	1.25995289	1.00208178	2.3412896
[9, 11]	2.07470138	0.50202918	2.00536206	1.8216071	2.04848775	2.10432197	0.41969037	0.	1.27075775	0.61398175	1.95969888
[18, 20]	2.85149624	1.70311233	0.75821179	3.07290779	3.29246547	2.77448783	1.25995289	1.27075775	0.	1.73012033	2.01954553
[8, 2]	1.49062715	0.76521576	2.39714678	1.38159872	1.58584933	1.55165945	1.00208178	0.61398175	1.73012033	0.	1.69434366
[20, -10]	1.41195305	2.37519147	2.19491458	2.74393318	2.8609953	1.19857339	2.3412896	1.95969888	2.01954553	1.69434366	0.

Figure 10: Matrix showing the Euclidean distance between every data point.

From this matrix, we can see that every data point has a distance less than 1 between at least one other data point except for [20, -10]. Since MinPts is 2, if any data point has another point in its neighborhood, then by default, it will be a core point since the data point itself is counted as in its own neighborhood. This means that the core points are: [10, -20], [5, 12], [24, 22], [-2, -7], [-3, -10], [12, -20], [8, 17], [9, 11], [18, 20], [8, 2], and the only noise point is [20, -10]

3.2 Discuss the effect of changing the ϵ value in DBSCAN. What happens when its value is increased or decreased? How does it affect the resulting clusters and noise points?

Increasing ϵ increases the radius of the neighborhood, increasing the odds that a data's points neighborhood might contain more neighbors. This can then lead to an increase in the number of core points and a decrease in the number of noise points. Conversely, decreasing ϵ decreases the radius of a data point's neighborhood, thus potentially decreasing the number of core points and an increase in the number of noise points.

For example, if we increase ϵ to 2 for the above data, [20, -10] would no longer be a noise point and would instead be considered a core point. If we decrease ϵ to 0.5, then [5,12], [24, 22], [18, 20], and [8, 2] would no longer be core points and would instead be considered noise points.

3.3 Discuss the advantages and limitations of DBSCAN as compared to KMeans and hierarchical clustering.

In terms of calculations and resource usage, DBSCAN only requires the distance calculations to be run a single time as opposed to KMeans which runs the calculations at every iteration. Hierarchical clustering only runs the calculations once as well but requires the algorithm to compare values and save either the smaller or larger one for each cluster at each iteration.

DBSCAN is also deterministic meaning it will obtain the same result each time as long as the parameters remain the same. KMeans is not deterministic and can give different results each time based on what centroids are used initially. Hiearchical is also deterministic.

KMeans struggles with outliers while DBSCAN and hierarchical clustering are able to manage and label them fairly well.

While KMeans can handle sparse data and still create decent clusters, DBSCAN can struggle and requires much more fine tuning of the variables to get good results.

3.4 Apply the Local Outlier Factor algorithm for outlier detection to the same dataset. Discuss the differences you observe between the two methods in terms of identifying outliers. How do the parameter settings of LOF (e.g. the number of neighbors) affect the points that are identified as outliers?

In order to calculate the local outlier factors for each data point, we again use the sklearn library, making use of the LocalOutlierFactor function.

```
from sklearn.neighbors import LocalOutlierFactor

clf = LocalOutlierFactor(n_neighbors = 2)
y_pred = clf.fit_predict(data3)
x_scores = clf.negative_outlier_factor_

plt.scatter(data3[:, 0], data3[:, 1], color = "k", s= 3.0)
radius = (x_scores.max() - x_scores) / (x_scores.max() - x_scores.min())
scatter = plt.scatter(data3[:, 0], data3[:, 1], s=1000 * radius, edgecolors = "r",
                                                facecolors="None")
plt.axis("tight")
plt.xlim((-2, 2))
plt.ylim((-2, 2))
plt.show()

print(x_scores)
print(y_pred)
```

The above code produces the following graph.

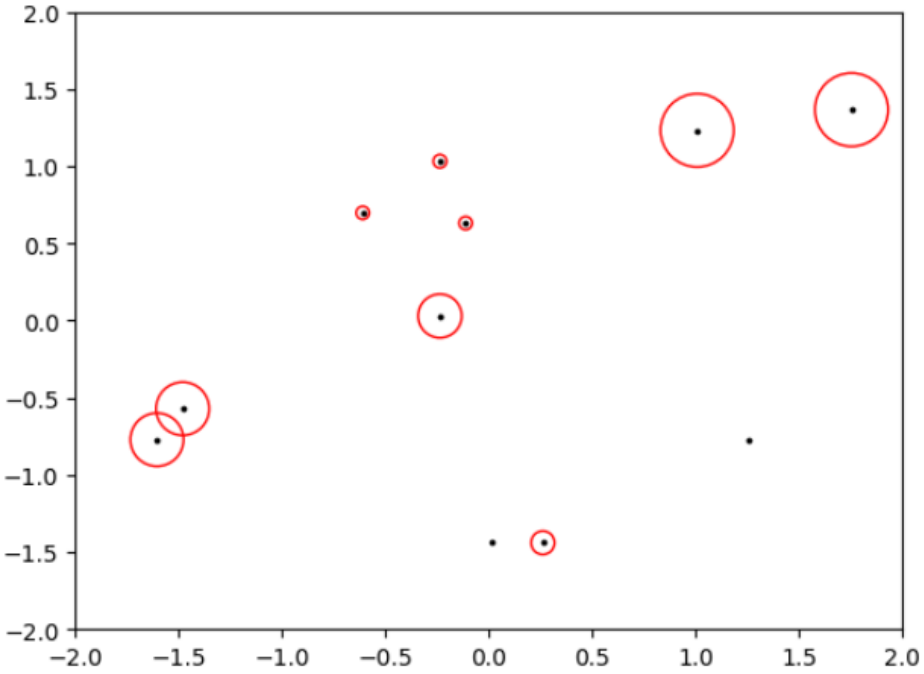

Figure 11: Graph of each data point along with a circle indicating the size of its outlier factor

The score for each data point is shown below.

Observation	**Data Point**	**Outlier Factor**
1	(10, -20)	0.96221906
2	(5, 12)	0.99940374
3	(24, 22)	2.12799974
4	(-2, -7)	1.57578793
5	(-3, -10)	1.57578793
6	(12, -20)	1.08173817
7	(8, 17)	1.00119394
8	(9, 11)	0.99940374
9	(18, 20)	2.12605833
10	(8, 2)	1.37526286
11	(20, -10)	0.96221906

Table 1: Outlier score for each data point

If a point has a score close to 1, it means that is is part of a cluster, but if it has a value greater than $1 + \epsilon$, then it is an outlier. With $\epsilon = 1$, we can see that [24, 22] and [20, -10] are considered outliers in this case.

DBSCAN only found one outlier, [20, -10], whereas LOF found two. If we go back and re-examine Figure 10, we can see that [24, 22] does have one of the larger distances from the nearest neighbor, indicating that it is a bit farther away than other core points. However, based on the way the parameters are tuned, LOF identified [24, 22] as an outlier but DBSCAN did not.

If we increase MinPts, LOF will require more points to be within the neighborhood of a data

point for it to be considered a core point. Decreasing it will do the opposite. In this case, reducing MinPts to 1 would mean that every single data point is a core point since it is automatically counted as part of its own neighborhood.

Increasing ϵ will increase the threshold for a point to be considered an inlier for LOF. Intuitively, decreasing ϵ will do the opposite. For example, if ϵ is increased to 2, then no outliers will be found as all of the data points have an outlier factor less than $1 + \epsilon$. If we decrease it to 0.5, then [-2, -7] and [-3, -10] would then be considered outliers as well.