TUL221 – Mini Project

Pan Eyal, Ilana Pervoi

Introduction

For our mini project in the subject of “Unsupervised Learning”, we chose to Implement 3 well-known clustering algorithms, test them, and compare their results to each other as well as to existing publicly available implementations of those.

(All our algorithms code will be added along as a ZIP file, and will be attached in the end of this PDF)

The 3 clustering algorithms that we chose are:

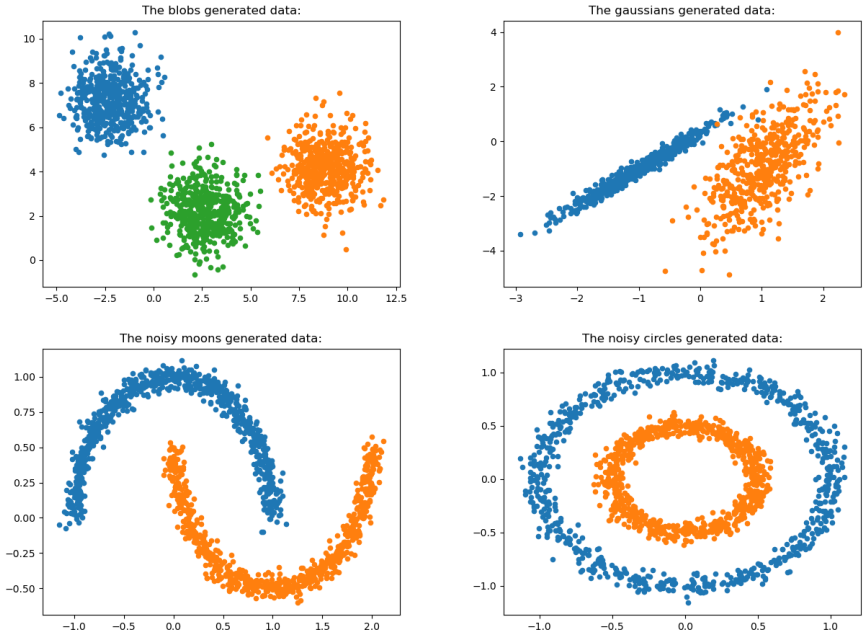
- DBSCAN
- Hierarchical clustering
- Affinity Propagation

We will start by explaining each algorithm and compare our results to sklearns’ algorithms results. After reviewing each of them, we will compare them to each other and conclude their positive and negative characteristics on different data inputs.

(Notice that the coloring assigned to each cluster might be reversed between our implementation and sklearns’ one.)

Each algorithm will be tested on 4 datasets from 2D Euclidean space that can be represented as a scatter plot on screen. At the end, each sample-point will be assigned a color according to its assigned cluster.

The 4 datasets with their idealized clusters are:



DBSCAN (Density-Based Spatial Clustering of Applications with Noise)

DBSCAN algorithm receives as input 2 parameters: ‘epsilon’ and ‘min neighbors’.

DBSCAN will iterate over the sample-points and will follow the next logic:
If the current sample-point didn’t allocate to a cluster and the datapoint has a desirable density (meaning that a circle with radius equals to ‘epsilon’ around the current data-point contains satisfactory amount of neighbors), it forms a cluster and expands to its neighbors. Then, in a BFS manner: if the neighbors fulfill the density demand, they will expand the cluster themselves.

In this algorithm plotting, we chose outlier sample-points to be plotted as black dots.

The received classification of our DBSCAN implementation compared to sklearn:

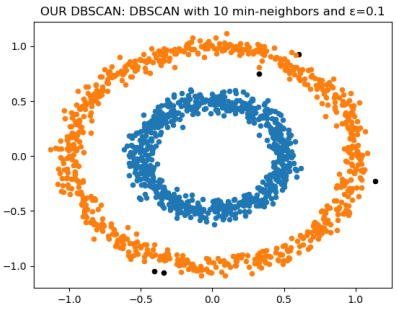


Figure 1.1

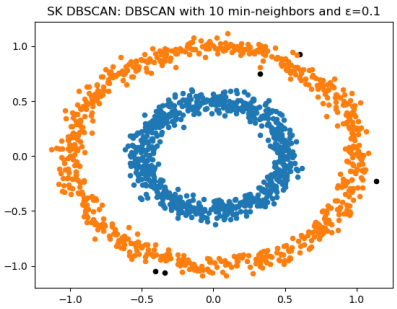


Figure 1.2

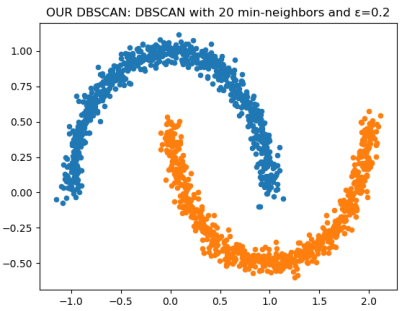


Figure 1.3

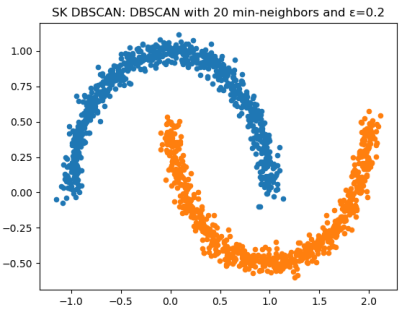


Figure 1.4

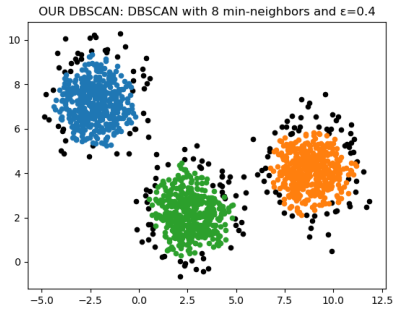


Figure 1.5

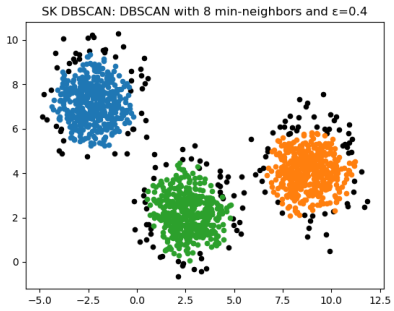


Figure 1.6

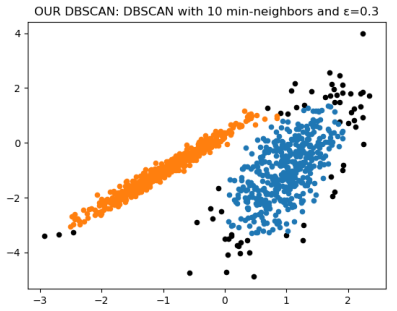


Figure 1.7

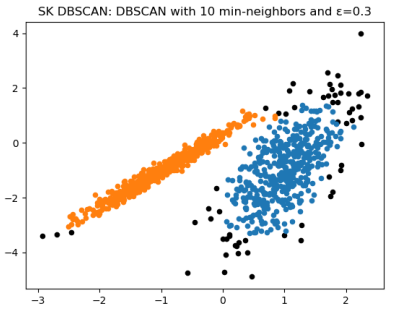


Figure 1.8

As we can observe, our implementation and sklearns' implementations managed to find the same number of clusters as generated. This is a nice attribute of DBSCAN as it does not relay on receiving the number of clusters as input.

However, the main drawback of DBSCAN is reflected in figures 1.5-1.8. In those figures, a lot of sample points did not get assigned to a cluster. This can be explained because the clusters are more spread across its edges, thus the density of the sample-point is lower, which is a deciding factor of DBSCAN for clustering.

Furthermore, DBSCAN requires tuning regarding the input parameters (epsilon and min neighbors) so that the clustering will yield meaningful insights. When epsilon is too large, clusters will merge, whereas if the value is too small, we will get a lot of outliers. In contrast, with too large min neighbors, the desired density may not be reached, and clusters will not form, whereas for too small min neighbors, we permit very low density, hence every point may form a cluster.

Hierarchical clustering

Hierarchical clustering algorithm receives as input the desired number of clusters: ‘k’.

In our implementation we chose to implement ‘single’ and ‘complete’ linkages with agglomerative approach. In ‘single’ linkage, at every iteration, the closest clusters will be chosen by comparing the closest sample-points between each two clusters. In ‘complete’ linkage, at every iteration, the closest clusters will be chosen by comparing the farthest sample-points between each two clusters. The algorithm will merge clusters respectively according to the logic above, until ‘k’ clusters are reached.

In both linkage options, at the end of the algorithm, the formed remaining clusters are said to be distinct from each other, whereas the points within it are similar to each other.

The received classification of our Hierarchical Clustering implementation compared to sklearn:

‘Single’ Linkage

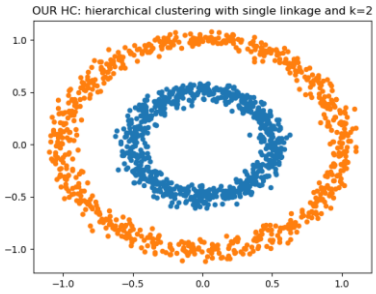


Figure 2.1

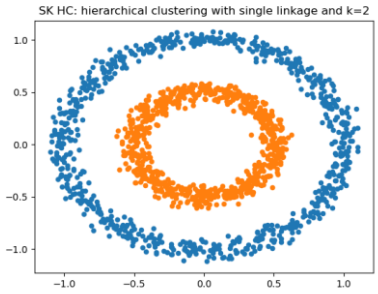


Figure 2.2

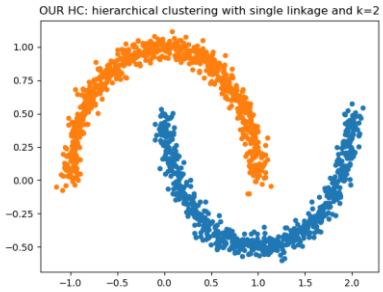


Figure 2.5

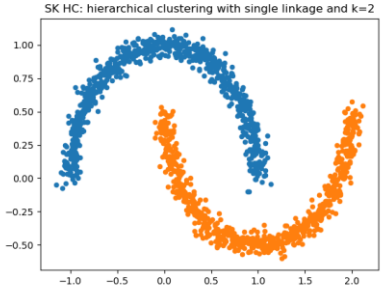


Figure 2.6

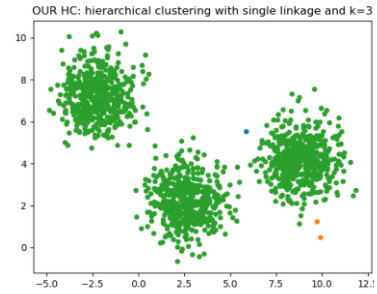


Figure 2.9

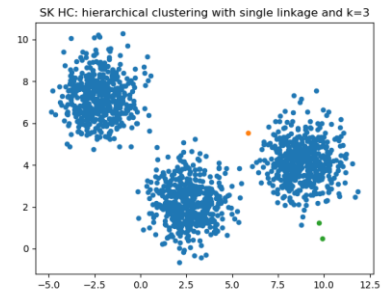


Figure 2.10

‘Complete’ Linkage

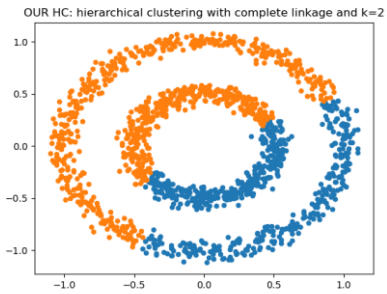


Figure 2.3

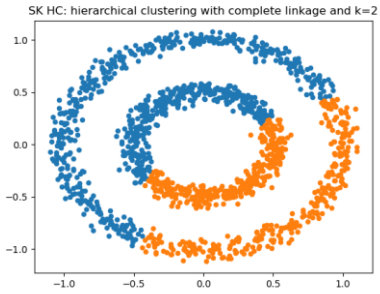


Figure 2.4

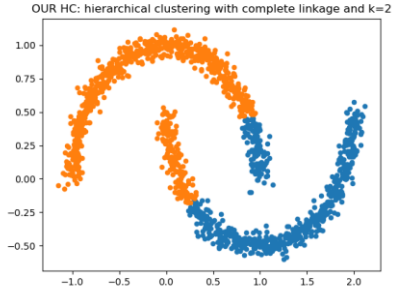


Figure 2.7

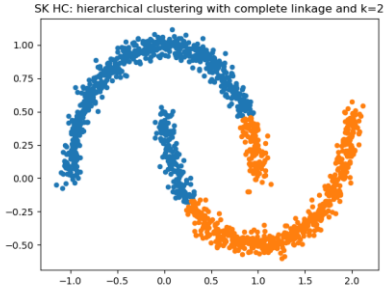


Figure 2.8

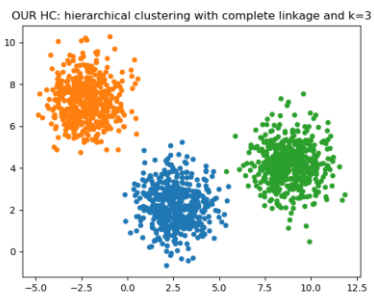


Figure 2.11

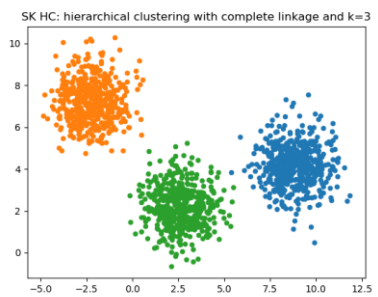


Figure 2.12

‘Single’ Linkage

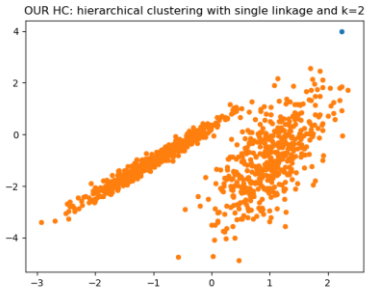


Figure 2.13

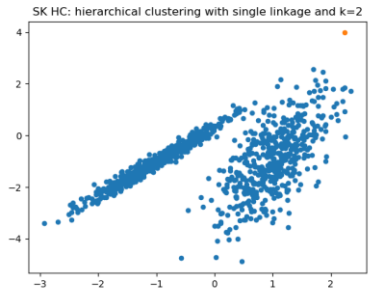


Figure 2.14

‘Complete’ Linkage

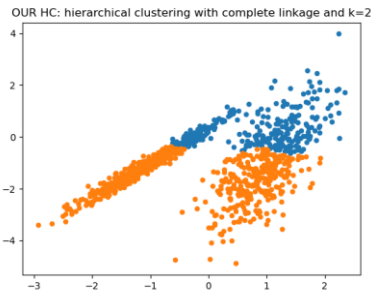


Figure 2.15

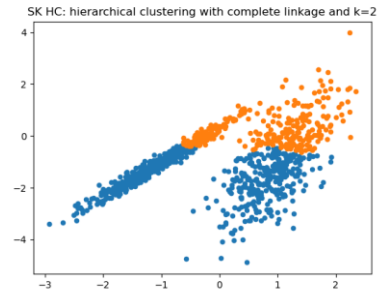


Figure 2.16

Firstly, our implementations and sklearns’ ones, received the same results. Interestingly, for different dataset we obtained drastically different results, meaning, some datasets received the desired clustering, and some datasets did not.

For figures 2.1-2.8, the single linkage worked perfectly while the complete linkage did not. However, in figures 2.9-2.12, the opposite happened. Sadly, figures 2.13-16 failed miserably on both ‘single’ and ‘complete’ linkage.

In “noisy circles” and “noisy moons” datasets, the shapes are mostly dense, continuous, and unevenly stretched on the plane. Therefore, in figures 2.1-2.8, ‘single’ linkage that looks for the minimal distanced between the merged clusters can maintain the continuity of the shape while merging. The ‘complete’ linkage on the other hand, fails. It tries to minimize the distance between the two farthest sample-points of each cluster, in contrary to the uneven stretch of the shape. This results in absorption from one cluster to the other.

For “blobs” dataset, the three generated clusters are quite close to each other, and even some sample-points overlap. Furthermore, the distance between the closest sample-points of some two clusters is closer than the distance between some two sample-points on the edge of the more spread (right) cluster. Therefore, in figures 2.9-2.12, the ‘single’ linkage will fail as the three clusters will merge, and some relatively distant parted sample-points will remain unmerged. However, the ‘complete’ linkage will perform better. That is because each blob is globular and evenly spread, as the furthest points in each blob are the same (the circle diameter).

Gaussians with different variance, like in our “gaussians” dataset, are non-globular and unevenly spread. We saw that in “noisy circles” and “noisy moons” datasets from figures 2.1-2.8, this data attribute cause problems to the ‘complete’ linkage method. Moreover, the distance between the closest sample-points of the two clusters, is closer than the distance between some two sample-points on the edge of the more spread (right) cluster, just as in the “blobs” from figures 2.9-2.12. That was a hurdle for the ‘single’ linkage. Therefore, we could expect that the “gaussians” in figures 2.13-2.16 will cause both ‘single’ and ‘complete’ linkages to fail as it did.

Affinity Propagation

Affinity Propagation algorithm receives as input: 'preference', 'max iteration', 'convergence iteration' and 'damping'.

In this algorithm, in every iteration two matrices being updated. The first is the responsibility matrix 'R' and the second is the availability matrix 'A'. When $R(i,j)$ represent the responsibility that sample 'j' is the exemplar for sample 'i', and $A(i,j)$ represent how certain sample 'i' to choose 'j' as its exemplar. The 'preference' argument will affect the amount of influence each sample-point will have on the other sample-points to choose itself as their exemplar. The Algorithm will run until 'max iteration' has been reached, or that the number of exemplars did not change for 'convergence iteration'. In each iteration, exemplars are chosen by popularity and adequacy.

In our plotting, the chosen exemplars plotted as larger dots with black edges around them. In all runs, 'convergence iteration' is set to be 25 and in the title of each figure, if preference is set to "None" it means the priori preference will be the median of the similarities of all sample-points.

The received classification of our Hierarchical Clustering implementation compared to sklearn:

Fitted 'Preference'

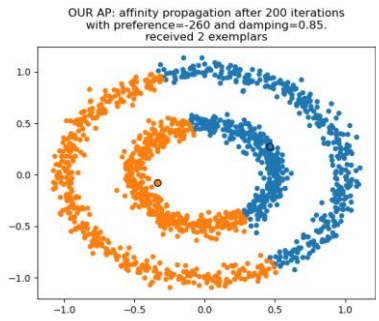


Figure 3.1

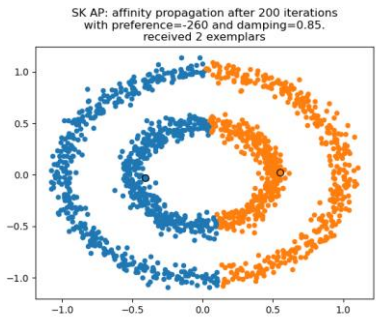


Figure 3.2

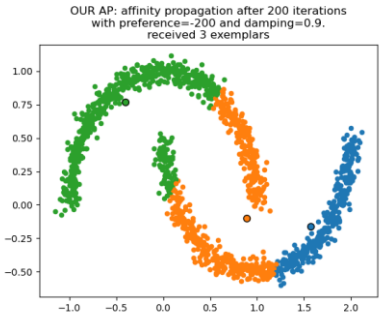


Figure 3.5

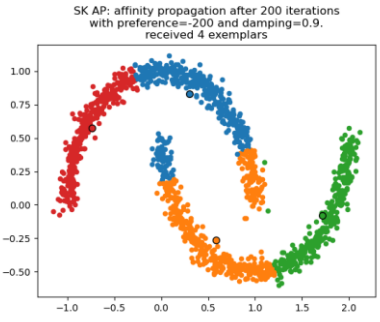


Figure 3.6

Mean 'Preference'

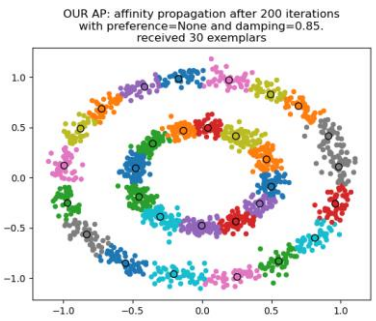


Figure 3.3

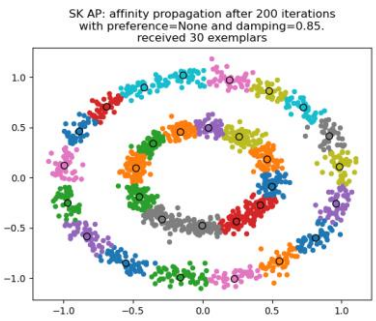


Figure 3.4

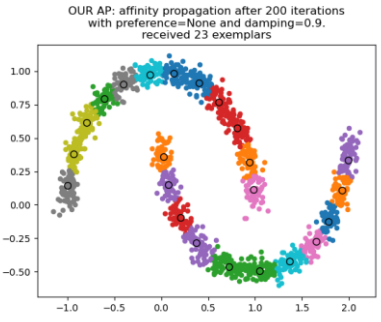


Figure 3.7

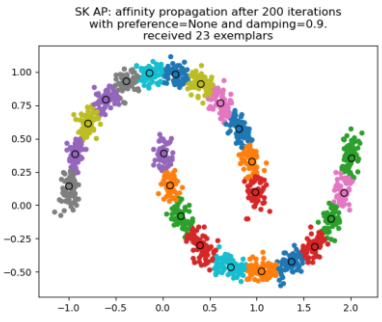


Figure 3.8

Fitted 'Preference'

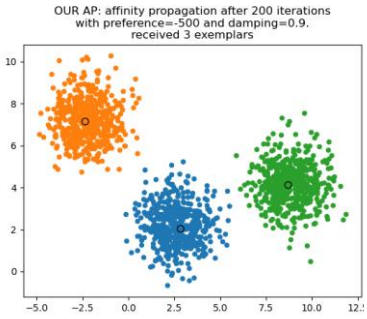


Figure 3.9

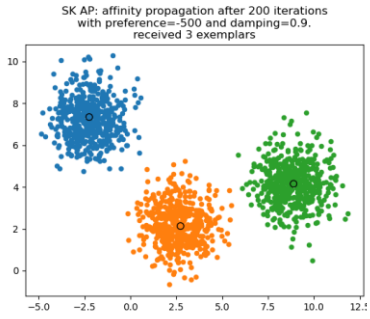


Figure 3.10

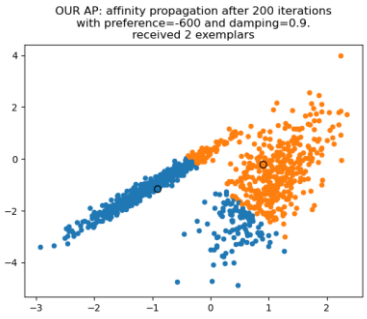


Figure 3.13

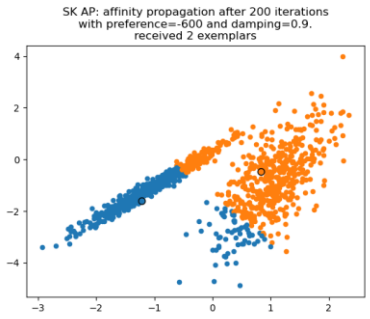


Figure 3.14

Mean 'Preference'

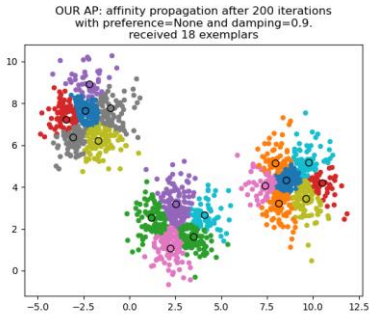


Figure 3.11

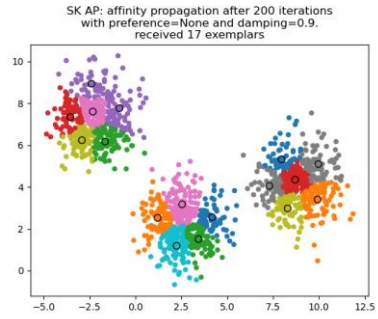


Figure 3.12

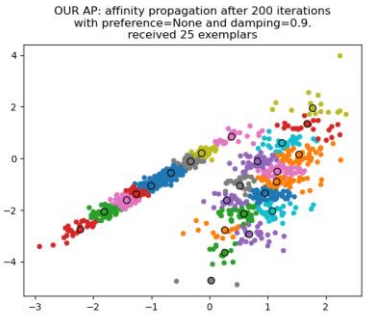


Figure 3.15

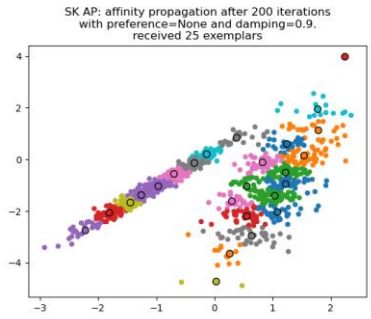


Figure 3.16

As we can observe, the only difference between our implementation and sklearns' one can be shown on the "noisy moon" datasets. We received 3 clusters and sklearns' one received 4. Moreover, Affinity Propagation algorithm only worked well on the "blobs" dataset.

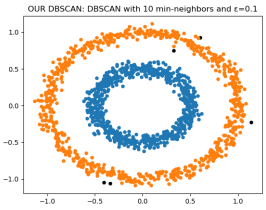
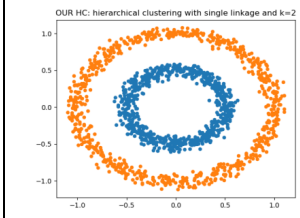
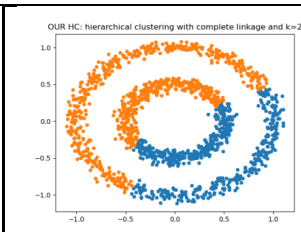
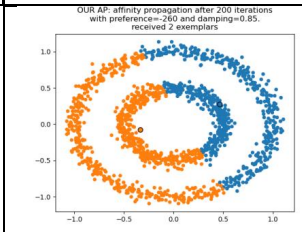
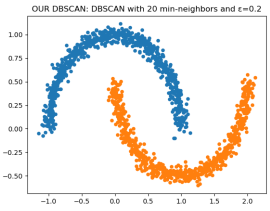
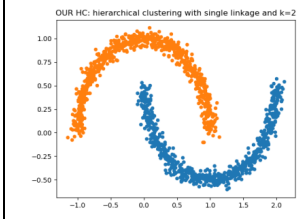
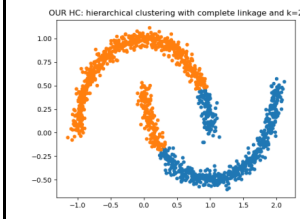
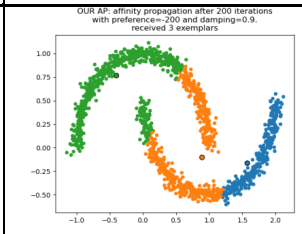
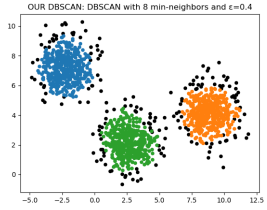
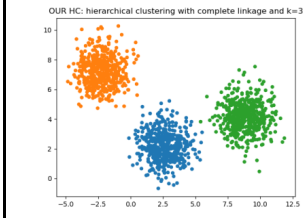
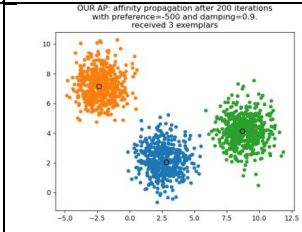
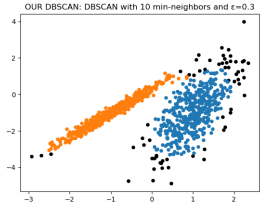
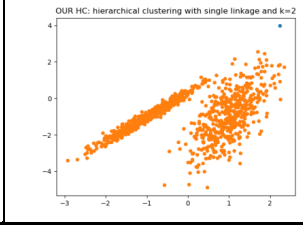
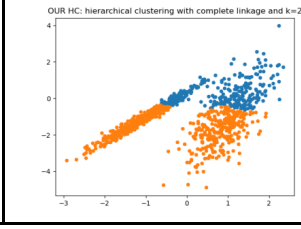
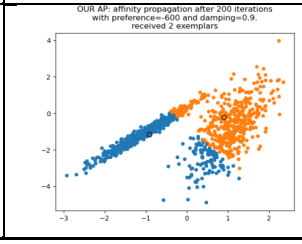
There are two main drawbacks we encountered and are reflected in the figures:

One is that Affinity Propagation is sensitive to the preference and damping parameters. In every data set we compared the results between the median preference and more inflated number and received completely different clustering. The reason behind it is that the preference effects the likelihood of one point to choose itself as an exemplar. Meaning, when we chose the median, the points have equal estimates whether they should be exemplars. As we can see from every figure with preference equals to median. On the other hand, when choosing a small preference, less points are compatible to be an exemplar and therefore, less clusters will emerge.

The second drawback is that even with stable parameters, good classification is not guaranteed. as we can see in figures 3.1-3.2, 3.13-3.14 the algorithm did find 2 clusters, however it is the wrong grouping in comparison to the real generated datasets.

Despite having major drawbacks, Affinity Propagation algorithm does not have to receive a specific number of clusters in advance.

DBSCAN vs. Hierarchical clustering vs. Affinity Propagation

DBSCAN	Hierarchical clustering		Affinity Propagation
	Single linkage	Complete linkage	
			
			
			
			

Overall, DBSCAN has good performance with high accuracy on any of the given datasets as long as it receives the right parameters. Its' main drawback that arises from the comparison, is that some sample points never got assigned to any cluster. The other algorithms don't suffer from the same issue, but they have some lack of clarity when they try to cluster on certain datasets.

As a general conclusion for mostly dense, continuous, non-globular data shape like "noisy circles" and "noisy moons" DBSCAN and Single linkage Hierarchical Clustering will manage to receive good results.

For globular data shapes as "blobs", Complete linkage Hierarchical Clustering and Affinity Propagation has the best performance.

And lastly, for non-globular data shape, that may have long distance between two datapoints on the edge of one of the clusters, compared to the distance between the clusters themselves (such as the "gaussians" dataset), DBSCAN is the only one that performed well.

If the number of the desired clusters is unknown, DBSCAN and Affinity Propagation would still be able to find the right clusters, as long as the right input parameters are provided. DBSCAN will need 'epsilon' and 'min neighbors' parameters to specify the 'density', and Affinity Propagation will need the 'preference' and 'damping' that will affect the amount of returned clusters. All those parameters need to be fine-tuned, with some priori knowledge about the dataset or with trial and error.

However, if the number of the desired clusters is known, the Hierarchical Clustering algorithm wouldn't need any more information to run. It can merge clusters based on distance until only one cluster is remaining. Based on the desired cluster number, it knows when to stop and return the relevant clusters.

In term of run-time, where N is the dataset size:

DBSCAN runs in $O(N^2)$, as we iterate over all the sample points in the dataset possibly multiple times, check for its' neighbors among all the points and visit them as well. Hierarchical Clustering runs in $O(N^2)$, as we calculate pairwise distance. Then the cluster merging is performed in $O(N)$.

Affinity propagation runs in $O(N^2T)$ (where T is the number of iterations until convergence), as calculating the desired responsibility and availability matrices is needed in each iteration. Therefore, DBSCAN and Hierarchical Clustering are preferable for conserving time.

In term of predicting new data based on prior fitting, Affinity Propagation is the only algorithm that requires $O(|Clusters|)$ times to fit the new data, as we already have the chosen exemplars (an exemplar per cluster), and we only need to determine the closest one to the new point. For all other algorithms, a comparison between the new sample point and the whole dataset points is needed.

Our DBSCAN implementation:

```
import numpy as np
import queue

class DBSCAN:
    def __init__(self, samples, epsilon, min_neighbours):
        self.samples = samples
        self.n = len(samples)
        self.epsilon = epsilon
        self.min_neighbours = min_neighbours
        self.samples_labels = np.full(self.n, -1)
        self.neighbours_lists = []

    # Find all neighbour points at epsilon distance
    def find_neighbour_points(self, i):
        neighbour_points = []
        center = self.samples[i]
        for j in range(self.n):
            if np.linalg.norm(center - self.samples[j]) <= self.epsilon and j != i:
                neighbour_points.append(j)
        return neighbour_points

    def set_neighbours_lists(self):
        for i in range(self.n):
            self.neighbours_lists.append(self.find_neighbour_points(i))

    def update_clusters(self, i):
        q = queue.Queue()
        visited = self.samples_labels[i] != -1
        is_core = len(self.neighbours_lists[i]) >= self.min_neighbours
        if is_core and not visited:
            self.samples_labels[i] = i
            q.put(i)

        while not q.empty():
            current = q.get()

            is_core = len(self.neighbours_lists[current]) >= self.min_neighbours
            if is_core:
                for neighbour in self.neighbours_lists[current]:
                    visited = self.samples_labels[neighbour] != -1
                    if not visited:
                        self.samples_labels[neighbour] = i
                        q.put(neighbour)


    def run(self):
        self.set_neighbours_lists()
        for i in range(self.n):
            self.update_clusters(i)

        cluster_number = 0
        unique_clusters = np.unique(self.samples_labels)
        for unique_cluster in unique_clusters:
            if unique_cluster != -1:
                self.samples_labels[self.samples_labels == unique_cluster] = cluster_number
                cluster_number += 1

        return self.samples_labels
```

Our Hierarchical Clustering implementation:

```
import numpy as np
class hierarchical_clustering:
    def __init__(self, samples, linkage, k):
        self.samples = samples
        self.n = samples.shape[0]
        if linkage == 'single':
            self.update_next_linkage = self.single_linkage
        elif linkage == 'complete':
            self.update_next_linkage = self.complete_linkage
        self.k = k
        self.dist_matrix = self.compute_dist_matrix()
        self.samples_labels = np.array(range(self.n))
        self.min_dist_idx = [-1, -1]
    def compute_dist(self, xi, xj):
        return np.linalg.norm(xi -xj)
    def compute_dist_matrix(self):
        dist_matrix = np.zeros((self.n, self.n))
        for i in range(self.n):
            for j in range(i):
                dist = self.compute_dist(self.samples[i], self.samples[j])
                dist_matrix[i,j] = dist
                dist_matrix[j,i] = dist
            # fill diagonal with infinity values, that way clusters would not choose itself for combining
            dist_matrix[i, i] = np.inf
        return dist_matrix
    def single_linkage(self):
        # merge 'b' cluster to 'a' cluster
        a = self.min_dist_idx[0]
        b = self.min_dist_idx[1]
        for i in range(self.n):
            if (i != a and i != b):
                temp = min(self.dist_matrix[a][i], self.dist_matrix[b][i])
                self.dist_matrix[a][i] = temp
                self.dist_matrix[i][a] = temp
        # 'b' cluster merged into 'a'. Set dist from 'b' cluster to all other clusters to be infinity
        self.dist_matrix[b, :] = np.inf
        self.dist_matrix[:, b] = np.inf
    def complete_linkage(self):
        # merge 'b' cluster to 'a' cluster
        a = self.min_dist_idx[0]
        b = self.min_dist_idx[1]

        for i in range(self.n):
            if (i != a and i != b):
                temp = max(self.dist_matrix[a][i], self.dist_matrix[b][i])
                self.dist_matrix[a][i] = temp
                self.dist_matrix[i][a] = temp
        # 'b' cluster merged into 'a'. Set dist from 'b' cluster to all other clusters to be infinity
        self.dist_matrix[b, :] = np.inf
        self.dist_matrix[:, b] = np.inf
    def update_min_dist(self):
        self.min_dist_idx =  np.unravel_index(np.argmin(self.dist_matrix), (self.n, self.n))
    def update_clusters(self, i):
        self.update_min_dist()
        self.update_next_linkage()
        # Manipulating the dictionary to keep track of cluster formation in each step
        cluster_a = self.samples_labels[self.min_dist_idx[0]]
        cluster_b = self.samples_labels[self.min_dist_idx[1]]
        self.samples_labels[self.samples_labels == cluster_a] = self.n + i
        self.samples_labels[self.samples_labels == cluster_b] = self.n + i
    def run(self):
        # start from iteration number 0, and stop k iterations from the end
        for i in range(self.n - self.k):
            self.update_clusters(i)
        cluster_number = 0
        unique_clusters = np.unique(self.samples_labels)
        for unique_cluster in unique_clusters:
            self.samples_labels[self.samples_labels == unique_cluster] = cluster_number
            cluster_number += 1
        return self.samples_labels
```

Our Affinity Propagation implementation:

```
import numpy as np
class Affinity_propagation:
    # if preference == None, preference as median will be used
    def __init__(self, samples, preference, max_iter, convergence_iter, damping):
        self.samples = samples
        self.samples_labels = []
        self.n = samples.shape[0]
        self.preference = preference
        self.damping = damping
        self.S = self.compute_S()
        self.A = np.zeros((self.n, self.n))
        self.R = np.zeros((self.n, self.n))
        self.max_iter = max_iter
        self.convergence_iter = convergence_iter
        self.iterations_without_change = 0
    def similarity(self, xi, xj):
        return -(np.linalg.norm(xi -xj))**2
    def compute_S(self):
        S = np.zeros((self.n, self.n))
        for i in range(self.n):
            for j in range(i):
                similarity = self.similarity(self.samples[i], self.samples[j])
                S[i,j] = similarity
                S[j,i] = similarity
        if self.preference is None:
            self.preference = np.median(S)
        np.fill_diagonal(S, self.preference)
        return S
    def update_responsibilities(self):
        S_A = self.S + self.A
        # fill diagonal with infinity values, that way a sample would not choose itself as the exampler
        np.fill_diagonal(S_A, -np.inf)
        rows_max_idx = np.argmax(S_A, axis=1)
        rows_max = S_A[range(self.n), rows_max_idx]
        # index of max value may be pointing to itself, hence, compute the next max value
        S_A[range(self.n), rows_max_idx] = -np.inf
        next_rows_max = np.max(S_A, axis=1)
        # fill max_matrix with maximum values for each row
        max_matrix = np.transpose(np.tile(rows_max, (self.n, 1)).reshape((self.n, self.n)))
        # fix maximum value of the index of the real maximum, as k=k' is not desired.
        max_matrix[range(self.n), rows_max_idx] = next_rows_max
        self.R = self.R * self.damping + (1 - self.damping) * (self.S - max_matrix)
    def update_availabilities(self):
        R_clipped = self.R.copy()
        R_clipped = np.clip(R_clipped, 0, np.inf) # only sum of positive values is needed
        np.fill_diagonal(R_clipped, 0) # diagonal is not included in sum
        A = R_clipped.copy()
        # column wise sum including elements that i=i'
        A = np.tile(A.sum(axis=0), (self.n, 1)).reshape((self.n, self.n))
        # remove R(i,k) from A(i,k) that was wrongfully added
        A -= R_clipped
        A = A + self.R[range(self.n), range(self.n)]
        # choose minimum of 0 and R(k,k) + sum(max(0, R(i',k)) over i' st. i' not in {i,k}
        A = np.clip(A, -np.inf, 0)
        # fill A diagonal with sum(max(0, R(i',k)) over i' st. i' != k
        A[range(self.n, self.n)] = R_clipped.sum(axis=0)
        self.A = self.A * self.damping + (1 - self.damping) * A
    def run(self):
        for i in range(self.max_iter):
            self.update_responsibilities()
            self.update_availabilities()
            # check for convergence iteration restriction
            old_exemplars_amount = len(np.unique(self.samples_labels))
            sol = self.A + self.R
            self.samples_labels = np.argmax(sol, axis=1)
            new_exemplars_amount = len(np.unique(self.samples_labels))
            if old_exemplars_amount == new_exemplars_amount:
                self.iterations_without_change += 1
            else:
                self.iterations_without_change = 0
            if self.iterations_without_change >= self.convergence_iter:
                break
        cluster_number = 0
        unique_clusters = np.unique(self.samples_labels)
        for unique_cluster in unique_clusters:
            self.samples_labels[self.samples_labels == unique_cluster] = cluster_number
            cluster_number += 1
        return self.samples_labels, unique_clusters
```