Module 7 HW - Question 1

Implement k-means from scratch. You are given a dataset X whose rows represent different data points, you are asked to perform a k-means clustering on this dataset using the Manhattan Distance, k is chosen as 3.

$$X = \begin{bmatrix} 5.7 & 64 \\ 4.7 & 58 \\ 6.1 & 56 \\ 4.6 & 64 \\ 5.4 & 84 \\ 4.9 & 60 \\ 5.0 & 62 \\ 6.4 & 62 \\ 5.1 & 76 \\ 6.0 & 60 \end{bmatrix}$$

The Manhattan Distance of (x_1, y_1) and (x_2, y_2) is calculated by

$\text{Dist} = |x_1 - x_2| + |y_1 - y_2|$

Module 7 HW - Question 1.a

Since first column and second column are not on the same scale. Before running K-means, this dataset needs to be preprocessed, Show the preprocessed dataset. (Answer in the format of [x1, x2], round your results to two decimal places, same as problems b and c)

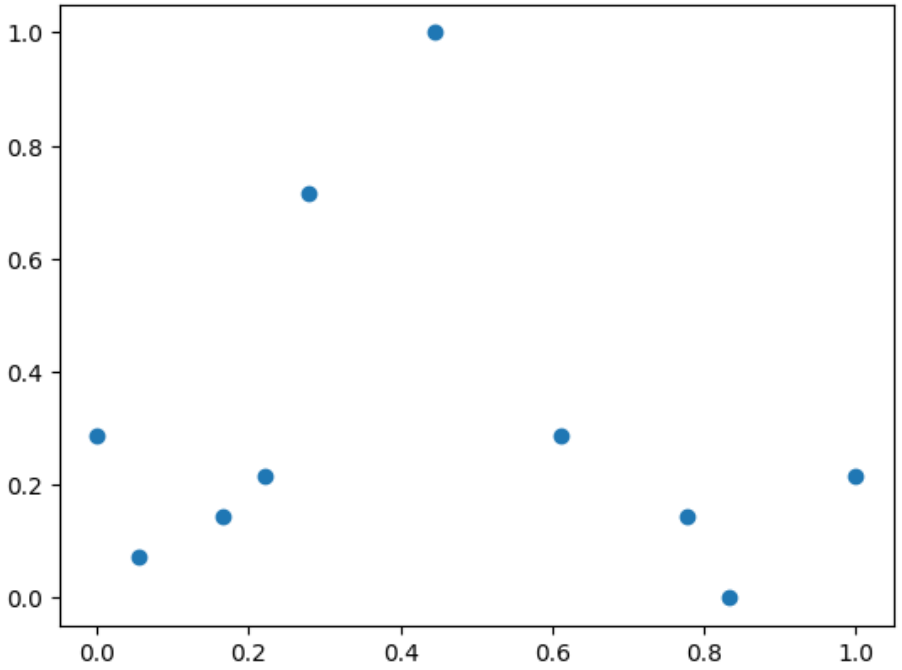
We will conduct normalization on the dataset in order to be sure that the values are on the same scale. We will use Min-Max Scaling($x_{\text{normalized}} = \frac{x - min(x)}{max(x) - min(x)}$) to conduct the normalization on the data. This normalization will scale the first feature data and second feature data to values between 0 and 1.

In [ ]:
```
# import required packages
import numpy as np
import copy
```

```
# set the data into an array
X = np.array([
    [5.7, 64],
    [4.7, 58],
    [6.1, 56],
    [4.6, 64],
    [5.4, 84],
    [4.9, 60],
    [5.0, 62],
    [6.4, 62],
    [5.1, 76],
    [6.0, 60]
])

# define a function for min max scaling
def min_max_scale(X):

    X_pp = copy.deepcopy(X)
    for col in range(X.shape[1]):
        for row in range(len(X)):
            X_pp[row, col] = (X[row, col] - X[:, col].min())/(X[:, col].max() - X[:, col].min())

    return X_pp
```

In [ ]:
```
# import plotting packages
import matplotlib.pyplot as plt

# apply min max scaling to our data
X_pp = min_max_scale(X)
print(X_pp)

# plot the normalized data
plt.scatter(X_pp[:, 0], X_pp[:, 1])
plt.show()
```

```
[[0.61111111 0.28571429]
 [0.05555556 0.07142857]
 [0.83333333 0.        ]
 [0.         0.28571429]
 [0.44444444 1.        ]
 [0.16666667 0.14285714]
 [0.22222222 0.21428571]
 [1.         0.21428571]
 [0.27777778 0.71428571]
 [0.77777778 0.14285714]]
```



Module 7 HW - Question 1.b

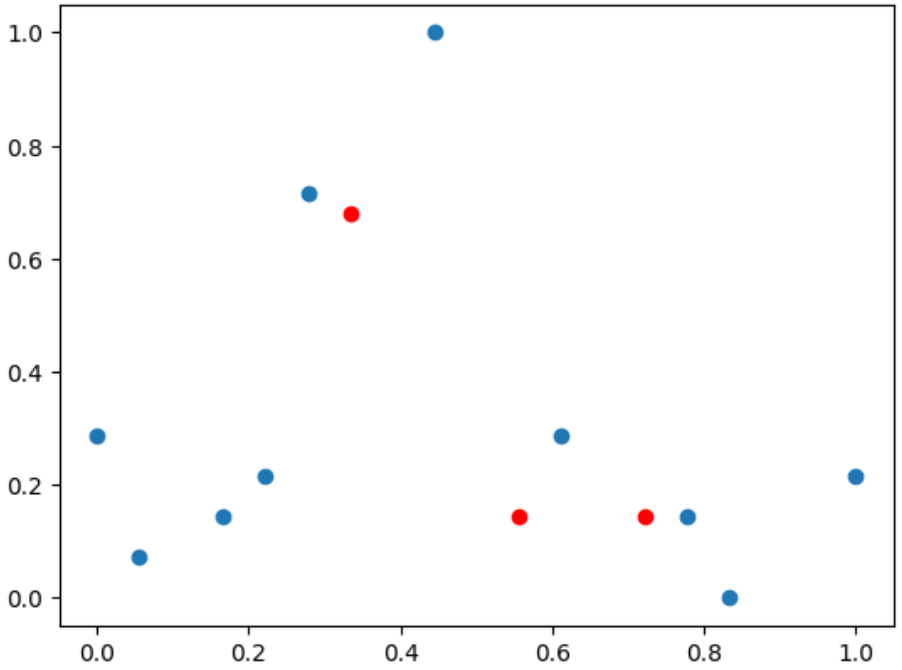
Suppose the initial centroids of the clusters are $\mu_1 = [5.6, 60]$, $\mu_2 = [5.9, 60]$, $\mu_3 = [5.2, 75]$. What's the center of the second cluster after two iterations?

These centroids need to be normalized first. Once they are normalized then we can calculate the first iteration of manhattan distances between each value and the centroids to determine which values lie closest to each centroid. Once we identify those values we can then calculate the mean distances in order to find the second clusters/centroids.

In [ ]:
```
# set un-normalized centroids in arrays
c1 = np.array([[5.6, 60]])
c2 = np.array([[5.9, 60]])
c3 = np.array([[5.2, 75]])

# set min and max values for each column of the dataset.
X_0_min = X[:, 0].min()
X_0_max = X[:, 0].max()
X_1_min = X[:, 1].min()
```

```
X_1_max = X[:, 1].max()

# standardize data based on original data
c1_pp = np.array([
    [(c1[0, 0]-X_0_min)/(X_0_max-X_0_min), (c1[0, 1]-X_1_min)/(X_1_max-X_1_min)]
])

c2_pp = np.array([
    [(c2[0, 0]-X_0_min)/(X_0_max-X_0_min), (c2[0, 1]-X_1_min)/(X_1_max-X_1_min)]
])

c3_pp = np.array([
    [(c3[0, 0]-X_0_min)/(X_0_max-X_0_min), (c3[0, 1]-X_1_min)/(X_1_max-X_1_min)]
])

# print the results
print('c1 preprocessed: ', c1_pp)
print('c2 preprocessed: ', c2_pp)
print('c3 preprocessed: ', c3_pp)

# plot the normalized centroids
plt.scatter(X_pp[:, 0], X_pp[:, 1])
plt.scatter(np.array([c1_pp[0], c2_pp[0], c3_pp[0]])[:, 0], np.array([c1_pp[0], c2_pp[0], c3_pp[0]])[:, 1], color='red')
plt.show()
```

```
c1 preprocessed:  [[0.55555556 0.14285714]]
c2 preprocessed:  [[0.72222222 0.14285714]]
c3 preprocessed:  [[0.33333333 0.67857143]]
```



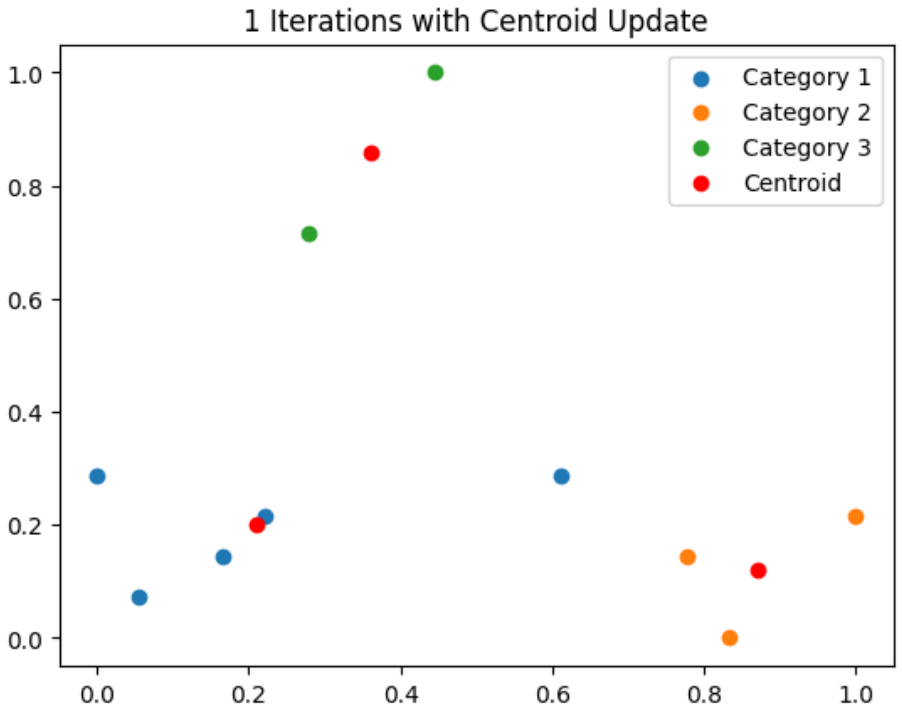
Now we can calculate the manhatten distance of the centroids between each point and determine which points are closests to each centroid.

In [ ]:
```
# creating a centroids dictionary to iterate through preprocessed centroid data
cent_dict = {1: c1_pp, 2: c2_pp, 3: c3_pp}

# manhatten distance calculations and new centroid locations
def manhattan_dist_calc(X, centroids_dict):

    # copy array to return newly created array with centroid and remove cluster label if it is available in the dataset
    X_md = copy.deepcopy(X)
    X_md = X_md[:, 0:2]
    X_md = np.hstack((X_md, np.zeros((X_md.shape[0], 1))))

    # dictionary to hold centroid distance values to answer question 1.b
    centroid_distance_values = {}

    # check the distance of each point in the array from each centroid and appoint the smallest value
    for row in range(len(X)):
        # used to hold manhatten distance values for each centroid
        temp_cent_dict_ = {}
        for dict_name in centroids_dict:
            # calculate the manhatten distance for the centroid
            temp_cent_dict_.update({dict_name: abs(X[row, 0] - centroids_dict[dict_name][0][0]) + np.abs(X[row, 1] - centroids_dic
```

```
        # get minimum valued centroid to set as label for that cluster
        min_centroid = min(zip(temp_cent_dict_.values(), temp_cent_dict_.keys()))[1]
        centroid_distance_values.update({min(zip(temp_cent_dict_.values(), temp_cent_dict_.keys()))[1]: min(zip(temp_cent_dict_.va
        X_md[row, 2] = min_centroid

    # calculate new centroids
    # list of unique centroid categories
    centroid_cat = np.unique(X_md[:, 2])

    # empty array that will hold our new centroids
    centroid_array = np.zeros((len(np.unique(X_md[:, 2])), 2))

    # updating the centroids dictionary so we can re-run this function to find convergence
    centroids_dict_copy = copy.deepcopy(centroids_dict)

    # loop to iterate through data and calculate centroid means
    for num, cent_cat in enumerate(centroid_cat):
        X_cent_mean = X_md[X_md[:, 2] == cent_cat][:, 0].mean()
        y_cent_mean = X_md[X_md[:, 2] == cent_cat][:, 1].mean()

        # setting centroid means into a centroid array that will be used for visualizations
        centroid_array[num, 0] = X_cent_mean
        centroid_array[num, 1] = y_cent_mean

        # setting centroid means into a new dictioanry that we will use for iterations
        centroids_dict_copy[cent_cat] = np.array([[X_cent_mean, y_cent_mean]])


    return X_md, centroid_array, centroids_dict_copy, centroid_distance_values

X_labeled, cent_array1, new_cent_dict1, centroid_distance_values1 = manhattan_dist_calc(X_pp, cent_dict)

print('normalized array with centroid category set using Manhatten Distances')
print(X_labeled)

print('\n\nInitial Centroids')
print(cent_dict, '\n\n')

print('1 Iteration: New Centroids')
print(new_cent_dict1, '\n\n')

print('1 Iteration: Centroid Distance Values')
print(centroid_distance_values1, '\n\n')

# extracting x, y, and category values
x = X_labeled[:, 0]
y = X_labeled[:, 1]
categories = X_labeled[:, 2]

# get unique categories
```

```
unique_categories = np.unique(categories)

# plot each category
for category in unique_categories:
    # select data for the category
    idx = categories == category
    plt.scatter(x[idx], y[idx], label=f'Category {int(category)}')
plt.scatter(cent_array1[:, 0], cent_array1[:, 1], color='red', label='Centroid')
plt.legend()
plt.title('1 Iterations with Centroid Update')
plt.show()
```

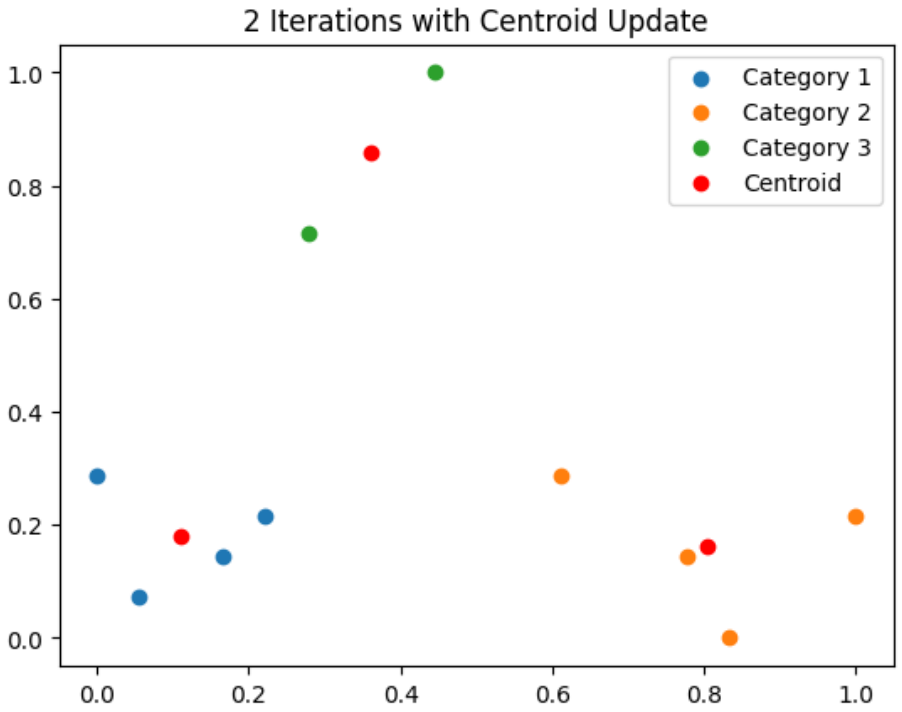
```
normalized array with centroid category set using Manhatten Distances
[[0.61111111 0.28571429 1.        ]
 [0.05555556 0.07142857 1.        ]
 [0.83333333 0.         2.        ]
 [0.         0.28571429 1.        ]
 [0.44444444 1.         3.        ]
 [0.16666667 0.14285714 1.        ]
 [0.22222222 0.21428571 1.        ]
 [1.         0.21428571 2.        ]
 [0.27777778 0.71428571 3.        ]
 [0.77777778 0.14285714 2.        ]]


Initial Centroids
{1: array([[0.55555556, 0.14285714]]), 2: array([[0.72222222, 0.14285714]]), 3: array([[0.33333333, 0.67857143]])}


1 Iteration: New Centroids
{1: array([[0.21111111, 0.2       ]]), 2: array([[0.87037037, 0.11904762]]), 3: array([[0.36111111, 0.85714286]])}


1 Iteration: Centroid Distance Values
{1: 0.40476190476190443, 2: 0.05555555555555536, 3: 0.0912698412698415}
```



In [ ]:
```
print('Initial Centroids')
print(cent_dict, '\n\n')

print('1 Iteration: New Centroids')
print(new_cent_dict1, '\n\n')

print('1 Iteration: Centroid Distance Values')
print(centroid_distance_values1, '\n\n')

X_labeled2, cent_array2, new_cent_dict2, centroid_distance_values2 = manhattan_dist_calc(X_pp, new_cent_dict1)

print('2 Iteration: New Centroids')
print(new_cent_dict2, '\n\n')

print('2 Iteration: Centroid Distance Values')
print(centroid_distance_values2, '\n\n')

# extracting x, y, and category values
x = X_labeled2[:, 0]
y = X_labeled2[:, 1]
categories = X_labeled2[:, 2]

# get unique categories
```

```
unique_categories = np.unique(categories)

# plot each category
for category in unique_categories:
    # select data for the category
    idx = categories == category
    plt.scatter(x[idx], y[idx], label=f'Category {int(category)}')
plt.scatter(cent_array2[:, 0], cent_array2[:, 1], color='red', label='Centroid')
plt.legend()
plt.title('2 Iterations with Centroid Update')
plt.show()
```

```
Initial Centroids
{1: array([[0.55555556, 0.14285714]]), 2: array([[0.72222222, 0.14285714]]), 3: array([[0.33333333, 0.67857143]])}


1 Iteration: New Centroids
{1: array([[0.21111111, 0.2       ]]), 2: array([[0.87037037, 0.11904762]]), 3: array([[0.36111111, 0.85714286]])}


1 Iteration: Centroid Distance Values
{1: 0.40476190476190443, 2: 0.05555555555555536, 3: 0.0912698412698415}


2 Iteration: New Centroids
{1: array([[0.11111111, 0.17857143]]), 2: array([[0.80555556, 0.16071429]]), 3: array([[0.36111111, 0.85714286]])}


2 Iteration: Centroid Distance Values
{2: 0.11640211640211638, 1: 0.025396825396825362, 3: 0.22619047619047639}
```



After two iterations the centroid of the second cluster is at [0.87, 0.12]

Module 7 HW - Question 1.c

What's the center of the third cluster when the clustering converges?

It seems that convergence occurs on the second iteration when we see that the first iteration centroids equals the second iteration centroids.
The center of the third cluster is [0.36, 0.86]

Module 7 HW - Question 1.d

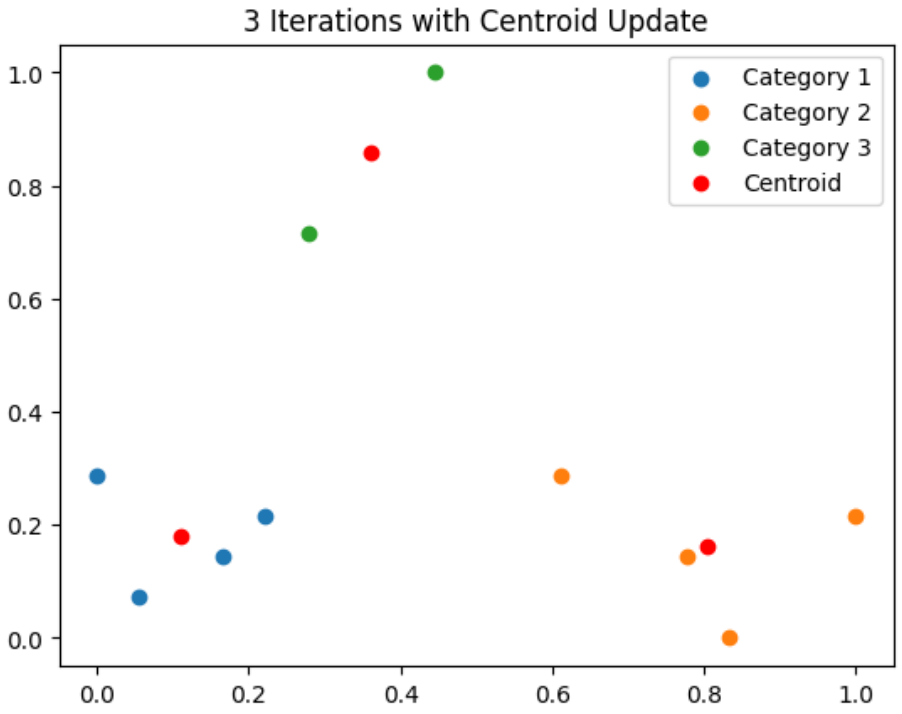
How many iterations are required for the clusters to converge?

We will run a 3rd iteration to be sure that we have met convergence at iteration 2

In [ ]:
```
print('Initial Centroids')
print(cent_dict, '\n\n')

print('1 Iteration: New Centroids')
print(new_cent_dict1, '\n\n')

print('1 Iteration: Centroid Distance Values')
print(centroid_distance_values1, '\n\n')

print('2 Iteration: New Centroids')
print(new_cent_dict2, '\n\n')

print('2 Iteration: Centroid Distance Values')
print(centroid_distance_values2, '\n\n')

X_labeled3, cent_array3, new_cent_dict3, centroid_distance_values3 = manhattan_dist_calc(X_pp, new_cent_dict2)

print('3 Iteration: New Centroids')
print(new_cent_dict2, '\n\n')

print('3 Iteration: Centroid Distance Values')
print(centroid_distance_values3, '\n\n')

# extracting x, y, and category values
x = X_labeled3[:, 0]
y = X_labeled3[:, 1]
categories = X_labeled3[:, 2]

# get unique categories
unique_categories = np.unique(categories)

# plot each category
for category in unique_categories:
    # select data for the category
    idx = categories == category
    plt.scatter(x[idx], y[idx], label=f'Category {int(category)}')
plt.scatter(cent_array3[:, 0], cent_array3[:, 1], color='red', label='Centroid')
plt.legend()
```

```
plt.title('3 Iterations with Centroid Update')
plt.show()
```

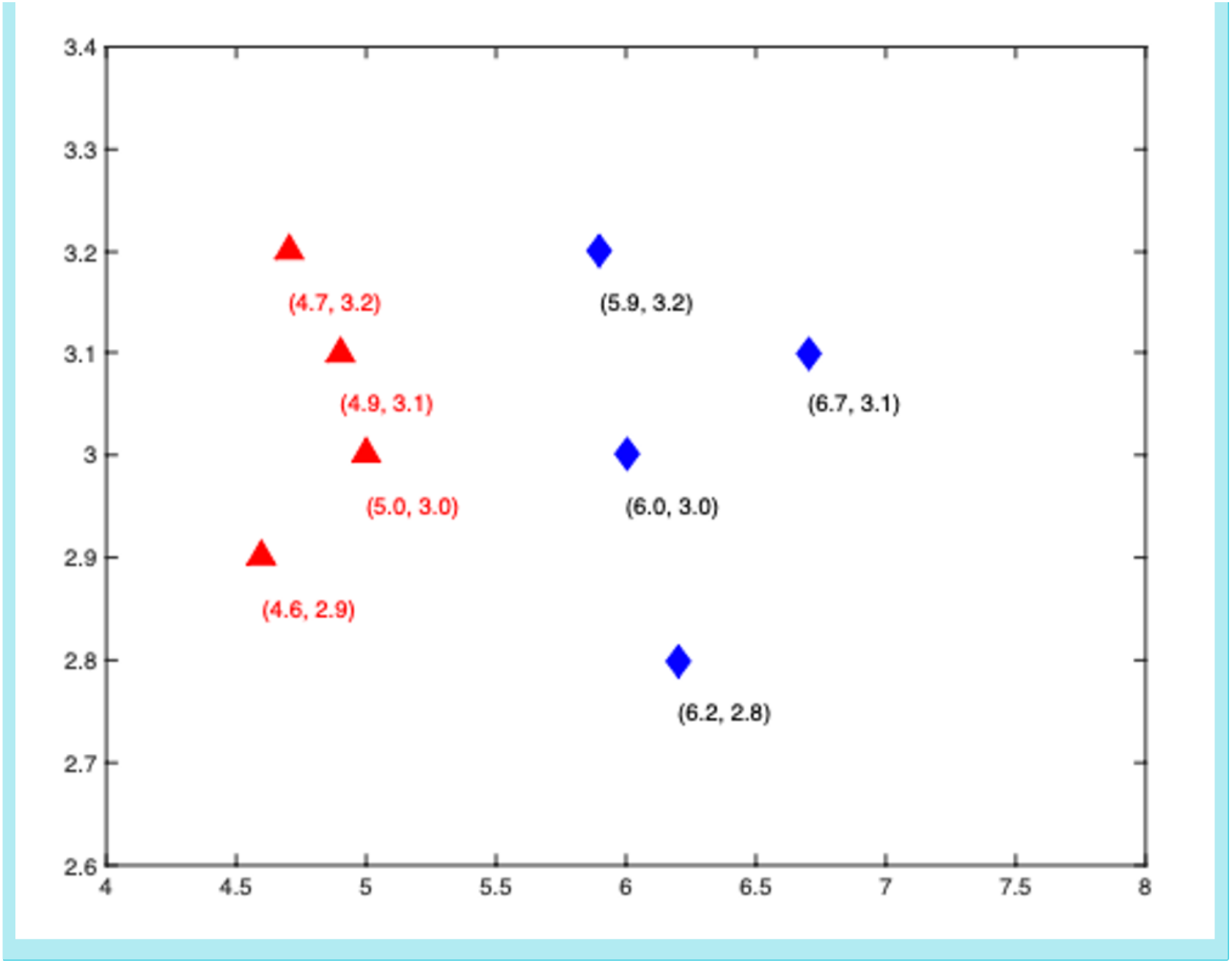
```
Initial Centroids
{1: array([[0.55555556, 0.14285714]]), 2: array([[0.72222222, 0.14285714]]), 3: array([[0.33333333, 0.67857143]])}


1 Iteration: New Centroids
{1: array([[0.21111111, 0.2       ]]), 2: array([[0.87037037, 0.11904762]]), 3: array([[0.36111111, 0.85714286]])}


1 Iteration: Centroid Distance Values
{1: 0.40476190476190443, 2: 0.05555555555555536, 3: 0.0912698412698415}


2 Iteration: New Centroids
{1: array([[0.11111111, 0.17857143]]), 2: array([[0.80555556, 0.16071429]]), 3: array([[0.36111111, 0.85714286]])}


2 Iteration: Centroid Distance Values
{2: 0.11640211640211638, 1: 0.025396825396825362, 3: 0.22619047619047639}


3 Iteration: New Centroids
{1: array([[0.11111111, 0.17857143]]), 2: array([[0.80555556, 0.16071429]]), 3: array([[0.36111111, 0.85714286]])}


3 Iteration: Centroid Distance Values
{2: 0.04563492063492064, 1: 0.14682539682539675, 3: 0.22619047619047639}
```



Centroid distances and Centroid positions require 2 iterations to converge

Module 7 HW - Question 2

Hierarchical Clustering: Suppose there are two clusters A (red) and B (blue), each has four members and plotted in Figure below, compute the distance between two clusters using Euclidean distance.



In order to calculte the Euclidean distance between the two clusters, we will need to first get the mean of each column, separated by their clusters. from this, we can the use the Euclidean distance formula, $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$. However, because we're conducting the calculations in an array, we can

make the calculation by $\sqrt{\sum_{i=1}^{n}(p_{2i} - p_{1i})^2}$

In [ ]:
```
h_clustering = np.array([
    [4.7, 3.2, 0],
    [5.9, 3.2, 1],
    [4.9, 3.1, 0],
    [6.7, 3.1, 1],
    [5.0, 2.9, 0],
    [6.0, 3.0, 1],
    [4.6, 2.9, 0],
    [6.2, 2.8, 1]
])
cluster_A_points = h_clustering[h_clustering[:, 2] == 0][:, :2]
cluster_B_points = h_clustering[h_clustering[:, 2] == 1][:, :2]
cluster_A_mean_points = np.mean(cluster_A_points, axis=0)
cluster_B_mean_points = np.mean(cluster_B_points, axis=0)

euclid_dist = np.sqrt(np.sum(cluster_A_mean_points - cluster_B_mean_points)**2)
print(f'Euclidean Distance Measure for the above data: {round(euclid_dist, 2)}')
```

```
Euclidean Distance Measure for the above data: 1.4
```

Module 7 HW - Question 2.a

What is the distance between the two farthest members (Complete-link) (round to four decimal places here, and next 2 problems)?

In order to determine the two farthest members by calculating the complete-link distance, we will need to calculate the pairwise euclidean distances, using np.linalg.norm which will calculate the pairwise between the two clusters. From there, we will get the maximum distance and the pair of points who make up this max difference.

In [ ]:
```
# Calculate all pairwise distances between points in cluster A and B
all_pairwise_distances = np.linalg.norm(cluster_A_points[:, np.newaxis] - cluster_B_points, axis=2)

# Find the minimum distance (the single-linkage distance)
complete_link_distance = np.max(all_pairwise_distances)

# Get the indices of the closest members
max_distance_indices = np.where(all_pairwise_distances == complete_link_distance)

# Extract the members that have the single link distance
farthest_members_A = cluster_A_points[max_distance_indices[0]]
farthest_members_B = cluster_B_points[max_distance_indices[1]]
```

```
complete_link_distance, farthest_members_A, farthest_members_B
print(f'max distance (complete-link distance): {round(complete_link_distance, 2)}\nfarthest points: {farthest_members_A, farthest_
```

```
max distance (complete-link distance): 2.11
farthest points: (array([[4.6, 2.9]]), array([[6.7, 3.1]]))
```

Module 7 HW - Question 2.b

What is the distance between the two closest members (Single-link)?

We've already calculated the pairwise distances, we will need to just use the same methods to extract the min distances of the pairwise euclidean distances.

In [ ]:
```
# Find the minimum distance (the single-linkage distance)
single_link_distance = np.min(all_pairwise_distances)

# Get the indices of the closest members
min_distance_indices = np.where(all_pairwise_distances == single_link_distance)

# Extract the members that have the single link distance
closest_members_A = cluster_A_points[min_distance_indices[0]]
closest_members_B = cluster_B_points[min_distance_indices[1]]

single_link_distance, closest_members_A, closest_members_B
print(f'min distance (single-link distance): {round(single_link_distance, 2)}\nfarthest points: {closest_members_A, closest_member
```

```
min distance (single-link distance): 0.95
farthest points: (array([[5. , 2.9]]), array([[5.9, 3.2]]))
```

Module 7 HW - Question 2.c

What is the average distance between all pairs (Average-link)?

Here, rather than get the min or max, we will get the mean of all pairwise euclidean distances.

In [ ]:
```
# Find the minimum distance (the single-linkage distance)
average_link_distance = np.mean(all_pairwise_distances)
print(f'mean distance (average-link distance): {round(average_link_distance, 2)}')
```

```
mean distance (average-link distance): 1.41
```

Module 7 HW - Question 2.d

Among all three distances above, which one is robust to noise?

- Complete-Link is the most robust to noise. Due to it's measures of defining clusters as the furthest neighbor between any group of points between clusters it results in clustering to those points who are relatively close to one another in order for them to be merged. This causes a reduction of influence from outliers and noise causing more tightly bound clusters.

Module 7 HW - Question 3

Fill out the code cells in hw_7.ipynb and answer the questions.

Unsupervised Learning - Clustering

We've shown several unsupervised learning algorithms in this unit, We first introduced the simplest clustering algorihtm k-means. Hierarchical clustering included agglomerative and divisive clustering algorithms. Finally we discussed a powerful density-based algorihtm DBSCAN that performs fairly well when the clusters have irregular shapes, In this coding assignment you will explore more clustering algorithms and become familiar with the sklearn's clustering package. You'll also need to tune the hyper parameters of these models and get a sense about how the hyperparameters influence the shape of resulting clusters.

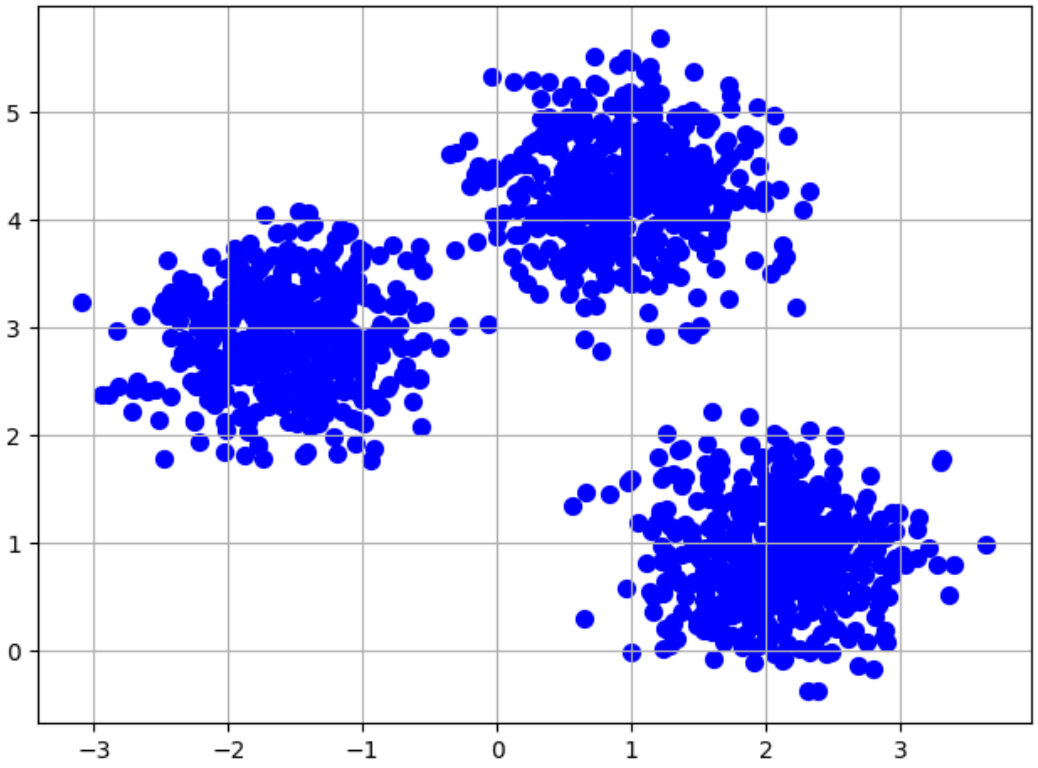
Loading the Dataset

Load dataset with different cluster shapes and try algorithms on them.

In [ ]:
```
from sklearn.datasets import make_blobs, make_moons, make_circles, make_swiss_roll
import matplotlib.pyplot as plt
import numpy as np
from sklearn.cluster import KMeans, AgglomerativeClustering, DBSCAN
from sklearn.preprocessing import StandardScaler
from sklearn import metrics
```

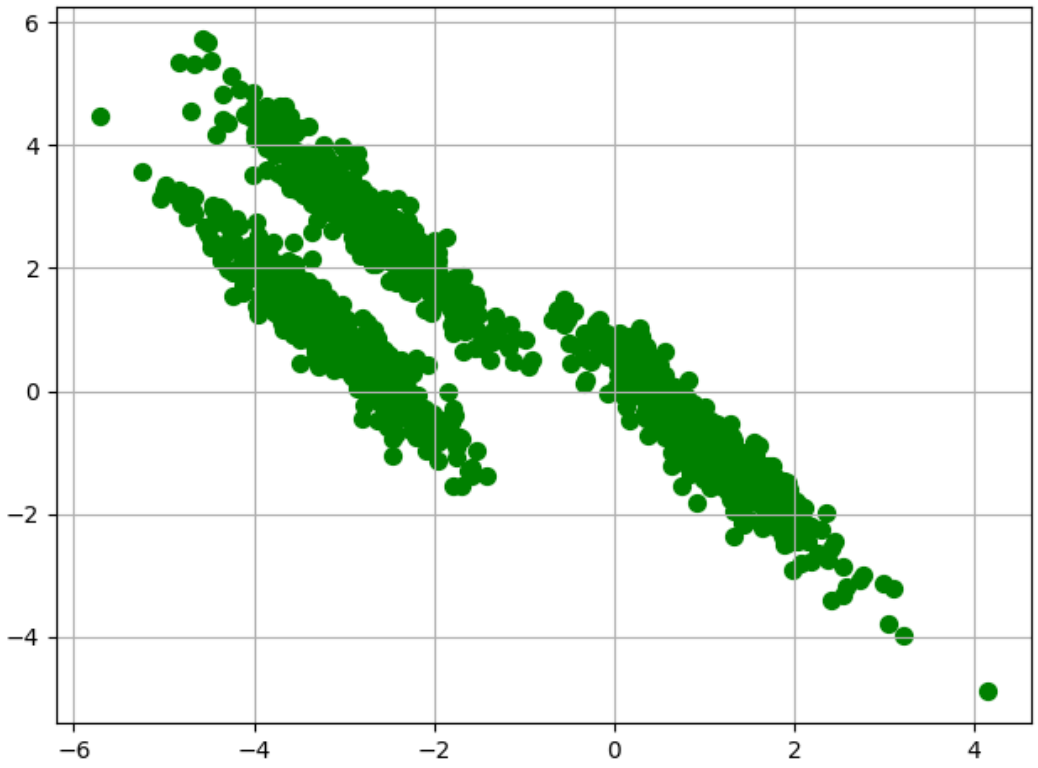
```
import time
import warnings
```

In [ ]:
```
n_samples = 1500
X_blobs, y_blobs = make_blobs(n_samples= n_samples,
                  n_features=2,
                  centers=3,
                  cluster_std=0.5,
                  shuffle=True,
                  random_state=0)
plt.scatter(X_blobs[:, 0], X_blobs[:, 1], c='blue', marker='o', s=50)
plt.grid()
plt.tight_layout()
plt.show()
```

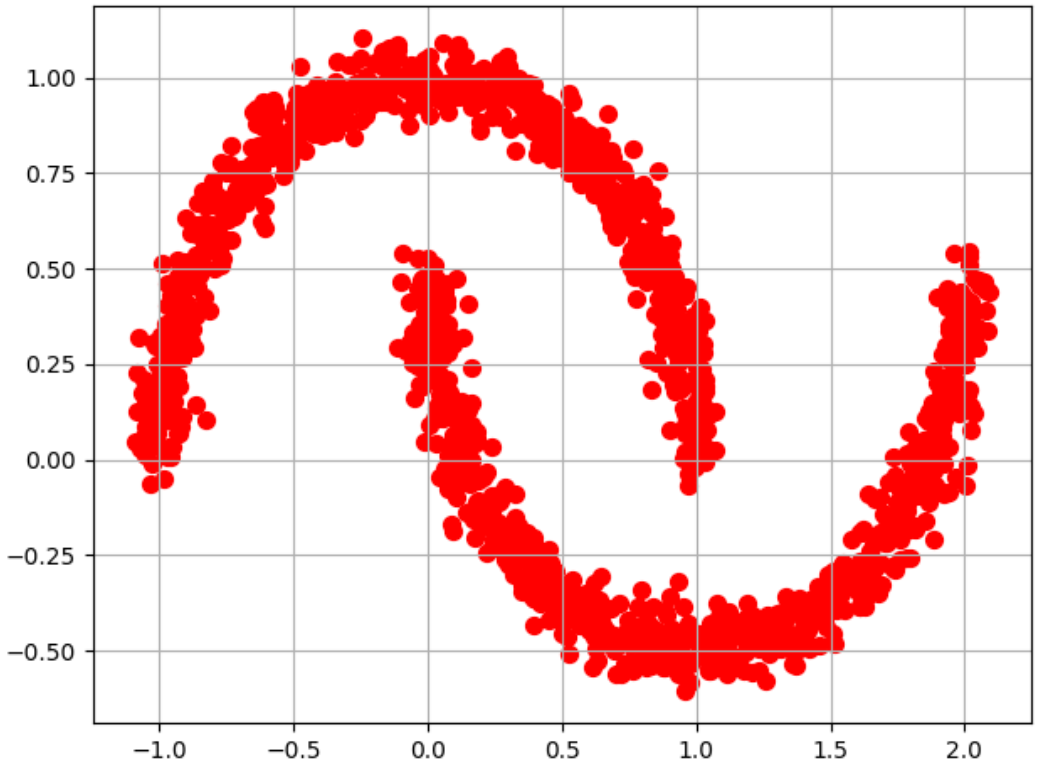


In [ ]:
```
random_state = 170
X_blobs1, y_blobs1 = make_blobs(n_samples=n_samples, random_state=random_state)
transformation = [[0.6, -0.6], [-0.4, 0.8]]
X_aniso = np.dot(X_blobs1, transformation)
aniso = (X_aniso, y_blobs1)
plt.scatter(aniso[0][:, 0], aniso[0][:, 1], c='green', marker='o', s=50)
```

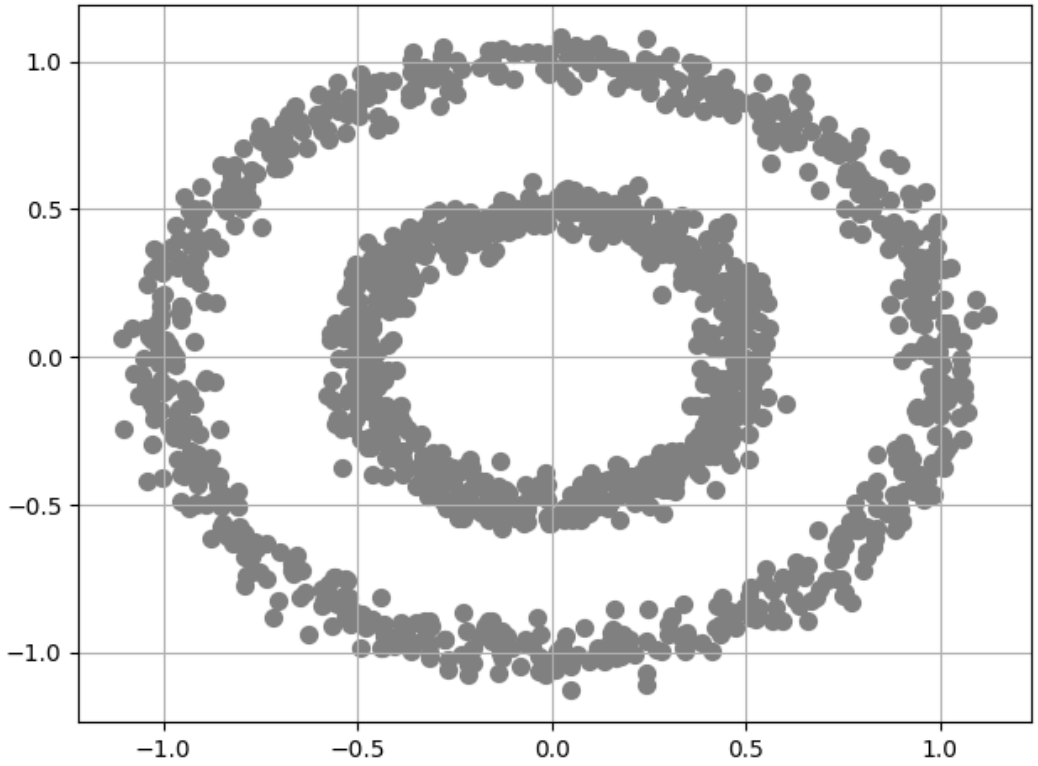
```
plt.grid()
plt.tight_layout()
plt.show()
```



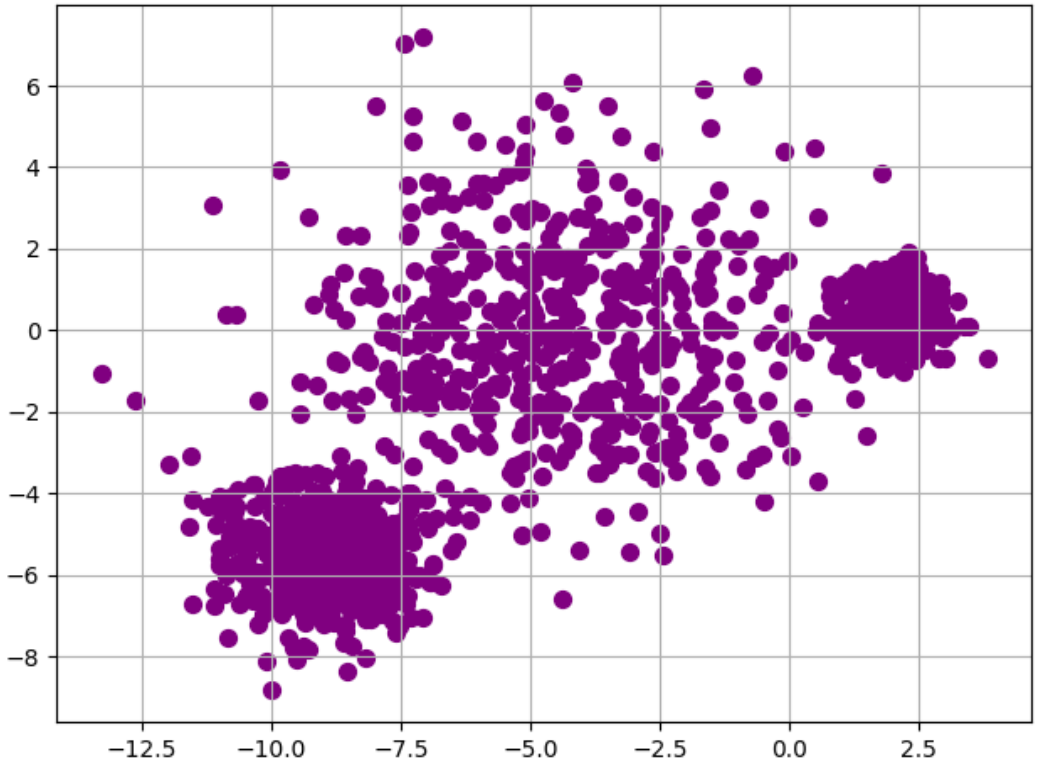
In [ ]:
```
noisy_moons = make_moons(n_samples=n_samples, noise=.05)
plt.scatter(noisy_moons[0][:, 0], noisy_moons[0][:, 1], c='red', marker='o', s=50)
plt.grid()
plt.tight_layout()
plt.show()
```



In [ ]:
```
noisy_circles = make_circles(n_samples=n_samples, factor=.5,
                                      noise=.05)
plt.scatter(noisy_circles[0][:, 0], noisy_circles[0][:, 1], c='grey', marker='o', s=50)
plt.grid()
plt.tight_layout()
plt.show()
```



In [ ]:
```
varied = make_blobs(n_samples=n_samples,
                            cluster_std=[1.0, 2.5, 0.5],
                            random_state=random_state)
plt.scatter(varied[0][:, 0], varied[0][:, 1], c='purple', marker='o', s=50)
plt.grid()
plt.tight_layout()
plt.show()
```



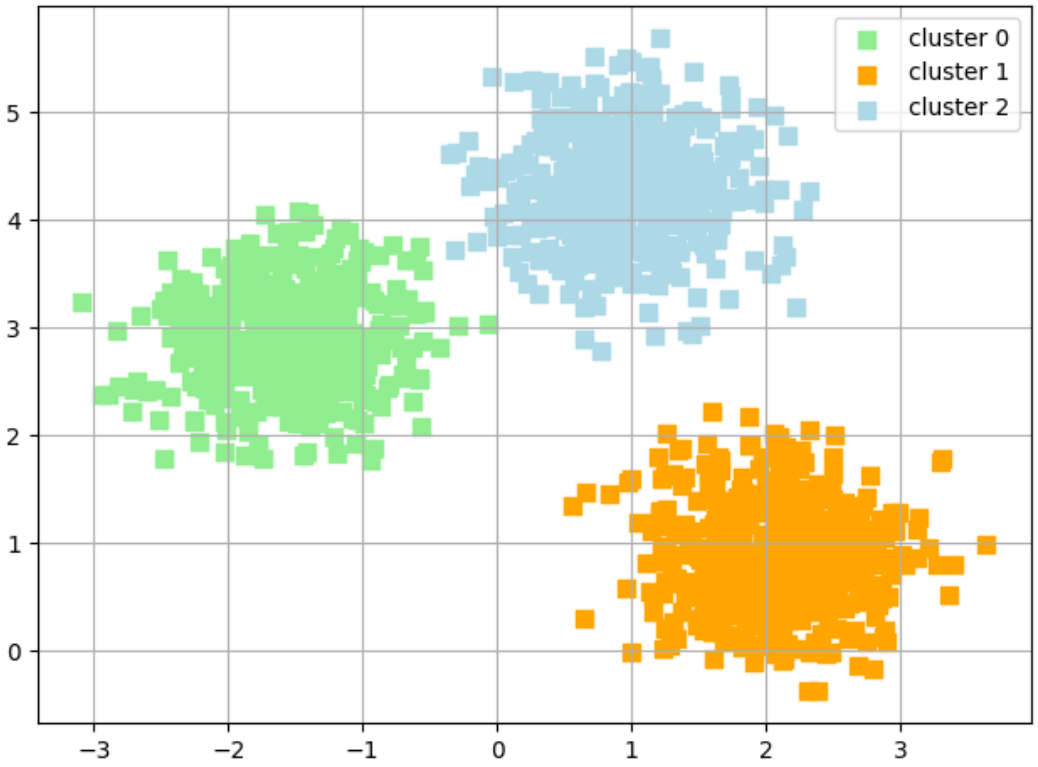
Grouping objects by similarity using k-means

K-means algorithm is simple but it performs poorly to enlongated clusters, or manifolds with irregular shapes.

In [ ]:
```
def print_cluster(model, n_clusters, X):
    y_km = model.fit_predict(X)
    color_list = ['lightgreen', 'orange', 'lightblue', 'red', 'yellow', 'brown', 'cyan']
    for i in range(n_clusters):
        plt.scatter(X[y_km == i, 0],
        X[y_km == i, 1],
        s=50,
        c=color_list[i],
        marker='s',
        label='cluster ' + str(i))
    plt.legend()
    plt.grid()
    plt.tight_layout()
    plt.show()
```

In [ ]:
```
km = KMeans(n_clusters=3,
            init='random',
            n_init=10,
            max_iter=300,
            tol=1e-04,
            random_state=0)
```

In [ ]:
```
print_cluster(km, 3, X_blobs)
```

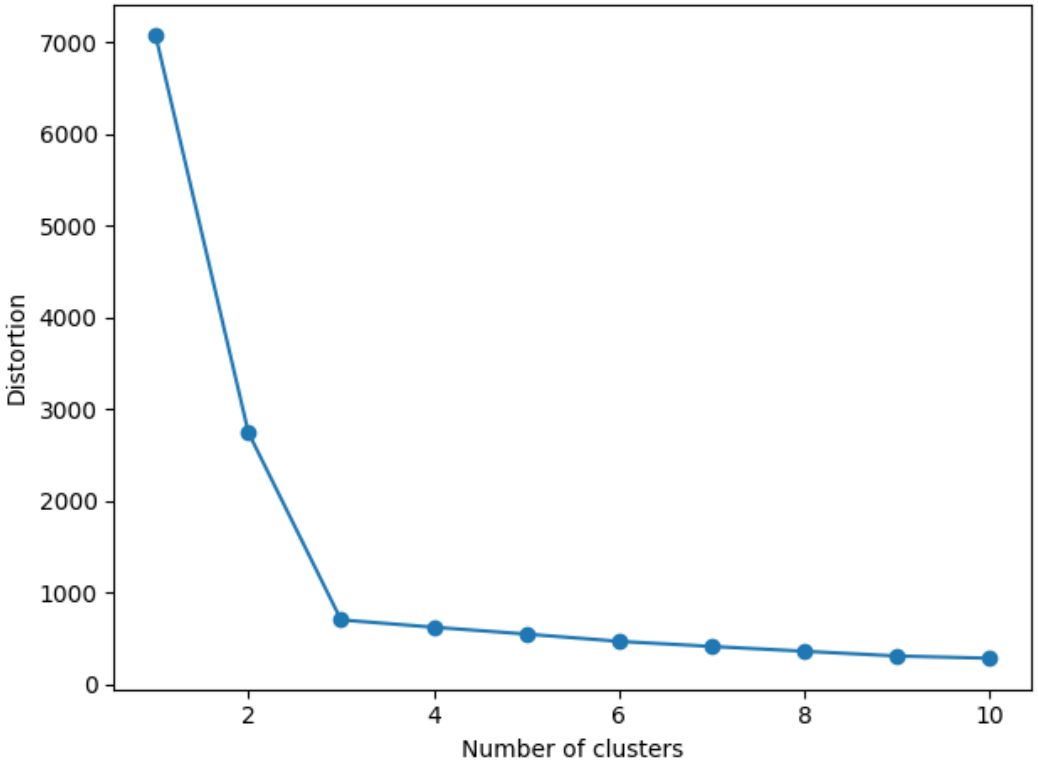


Using the elbow method to find the optimal number of clusters

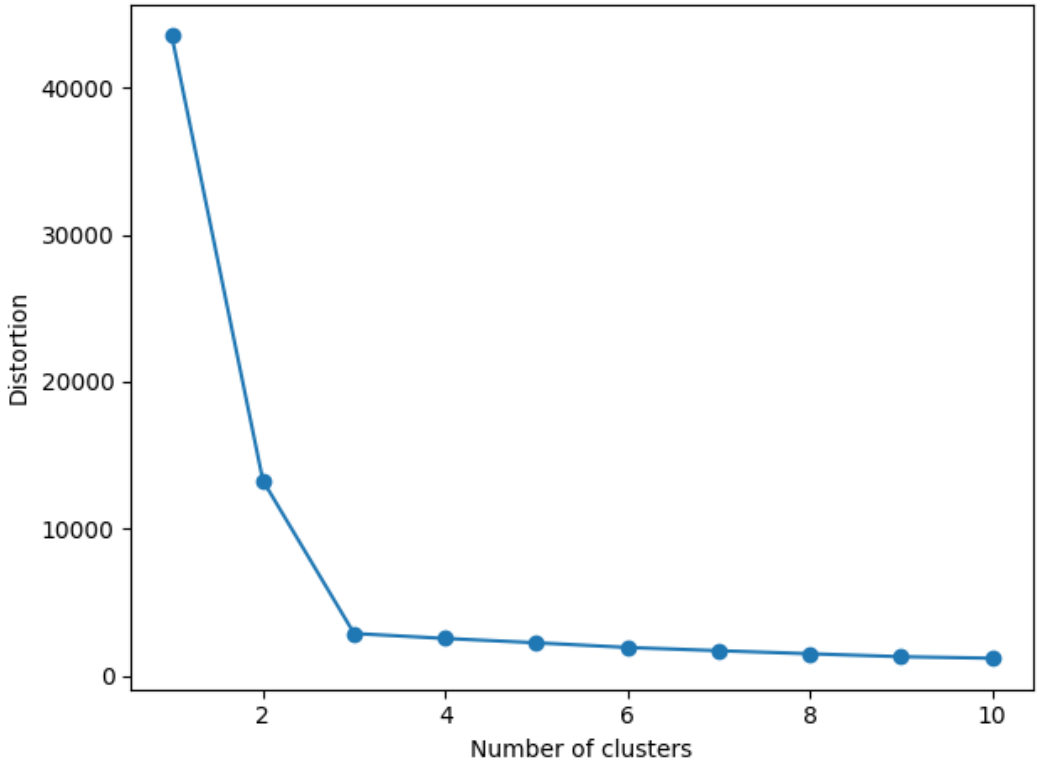
In [ ]:
```
def plot_distortions(data):
    distortions = []
    for i in range(1, 11):
        km = KMeans(n_clusters=i,
                init='k-means++',
                n_init=10,
                max_iter=300,
                random_state=0)
```

```
        km.fit(data)
        distortions.append(km.inertia_)
    plt.plot(range(1, 11), distortions, marker='o')
    plt.xlabel('Number of clusters')
    plt.ylabel('Distortion')
    plt.tight_layout()
    plt.show()
```

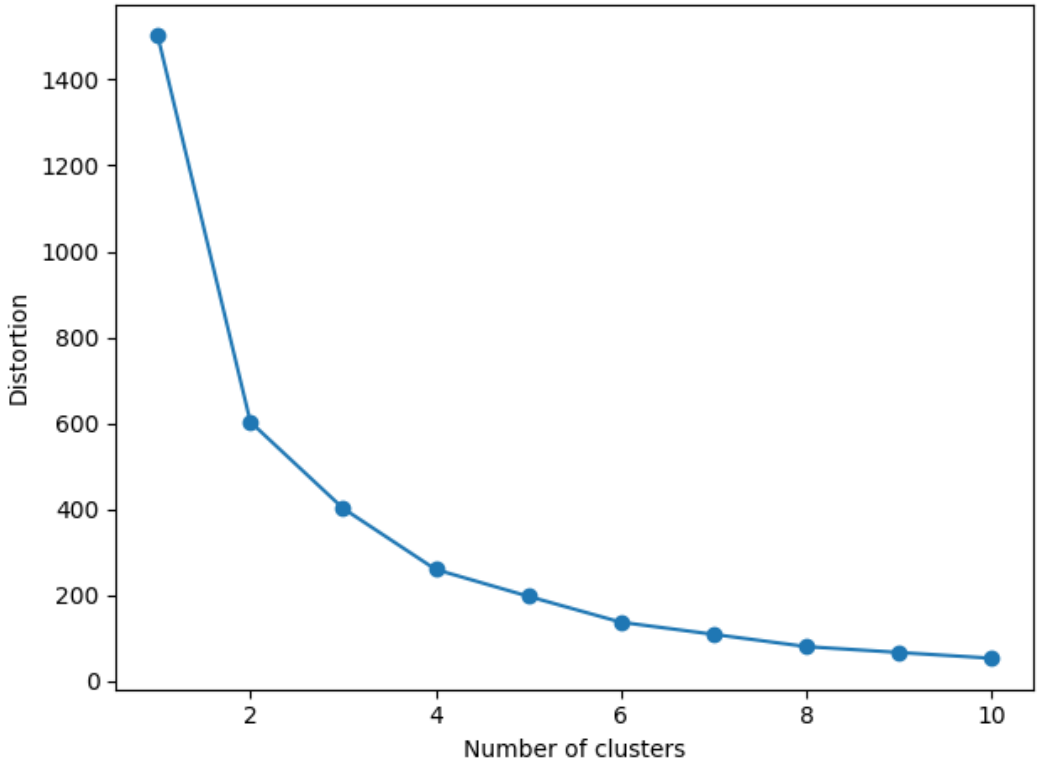
In [ ]:
```
plot_distortions(X_blobs) # select k=3
```



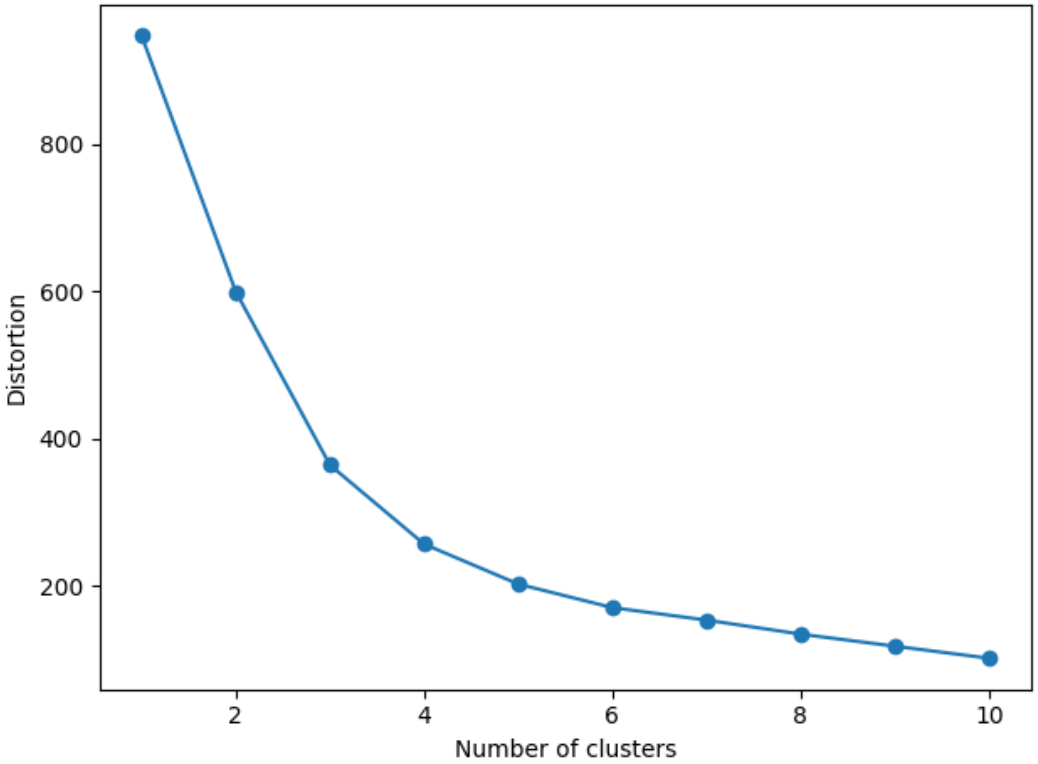
In [ ]:
```
# TODO :: run the kmeans on the other datasets and use elbow method to select the number of clusters.

plot_distortions(X_blobs1) # select k=3
```



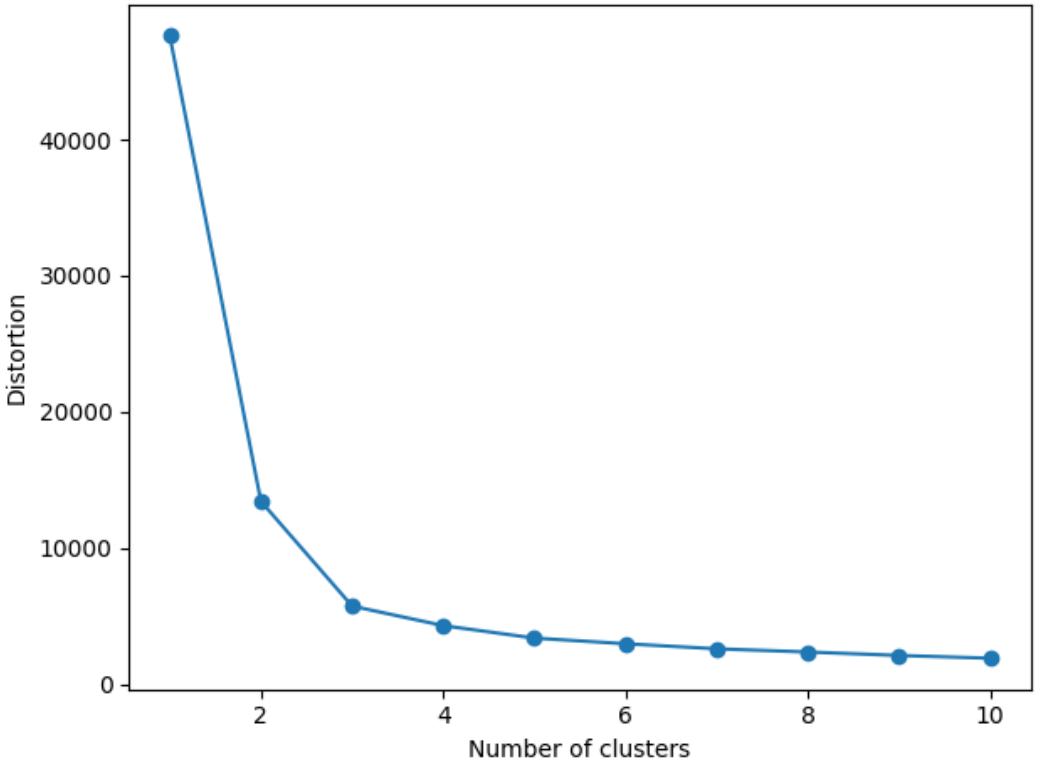
In [ ]:
```
plot_distortions(noisy_moons[0]) # k value is more difficult to distnguish, choose k=4
```



In [ ]:
```
plot_distortions(noisy_circles[0]) # k value is more difficult to distnguish, choose k=4
```



In [ ]:
```
plot_distortions(varied[0]) # select k=3
```



Hierarchical clustering

Hierarchical clustering is a general family of clustering algorithms that build nested clusters by merging or splitting them successively. This hierarchy of clusters is represented as a tree (or dendrogram). The root of the tree is the unique cluster that gathers all the samples, the leaves being the clusters with only one sample.

Agglomerative Clustering

The algorithm performs a hierarchical clustering using a bottom up approach. Agglomerative cluster has a “rich get richer” behavior that leads to uneven cluster sizes.

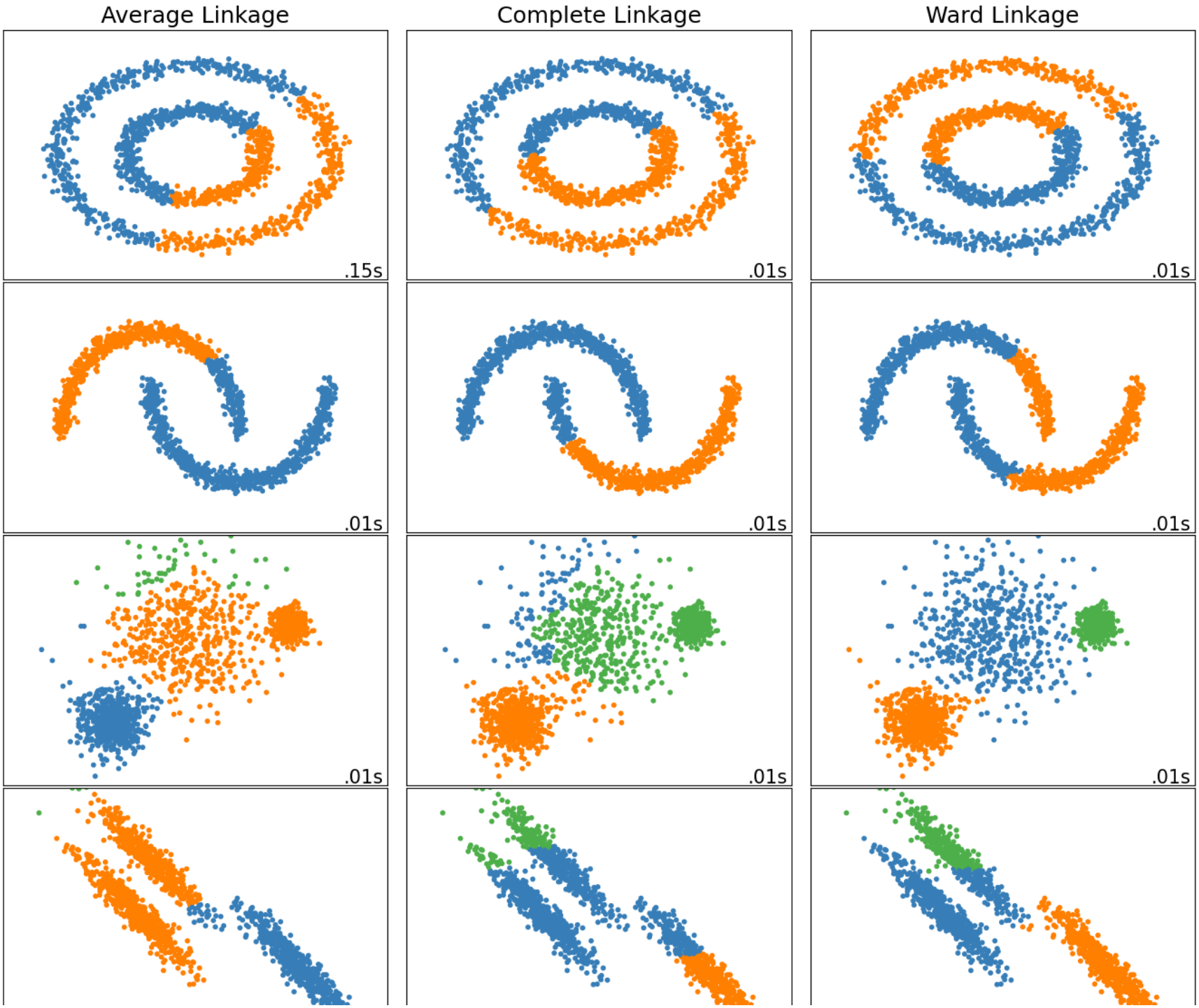
In [ ]:
```
from itertools import cycle, islice

# Set up cluster parameters
plt.figure(figsize=(9 * 1.3 + 2, 14.5))
plt.subplots_adjust(left=.02, right=.98, bottom=.001, top=.96, wspace=.05,
                    hspace=.01)
```

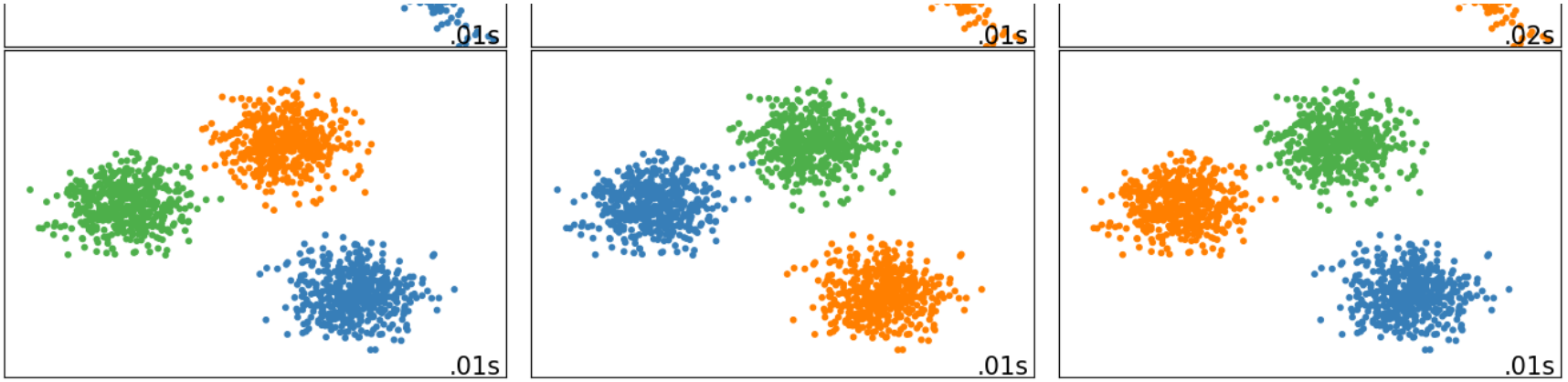
```
plot_num = 1

default_base = {'n_neighbors': 10,
                'n_clusters': 3}

datasets = [
    (noisy_circles, {'n_clusters': 2}),
    (noisy_moons, {'n_clusters': 2}),
    (varied, {'n_neighbors': 2}),
    (aniso, {'n_neighbors': 2}),
    ((X_blobs, y_blobs), {})]

for i_dataset, (dataset, algo_params) in enumerate(datasets):
    # update parameters with dataset-specific values
    params = default_base.copy()
    params.update(algo_params)

    X, y = dataset

    # normalize dataset for easier parameter selection
    X = StandardScaler().fit_transform(X)

    # ============
    # Create cluster objects
    # ============
    ward = AgglomerativeClustering(
        n_clusters=params['n_clusters'], linkage='ward')
    complete = AgglomerativeClustering(
        n_clusters=params['n_clusters'], linkage='complete')
    average = AgglomerativeClustering(
        n_clusters=params['n_clusters'], linkage='average')

    clustering_algorithms = (
        ('Average Linkage', average),
        ('Complete Linkage', complete),
        ('Ward Linkage', ward),
    )

    for name, algorithm in clustering_algorithms:
        t0 = time.time()

        # catch warnings related to kneighbors_graph
        with warnings.catch_warnings():
            warnings.filterwarnings(
                "ignore",
                message="the number of connected components of the " +
                "connectivity matrix is [0-9]{1,2}" +
                " > 1. Completing it to avoid stopping the tree early.",
                category=UserWarning)
```

```
            algorithm.fit(X)

        t1 = time.time()
        if hasattr(algorithm, 'labels_'):
            y_pred = algorithm.labels_.astype(np.int)
        else:
            y_pred = algorithm.predict(X)

        plt.subplot(len(datasets), len(clustering_algorithms), plot_num)
        if i_dataset == 0:
            plt.title(name, size=18)

        colors = np.array(list(islice(cycle(['#377eb8', '#ff7f00', '#4daf4a',
                                             '#f781bf', '#a65628', '#984ea3',
                                             '#999999', '#e41a1c', '#dede00']),
                                      int(max(y_pred) + 1))))
        plt.scatter(X[:, 0], X[:, 1], s=10, color=colors[y_pred])

        plt.xlim(-2.5, 2.5)
        plt.ylim(-2.5, 2.5)
        plt.xticks(())
        plt.yticks(())
        plt.text(.99, .01, ('%.2fs' % (t1 - t0)).lstrip('0'),
                 transform=plt.gca().transAxes, size=15,
                 horizontalalignment='right')
        plot_num += 1

plt.show()
```

```
/var/folders/hw/f8pnpzm163q0j3yww182vmgm0000gn/T/ipykernel_53833/1319219412.py:61: DeprecationWarning: `np.int` is a deprecated al
ias for the builtin `int`. To silence this warning, use `int` by itself. Doing this will not modify any behavior and is safe. When
replacing `np.int`, you may wish to use e.g. `np.int64` or `np.int32` to specify the precision. If you wish to review your current
use, check the release note link for additional information.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  y_pred = algorithm.labels_.astype(np.int)
/var/folders/hw/f8pnpzm163q0j3yww182vmgm0000gn/T/ipykernel_53833/1319219412.py:61: DeprecationWarning: `np.int` is a deprecated al
ias for the builtin `int`. To silence this warning, use `int` by itself. Doing this will not modify any behavior and is safe. When
replacing `np.int`, you may wish to use e.g. `np.int64` or `np.int32` to specify the precision. If you wish to review your current
use, check the release note link for additional information.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  y_pred = algorithm.labels_.astype(np.int)
/var/folders/hw/f8pnpzm163q0j3yww182vmgm0000gn/T/ipykernel_53833/1319219412.py:61: DeprecationWarning: `np.int` is a deprecated al
ias for the builtin `int`. To silence this warning, use `int` by itself. Doing this will not modify any behavior and is safe. When
replacing `np.int`, you may wish to use e.g. `np.int64` or `np.int32` to specify the precision. If you wish to review your current
use, check the release note link for additional information.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  y_pred = algorithm.labels_.astype(np.int)
/var/folders/hw/f8pnpzm163q0j3yww182vmgm0000gn/T/ipykernel_53833/1319219412.py:61: DeprecationWarning: `np.int` is a deprecated al
ias for the builtin `int`. To silence this warning, use `int` by itself. Doing this will not modify any behavior and is safe. When
replacing `np.int`, you may wish to use e.g. `np.int64` or `np.int32` to specify the precision. If you wish to review your current
use, check the release note link for additional information.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  y_pred = algorithm.labels_.astype(np.int)
/var/folders/hw/f8pnpzm163q0j3yww182vmgm0000gn/T/ipykernel_53833/1319219412.py:61: DeprecationWarning: `np.int` is a deprecated al
ias for the builtin `int`. To silence this warning, use `int` by itself. Doing this will not modify any behavior and is safe. When
replacing `np.int`, you may wish to use e.g. `np.int64` or `np.int32` to specify the precision. If you wish to review your current
use, check the release note link for additional information.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  y_pred = algorithm.labels_.astype(np.int)
/var/folders/hw/f8pnpzm163q0j3yww182vmgm0000gn/T/ipykernel_53833/1319219412.py:61: DeprecationWarning: `np.int` is a deprecated al
ias for the builtin `int`. To silence this warning, use `int` by itself. Doing this will not modify any behavior and is safe. When
replacing `np.int`, you may wish to use e.g. `np.int64` or `np.int32` to specify the precision. If you wish to review your current
use, check the release note link for additional information.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  y_pred = algorithm.labels_.astype(np.int)
/var/folders/hw/f8pnpzm163q0j3yww182vmgm0000gn/T/ipykernel_53833/1319219412.py:61: DeprecationWarning: `np.int` is a deprecated al
ias for the builtin `int`. To silence this warning, use `int` by itself. Doing this will not modify any behavior and is safe. When
replacing `np.int`, you may wish to use e.g. `np.int64` or `np.int32` to specify the precision. If you wish to review your current
use, check the release note link for additional information.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  y_pred = algorithm.labels_.astype(np.int)
/var/folders/hw/f8pnpzm163q0j3yww182vmgm0000gn/T/ipykernel_53833/1319219412.py:61: DeprecationWarning: `np.int` is a deprecated al
ias for the builtin `int`. To silence this warning, use `int` by itself. Doing this will not modify any behavior and is safe. When
replacing `np.int`, you may wish to use e.g. `np.int64` or `np.int32` to specify the precision. If you wish to review your current
use, check the release note link for additional information.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  y_pred = algorithm.labels_.astype(np.int)
/var/folders/hw/f8pnpzm163q0j3yww182vmgm0000gn/T/ipykernel_53833/1319219412.py:61: DeprecationWarning: `np.int` is a deprecated al
ias for the builtin `int`. To silence this warning, use `int` by itself. Doing this will not modify any behavior and is safe. When
```

```
replacing `np.int`, you may wish to use e.g. `np.int64` or `np.int32` to specify the precision. If you wish to review your current
use, check the release note link for additional information.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  y_pred = algorithm.labels_.astype(np.int)
/var/folders/hw/f8pnpzm163q0j3yww182vmgm0000gn/T/ipykernel_53833/1319219412.py:61: DeprecationWarning: `np.int` is a deprecated al
ias for the builtin `int`. To silence this warning, use `int` by itself. Doing this will not modify any behavior and is safe. When
replacing `np.int`, you may wish to use e.g. `np.int64` or `np.int32` to specify the precision. If you wish to review your current
use, check the release note link for additional information.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  y_pred = algorithm.labels_.astype(np.int)
/var/folders/hw/f8pnpzm163q0j3yww182vmgm0000gn/T/ipykernel_53833/1319219412.py:61: DeprecationWarning: `np.int` is a deprecated al
ias for the builtin `int`. To silence this warning, use `int` by itself. Doing this will not modify any behavior and is safe. When
replacing `np.int`, you may wish to use e.g. `np.int64` or `np.int32` to specify the precision. If you wish to review your current
use, check the release note link for additional information.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  y_pred = algorithm.labels_.astype(np.int)
/var/folders/hw/f8pnpzm163q0j3yww182vmgm0000gn/T/ipykernel_53833/1319219412.py:61: DeprecationWarning: `np.int` is a deprecated al
ias for the builtin `int`. To silence this warning, use `int` by itself. Doing this will not modify any behavior and is safe. When
replacing `np.int`, you may wish to use e.g. `np.int64` or `np.int32` to specify the precision. If you wish to review your current
use, check the release note link for additional information.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  y_pred = algorithm.labels_.astype(np.int)
/var/folders/hw/f8pnpzm163q0j3yww182vmgm0000gn/T/ipykernel_53833/1319219412.py:61: DeprecationWarning: `np.int` is a deprecated al
ias for the builtin `int`. To silence this warning, use `int` by itself. Doing this will not modify any behavior and is safe. When
replacing `np.int`, you may wish to use e.g. `np.int64` or `np.int32` to specify the precision. If you wish to review your current
use, check the release note link for additional information.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  y_pred = algorithm.labels_.astype(np.int)
/var/folders/hw/f8pnpzm163q0j3yww182vmgm0000gn/T/ipykernel_53833/1319219412.py:61: DeprecationWarning: `np.int` is a deprecated al
ias for the builtin `int`. To silence this warning, use `int` by itself. Doing this will not modify any behavior and is safe. When
replacing `np.int`, you may wish to use e.g. `np.int64` or `np.int32` to specify the precision. If you wish to review your current
use, check the release note link for additional information.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  y_pred = algorithm.labels_.astype(np.int)
/var/folders/hw/f8pnpzm163q0j3yww182vmgm0000gn/T/ipykernel_53833/1319219412.py:61: DeprecationWarning: `np.int` is a deprecated al
ias for the builtin `int`. To silence this warning, use `int` by itself. Doing this will not modify any behavior and is safe. When
replacing `np.int`, you may wish to use e.g. `np.int64` or `np.int32` to specify the precision. If you wish to review your current
use, check the release note link for additional information.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  y_pred = algorithm.labels_.astype(np.int)
```


Average Linkage
Complete Linkage
Ward Linkage
.15s
.01s
.01s
.01s
.01s
.01s
.01s
.01s
.01s

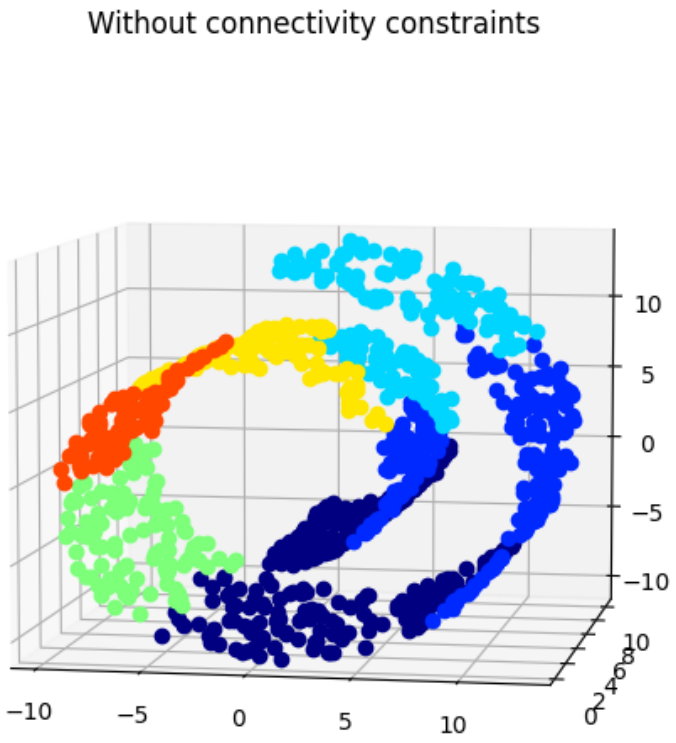


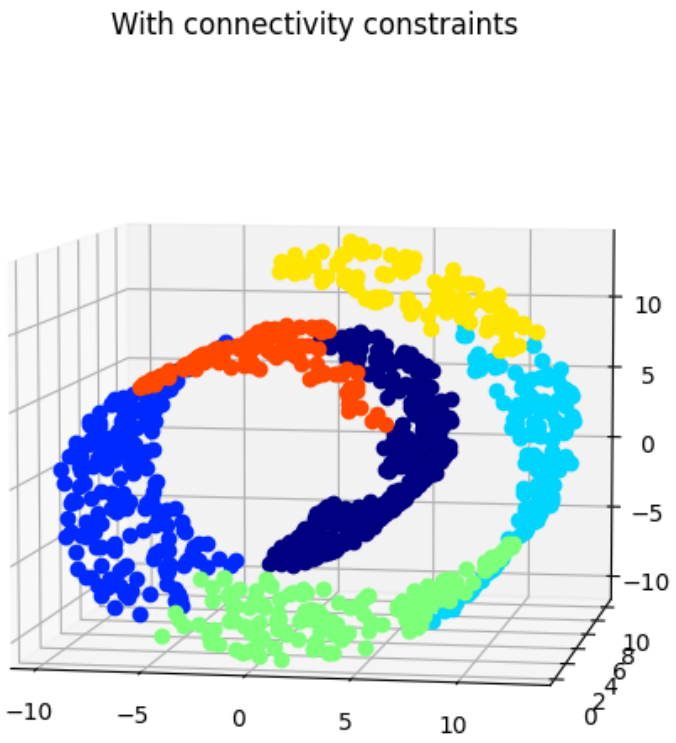
In [ ]:
```
import time as time
import numpy as np
import pylab as pl
import mpl_toolkits.mplot3d.axes3d as p3

###############################################################################
# Generate data (swiss roll dataset)
n_samples = 1000
noise = 0.05
X, _ = make_swiss_roll(n_samples=n_samples, noise=noise)
# Make it thinner
X[:, 1] *= .5

###############################################################################
# Compute clustering
print("Compute structured hierarchical clustering...")
st = time.time()
ward = AgglomerativeClustering(
        n_clusters=6, linkage='ward')
ward.fit(X)
if hasattr(algorithm, 'labels_'):
    label = ward.labels_.astype(np.int)
else:
    label = ward.predict(X)
print ("Elapsed time: {}".format(time.time() - st))

###############################################################################
# Plot result
fig = pl.figure()
ax = p3.Axes3D(fig)
ax.view_init(7, -80)
for l in np.unique(label):
    ax.plot3D(X[label == l, 0], X[label == l, 1], X[label == l, 2],
              'o', color=pl.cm.jet(np.float(l) / np.max(label + 1)))
```

```
pl.title('Without connectivity constraints')

###############################################################################
# Define the structure A of the data. Here a 10 nearest neighbors
from sklearn.neighbors import kneighbors_graph
connectivity = kneighbors_graph(X, n_neighbors=10)

###############################################################################
# Compute clustering
print("Compute structured hierarchical clustering...")
st = time.time()
ward = AgglomerativeClustering(
        n_clusters=6, linkage='ward', connectivity=connectivity)
ward.fit(X)
if hasattr(algorithm, 'labels_'):
    label = ward.labels_.astype(np.int)
else:
    label = ward.predict(X)
print ("Elapsed time: {}".format(time.time() - st))

###############################################################################
# Plot result
fig = pl.figure()
ax = p3.Axes3D(fig)
ax.view_init(7, -80)
for l in np.unique(label):
    ax.plot3D(X[label == l, 0], X[label == l, 1], X[label == l, 2],
              'o', color=pl.cm.jet(float(l) / np.max(label + 1)))
pl.title('With connectivity constraints')

pl.show()
```

```
Compute structured hierarchical clustering...
Elapsed time: 0.008779048919677734
Compute structured hierarchical clustering...
Elapsed time: 0.02070307731628418
```

```
/var/folders/hw/f8pnpzm163q0j3yww182vmgm0000gn/T/ipykernel_53833/1713572566.py:22: DeprecationWarning: `np.int` is a deprecated al
ias for the builtin `int`. To silence this warning, use `int` by itself. Doing this will not modify any behavior and is safe. When
replacing `np.int`, you may wish to use e.g. `np.int64` or `np.int32` to specify the precision. If you wish to review your current
use, check the release note link for additional information.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  label = ward.labels_.astype(np.int)
/var/folders/hw/f8pnpzm163q0j3yww182vmgm0000gn/T/ipykernel_53833/1713572566.py:30: MatplotlibDeprecationWarning: Axes3D(fig) addin
g itself to the figure is deprecated since 3.4. Pass the keyword argument auto_add_to_figure=False and use fig.add_axes(ax) to sup
press this warning. The default value of auto_add_to_figure will change to False in mpl3.5 and True values will no longer work in
3.6.  This is consistent with other Axes classes.
  ax = p3.Axes3D(fig)
/var/folders/hw/f8pnpzm163q0j3yww182vmgm0000gn/T/ipykernel_53833/1713572566.py:34: DeprecationWarning: `np.float` is a deprecated
alias for the builtin `float`. To silence this warning, use `float` by itself. Doing this will not modify any behavior and is saf
e. If you specifically wanted the numpy scalar type, use `np.float64` here.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  'o', color=pl.cm.jet(np.float(l) / np.max(label + 1)))
/var/folders/hw/f8pnpzm163q0j3yww182vmgm0000gn/T/ipykernel_53833/1713572566.py:50: DeprecationWarning: `np.int` is a deprecated al
ias for the builtin `int`. To silence this warning, use `int` by itself. Doing this will not modify any behavior and is safe. When
replacing `np.int`, you may wish to use e.g. `np.int64` or `np.int32` to specify the precision. If you wish to review your current
use, check the release note link for additional information.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  label = ward.labels_.astype(np.int)
/var/folders/hw/f8pnpzm163q0j3yww182vmgm0000gn/T/ipykernel_53833/1713572566.py:58: MatplotlibDeprecationWarning: Axes3D(fig) addin
g itself to the figure is deprecated since 3.4. Pass the keyword argument auto_add_to_figure=False and use fig.add_axes(ax) to sup
press this warning. The default value of auto_add_to_figure will change to False in mpl3.5 and True values will no longer work in
3.6.  This is consistent with other Axes classes.
  ax = p3.Axes3D(fig)
```


Without connectivity constraints
10
5
0
−5
−10
−10
−5
0
5
10
0
2
4
6
8
10



Question 1

Based on the code above, what is the difference between the two models? Which one performs better and why?

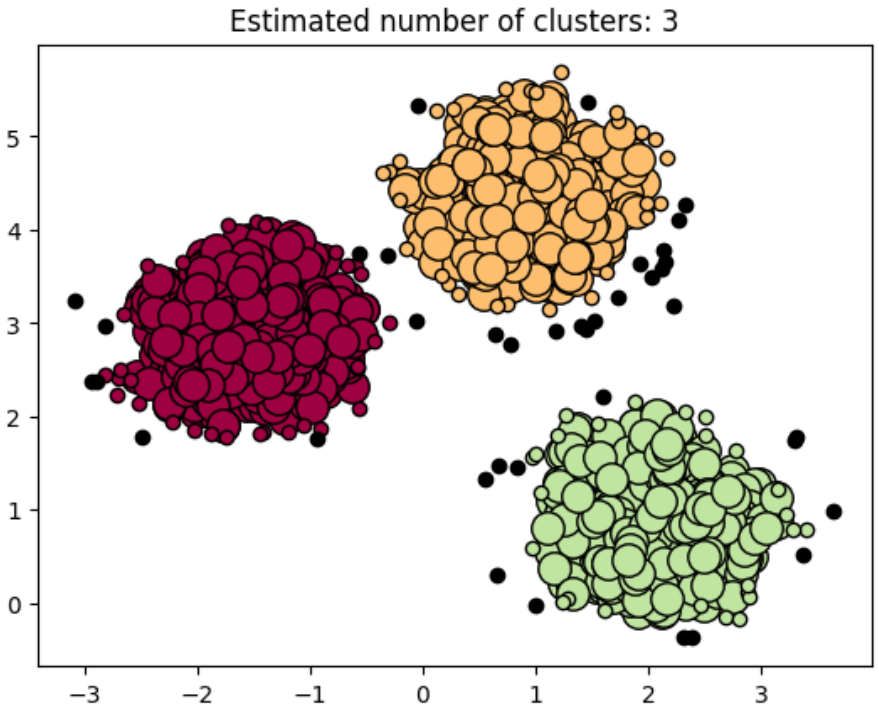
The apparent difference is the use of connectivity constraints. Issuing the constraint seems to create more defined clusters within the data and reduce, what seems to be, overlapping amonth clusters. The model using the connectivity constraint performs better. Using connectivity among each point in the dataset aids in the clustering of data by defining those points which are neighbors, as defined by a kneighbors_graph function, and will assign these neighbored points into a cluster. Defining these neighbors leads to improved interpretation of the clusters/dendrogram visuals as the neghboring/connectivity of the data is better defined. It can lead to reduced noise and/or outliers in the data. It also decrease comuptation time consumption of the model by conducting pairwise distance matrices/calculations on more well defined connectivity graph.

DBSCAN algorithm

As opposed to k-means algorithm which assumes that clusters are convex shape, DBSCAN views clusters as areas of high density separated by areas of low density.

In [ ]:
```
db = DBSCAN(eps=0.3, min_samples=10).fit(X_blobs)
core_samples_mask = np.zeros_like(db.labels_, dtype=bool)
core_samples_mask[db.core_sample_indices_] = True
labels = db.labels_

# Number of clusters in labels, ignoring noise if present.
n_clusters_ = len(set(labels)) - (1 if -1 in labels else 0)
n_noise_ = list(labels).count(-1)

print('Estimated number of clusters: %d' % n_clusters_)
print('Estimated number of noise points: %d' % n_noise_)
print("Homogeneity: %0.3f" % metrics.homogeneity_score(y_blobs, labels))
print("Completeness: %0.3f" % metrics.completeness_score(y_blobs, labels))
print("V-measure: %0.3f" % metrics.v_measure_score(y_blobs, labels))

# Black removed and is used for noise instead.
unique_labels = set(labels)
colors = [plt.cm.Spectral(each)
          for each in np.linspace(0, 1, len(unique_labels))]
for k, col in zip(unique_labels, colors):
    if k == -1:
        # Black used for noise.
        col = [0, 0, 0, 1]

    class_member_mask = (labels == k)

    xy = X_blobs[class_member_mask & core_samples_mask]
    plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col),
             markeredgecolor='k', markersize=14)

    xy = X_blobs[class_member_mask & ~core_samples_mask]
    plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col),
             markeredgecolor='k', markersize=6)

plt.title('Estimated number of clusters: %d' % n_clusters_)
plt.show()
```

```
Estimated number of clusters: 3
Estimated number of noise points: 38
Homogeneity: 0.976
Completeness: 0.902
V-measure: 0.937
```

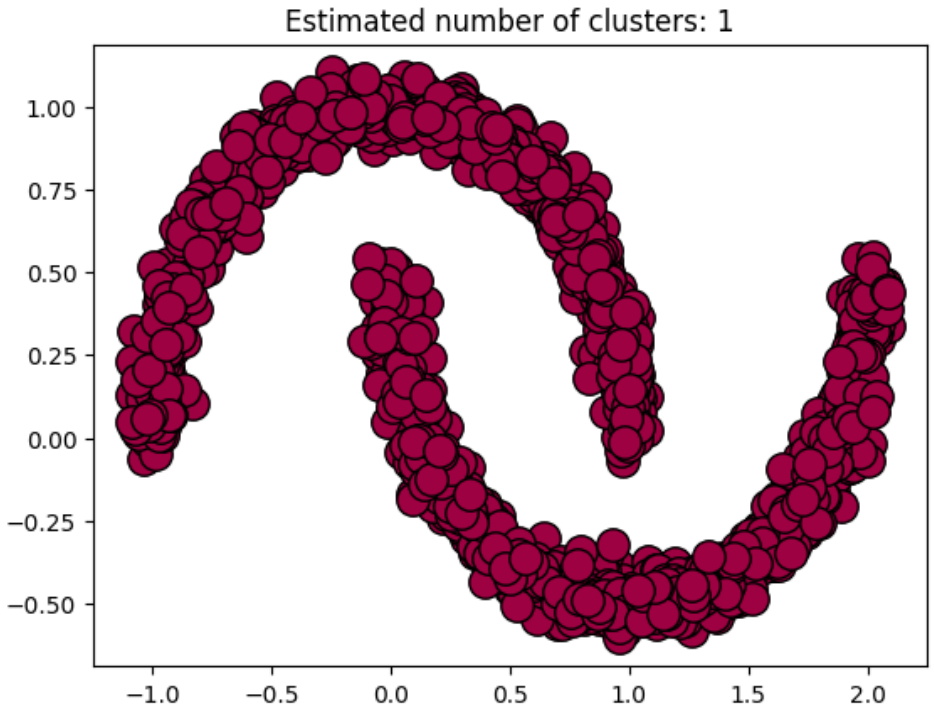


In [ ]:

Out[ ]:
```
array([[ 0.42702555, -0.84069836],
       [-0.3113457 , -0.46033825],
       [-0.49612248,  0.1908049 ],
       ...,
       [ 0.13924229,  0.44951696],
       [ 0.84246752,  0.5728653 ],
       [ 0.21238599, -1.00426767]])
```

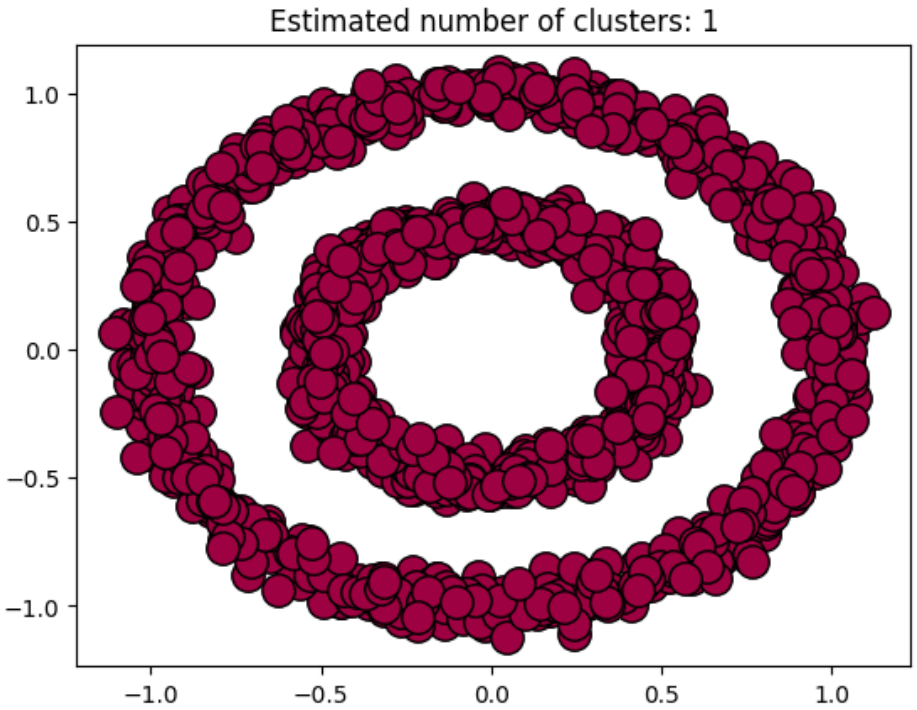
In [ ]:
```
db = DBSCAN(eps=0.3, min_samples=10).fit(noisy_moons[0])
core_samples_mask = np.zeros_like(db.labels_, dtype=bool)
core_samples_mask[db.core_sample_indices_] = True
labels = db.labels_

# Number of clusters in labels, ignoring noise if present.
n_clusters_ = len(set(labels)) - (1 if -1 in labels else 0)
n_noise_ = list(labels).count(-1)

# Black removed and is used for noise instead.
unique_labels = set(labels)
colors = [plt.cm.Spectral(each)
          for each in np.linspace(0, 1, len(unique_labels))]
```

```
for k, col in zip(unique_labels, colors):
    if k == -1:
        # Black used for noise.
        col = [0, 0, 0, 1]

    class_member_mask = (labels == k)

    xy = noisy_moons[0][class_member_mask & core_samples_mask]
    plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col),
             markeredgecolor='k', markersize=14)

    xy = noisy_moons[0][class_member_mask & ~core_samples_mask]
    plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col),
             markeredgecolor='k', markersize=6)

plt.title('Estimated number of clusters: %d' % n_clusters_)
plt.show()
```



In [ ]:
```
db = DBSCAN(eps=0.3, min_samples=10).fit(noisy_circles[0])

core_samples_mask = np.zeros_like(db.labels_, dtype=bool)
core_samples_mask[db.core_sample_indices_] = True
labels = db.labels_
```

```
# Number of clusters in labels, ignoring noise if present.
n_clusters_ = len(set(labels)) - (1 if -1 in labels else 0)
n_noise_ = list(labels).count(-1)

# Black removed and is used for noise instead.
unique_labels = set(labels)
colors = [plt.cm.Spectral(each)
          for each in np.linspace(0, 1, len(unique_labels))]
for k, col in zip(unique_labels, colors):
    if k == -1:
        # Black used for noise.
        col = [0, 0, 0, 1]

    class_member_mask = (labels == k)

    xy = noisy_circles[0][class_member_mask & core_samples_mask]
    plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col),
             markeredgecolor='k', markersize=14)

    xy = noisy_circles[0][class_member_mask & ~core_samples_mask]
    plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col),
             markeredgecolor='k', markersize=6)

plt.title('Estimated number of clusters: %d' % n_clusters_)
plt.show()
```

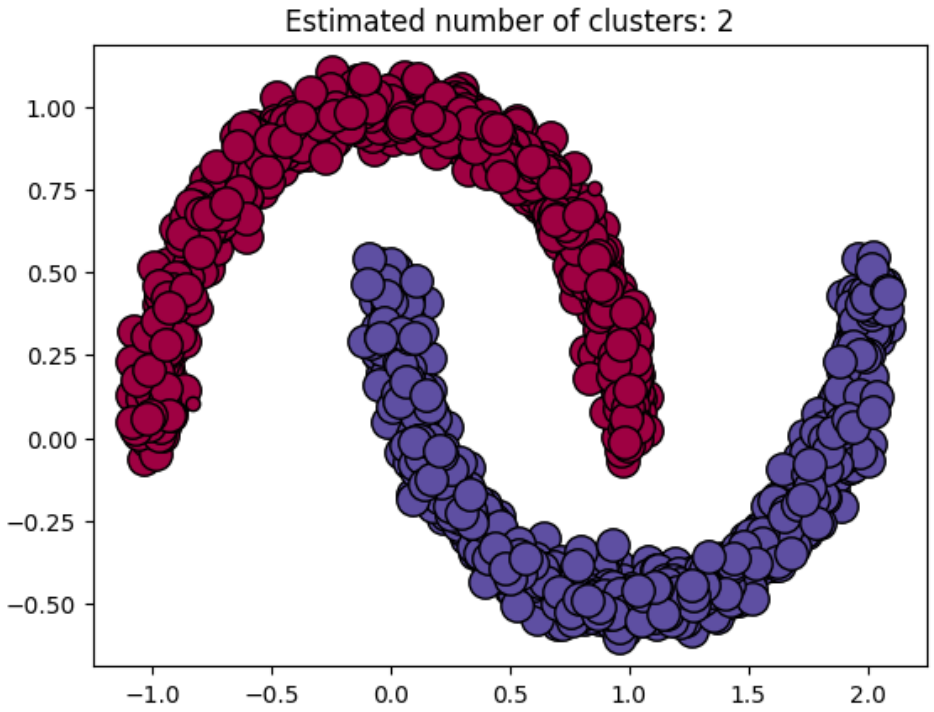


Question 2

eps and min_samples are two important parameters for the DBSCAN model. What are those two parameters? Tune the parameters of the model for noisy_circles and noisy_moon dataset to make it separate the clusters perfectly.

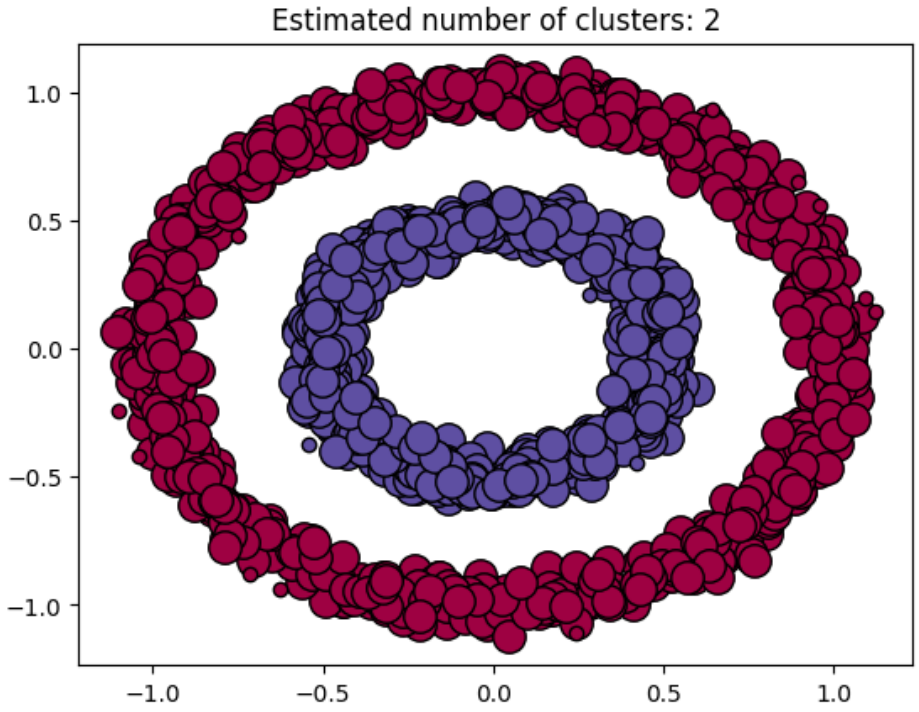
- **eps: This parameter is called the epsilon of the algorithm. It, essentially, determines how large the radius of a data point is that decides whether neighboring points will fall into the same "neighborhood" of that original point. If the radius is not large enough then it will cause much of the data to be seen as noise and will not cluster those points. However, if it is too large then some clusters in the data will start to converge with others causing incorrect labeling and increase in the capture of noise within the data.**
- **min_samples: defines the minimum number of samples within a grouping in order for that point to be considered a core point in the data. It works closely with eps in defining these core points as it is dependint on the eps radius, which captures points into it's neighborhood. If there are not enough points in the neighborhood, as defined by the min_samples, then it will not consider the core point and neighborhood as a cluster. The larger min_samples is, the harder it is for a neighborhood to become a cluster and increases the density requirements for a neighborhood to be defined as a cluster.**

In [ ]:
```
db = DBSCAN(eps=0.1, min_samples=5).fit(noisy_moons[0]) # best defined parameters
core_samples_mask = np.zeros_like(db.labels_, dtype=bool)
core_samples_mask[db.core_sample_indices_] = True
labels = db.labels_

# Number of clusters in labels, ignoring noise if present.
n_clusters_ = len(set(labels)) - (1 if -1 in labels else 0)
n_noise_ = list(labels).count(-1)

# Black removed and is used for noise instead.
unique_labels = set(labels)
colors = [plt.cm.Spectral(each)
          for each in np.linspace(0, 1, len(unique_labels))]
for k, col in zip(unique_labels, colors):
    if k == -1:
        # Black used for noise.
        col = [0, 0, 0, 1]

    class_member_mask = (labels == k)

    xy = noisy_moons[0][class_member_mask & core_samples_mask]
    plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col),
             markeredgecolor='k', markersize=14)

    xy = noisy_moons[0][class_member_mask & ~core_samples_mask]
    plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col),
             markeredgecolor='k', markersize=6)

plt.title('Estimated number of clusters: %d' % n_clusters_)
plt.show()
```



In [ ]:
```
db = DBSCAN(eps=0.1, min_samples=5).fit(noisy_circles[0]) # best defined parameters

core_samples_mask = np.zeros_like(db.labels_, dtype=bool)
core_samples_mask[db.core_sample_indices_] = True
labels = db.labels_

# Number of clusters in labels, ignoring noise if present.
n_clusters_ = len(set(labels)) - (1 if -1 in labels else 0)
n_noise_ = list(labels).count(-1)

# Black removed and is used for noise instead.
unique_labels = set(labels)
colors = [plt.cm.Spectral(each)
          for each in np.linspace(0, 1, len(unique_labels))]
for k, col in zip(unique_labels, colors):
    if k == -1:
        # Black used for noise.
        col = [0, 0, 0, 1]

    class_member_mask = (labels == k)

    xy = noisy_circles[0][class_member_mask & core_samples_mask]
    plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col),
```

```
            markeredgecolor='k', markersize=14)

    xy = noisy_circles[0][class_member_mask & ~core_samples_mask]
    plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col),
            markeredgecolor='k', markersize=6)

plt.title('Estimated number of clusters: %d' % n_clusters_)
plt.show()
```



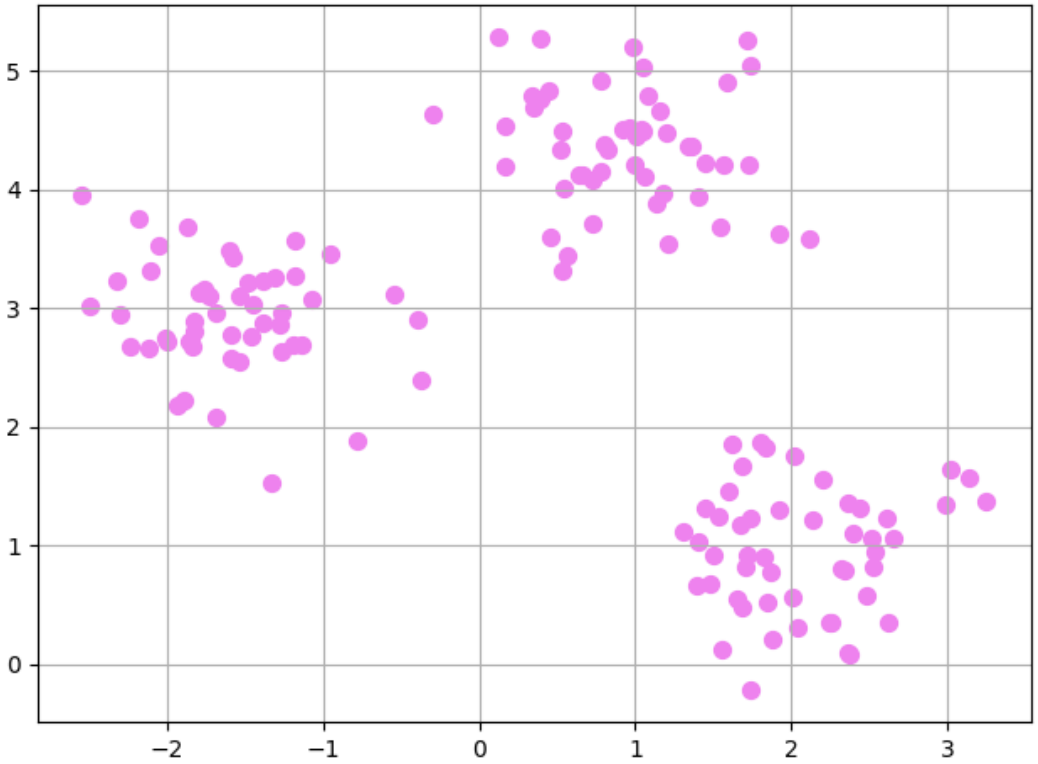
Mean-Shift Algorithm

Similiar to K-means, Mean shift algorithm locates the centroids of the clusters by shifting the points to density function maxima. Here is the procedure of the algorithm.

1. For each datapoint x ∈ X, find the neighbouring points N(x) of x.
2. For each datapoint x ∈ X, calculate the mean shift m(x).
3. For each datapoint x ∈ X, update x ← m(x).
4. Repeat 1. for n_iterations or until the points are almost not moving or not moving.

In [ ]:
```
import math

def euclid_distance(x, xi):
    return np.sqrt(np.sum((x - xi)**2))

def neighbourhood_points(X, x_centroid, distance = 5):
    eligible_X = []
    for x in X:
        distance_between = euclid_distance(x, x_centroid)
        # print('Evaluating: [%s vs %s] yield dist=%.2f' % (x, x_centroid, distance_between))
        if distance_between <= distance:
            eligible_X.append(x)
    return eligible_X

def gaussian_kernel(distance, bandwidth):
    val = (1/(bandwidth*math.sqrt(2*math.pi))) * np.exp(-0.5*((distance / bandwidth))**2)
    return val
```

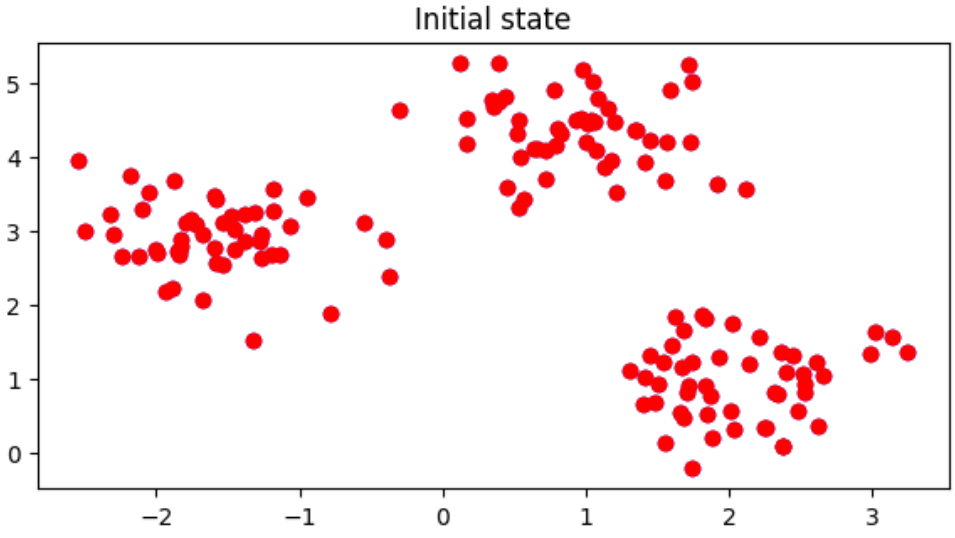
In [ ]:
```
n_samples = 150
X_blobs, y_blobs = make_blobs(n_samples= n_samples,
                  n_features=2,
                  centers=3,
                  cluster_std=0.5,
                  shuffle=True,
                  random_state=0)
plt.scatter(X_blobs[:, 0], X_blobs[:, 1], c='violet', marker='o', s=50)
plt.grid()
plt.tight_layout()
plt.show()
```

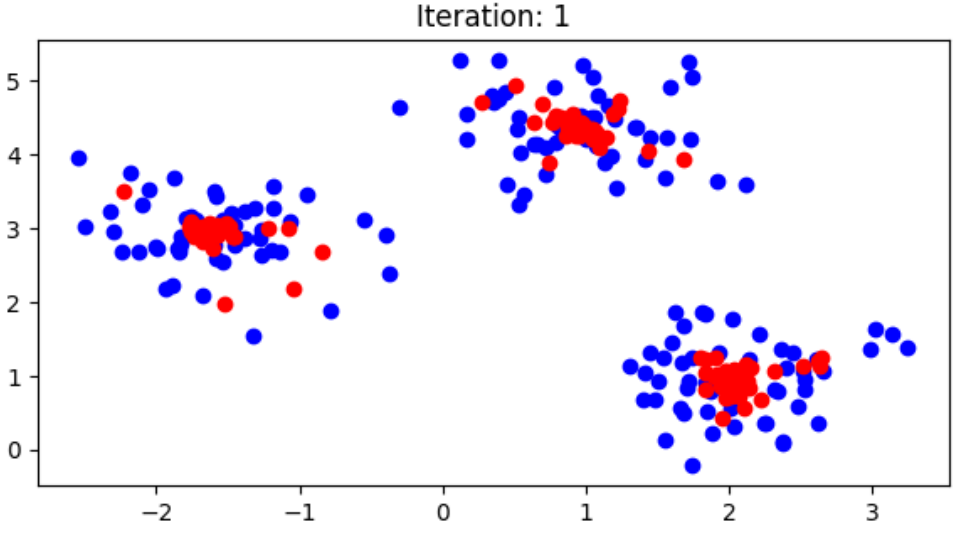


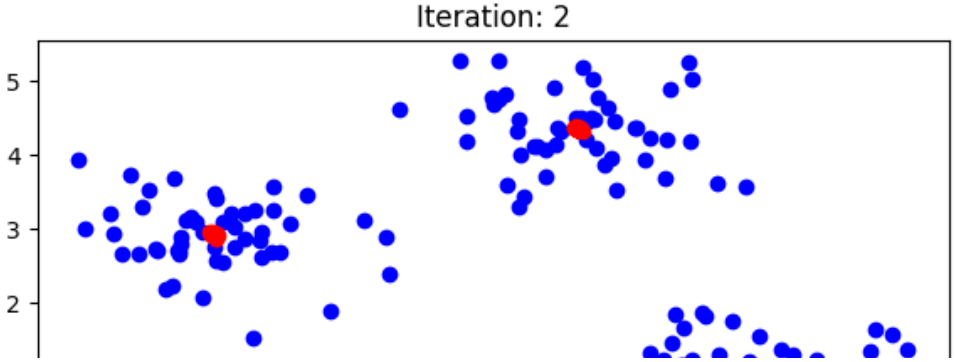
In [ ]:
```
original_data = X_blobs
# freeze the original points and make a copy
copy_points = np.copy(original_data)

look_distance = 1  # How far to look for neighbours.
kernel_bandwidth = 8  # Kernel parameter.
# a list to save the history move to do the visualization
past_X = []
n_iteration = 3
for i in range(n_iteration):

    for index, x in enumerate(copy_points):

        # for each datapoint x, find the neighbouring points N(x) of x
        neighbors = neighbourhood_points(copy_points, x, look_distance)
        # calculate the mean shift m(x)

        denominator = 0
        nominator = 0
        for n in neighbors:
            weight = gaussian_kernel(euclid_distance(x, n), kernel_bandwidth)
```

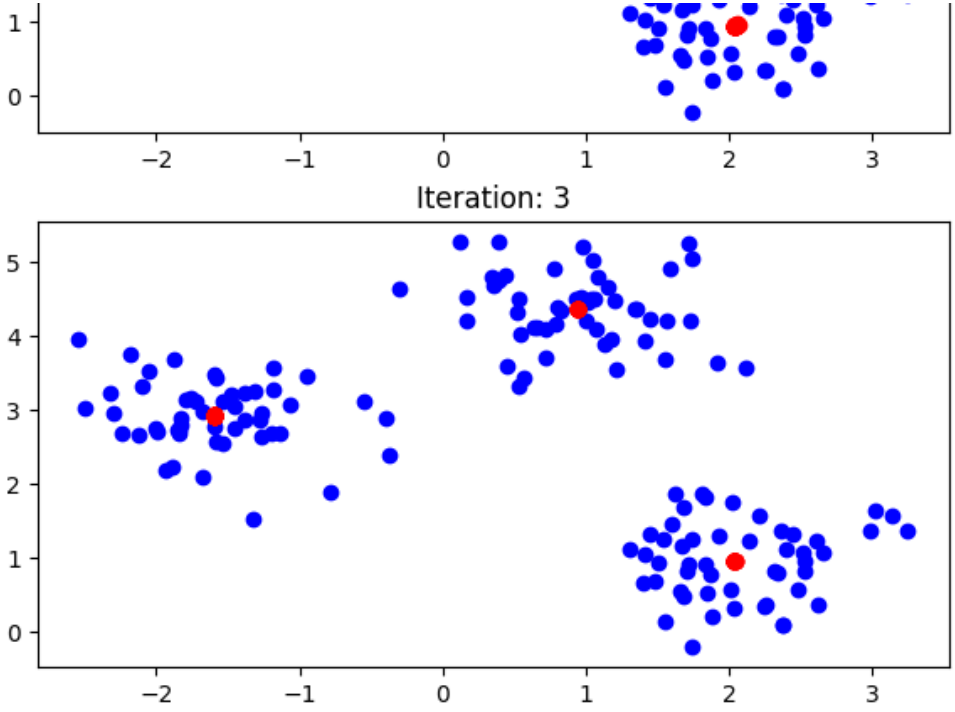
```
            nominator += weight * n
            denominator += weight
        new_x = nominator/(denominator+1e-4)
        copy_points[index] = new_x
    past_X.append(np.copy(copy_points))
```

In [ ]:
```
figure = plt.figure(1)
figure.set_size_inches((7, 20))
plt.subplot(n_iteration+2, 1, 1)
plt.title('Initial state')
plt.plot(original_data[:,0], original_data[:,1], 'bo')
plt.plot(original_data[:,0], original_data[:,1], 'ro')

for i in range(n_iteration):
    figure_index = i + 2
    plt.subplot(n_iteration+2, 1, figure_index)
    plt.title('Iteration: %d' % (figure_index - 1))
    plt.plot(original_data[:,0], original_data[:,1], 'bo')
    plt.plot(past_X[i][:,0], past_X[i][:,1], 'ro')
plt.show()
```


Initial state
5
4
3
2
1
0
−2
−1
0
1
2
3


Iteration: 1
5
4
3
2
1
0
−2
−1
0
1
2
3


Iteration: 2
5
4
3
2



Question 3

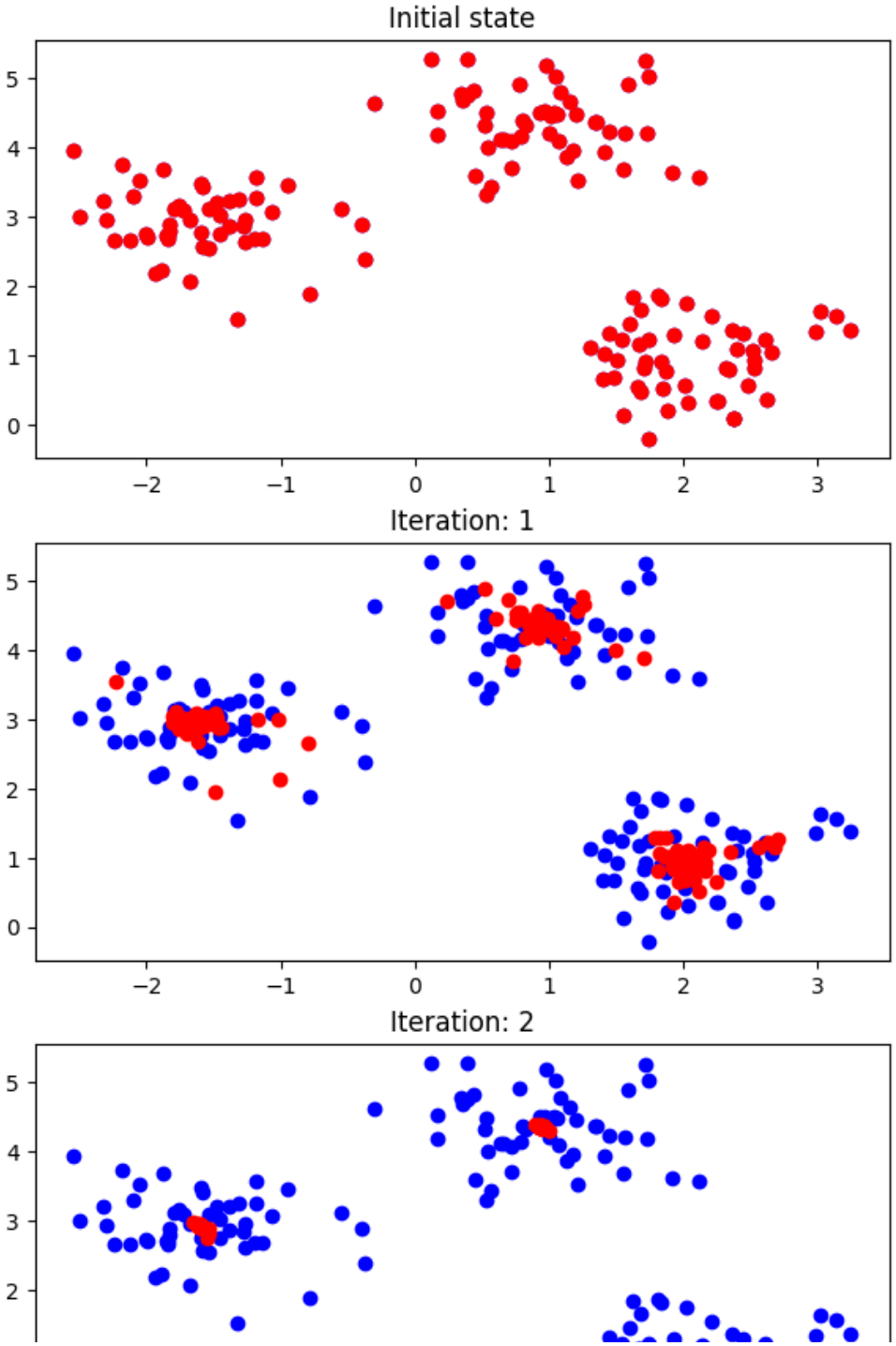
look_distance and kernel_bandwidth are two important parameters for mean shift algorithm. kernel_bandwidth simply determines the size of neighborhood over which the density will be computed. look_distance determines the region to look at when searching the surrounding neighbors. Tune these two parameters to make the cluster centroids exactly 3 for the above dataset. What is proper look_distance and kernel_bandwidth in your setting?

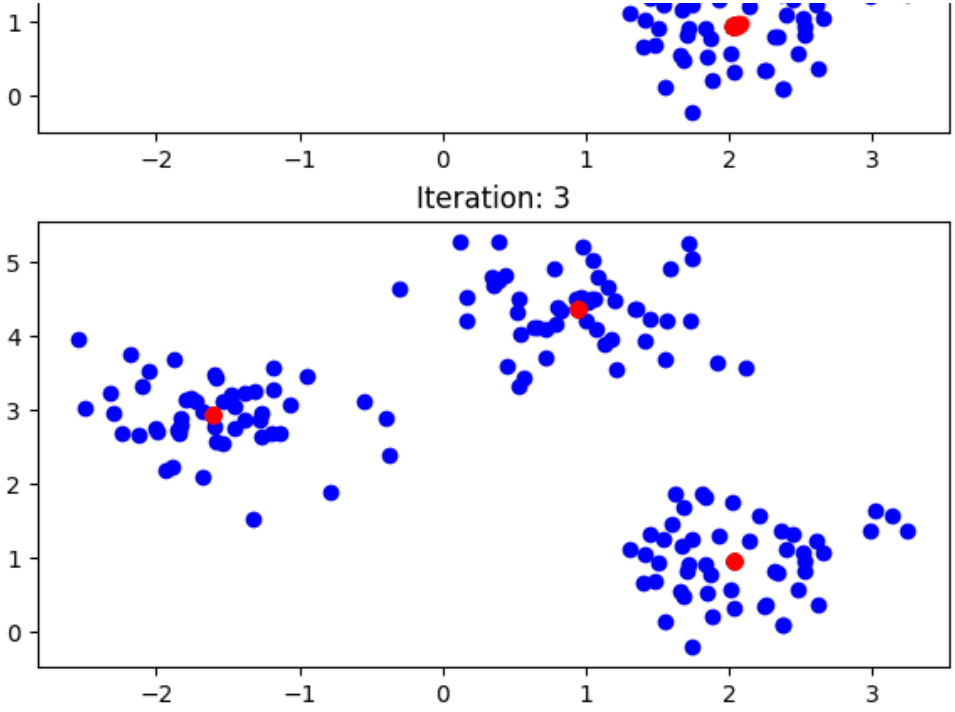
In [ ]:
```
original_data = X_blobs
# freeze the original points and make a copy
copy_points = np.copy(original_data)

look_distance = 1  # How far to look for neighbours.
kernel_bandwidth = 1  # Kernel parameter.
# a list to save the history move to do the visualization
past_X = []
n_iteration = 3
for i in range(n_iteration):

    for index, x in enumerate(copy_points):

        # for each datapoint x, find the neighbouring points N(x) of x
        neighbors = neighbourhood_points(copy_points, x, look_distance)
```

```
        # calculate the mean shift m(x)

        denominator = 0
        nominator = 0
        for n in neighbors:
            weight = gaussian_kernel(euclid_distance(x, n), kernel_bandwidth)
            nominator += weight * n
            denominator += weight
        new_x = nominator/(denominator+1e-4)
        copy_points[index] = new_x
    past_X.append(np.copy(copy_points))
```

In [ ]:

```
figure = plt.figure(1)
figure.set_size_inches((7, 20))
plt.subplot(n_iteration+2, 1, 1)
plt.title('Initial state')
plt.plot(original_data[:,0], original_data[:,1], 'bo')
plt.plot(original_data[:,0], original_data[:,1], 'ro')

for i in range(n_iteration):
    figure_index = i + 2
    plt.subplot(n_iteration+2, 1, figure_index)
    plt.title('Iteration: %d' % (figure_index - 1))
    plt.plot(original_data[:,0], original_data[:,1], 'bo')
    plt.plot(past_X[i][:,0], past_X[i][:,1], 'ro')
plt.show()
```


Initial state
Iteration: 1
Iteration: 2



It seems that increasing the look_distance more heavily influences the centroids while kernel_bandwidth does not infulence the centroids as much. However, the currently set values for look_distance and kernel_bandwidth at 1 and 8 distinctly places 3 centroids. I tested for these values at 1 and 1 and received, virtually, the same result

Question 4

Apply any of these algorithms to your favorite dataset. Possible applications of clustering algorihtm will include but not be limited to image segmentation and outlier detection.

Company Clustering Project

We will use the NYSE data set from Kaggle to determine which companies are essentially under a similar category according to their financials.

In [ ]:
```
import pandas as pd
import numpy as np
import copy
from sklearn.preprocessing import RobustScaler
```

```
from sklearn.decomposition import PCA
from sklearn.cluster import KMeans
from sklearn.pipeline import make_pipeline
import matplotlib.pyplot as plt

df = pd.read_csv('fundamentals.csv', index_col=0)
df.head()
```

Out[ ]:

	Ticker Symbol	Period Ending	Accounts Payable	Accounts Receivable	Add'l income/expense items	After Tax ROE	Capital Expenditures	Capital Surplus	Cash Ratio	Cash and Cash Equivalents	...	Total Current Assets	
0	AAL	2012-12-31	3.068000e+09	-222000000.0	-1.961000e+09	23.0	-1.888000e+09	4.695000e+09	53.0	1.330000e+09	...	7.072000e+09	9
1	AAL	2013-12-31	4.975000e+09	-93000000.0	-2.723000e+09	67.0	-3.114000e+09	1.059200e+10	75.0	2.175000e+09	...	1.432300e+10	1
2	AAL	2014-12-31	4.668000e+09	-160000000.0	-1.500000e+08	143.0	-5.311000e+09	1.513500e+10	60.0	1.768000e+09	...	1.175000e+10	1
3	AAL	2015-12-31	5.102000e+09	352000000.0	-7.080000e+08	135.0	-6.151000e+09	1.159100e+10	51.0	1.085000e+09	...	9.985000e+09	1
4	AAP	2012-12-29	2.409453e+09	-89482000.0	6.000000e+05	32.0	-2.711820e+08	5.202150e+08	23.0	5.981110e+08	...	3.184200e+09	2

5 rows × 78 columns

In [ ]:
```
df.shape
```

Out[ ]:
```
(1781, 78)
```

In [ ]:
```
# copy dataset to a new df
df_copy = copy.deepcopy(df)
df_copy.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 1781 entries, 0 to 1780
Data columns (total 75 columns):
 #   Column                                          Non-Null Count  Dtype
---  ------                                          --------------  -----
 0   Accounts Payable                                1781 non-null   float64
 1   Accounts Receivable                             1781 non-null   float64
 2   Add'l income/expense items                      1781 non-null   float64
 3   After Tax ROE                                   1781 non-null   float64
 4   Capital Expenditures                            1781 non-null   float64
 5   Capital Surplus                                 1781 non-null   float64
 6   Cash Ratio                                      1482 non-null   float64
 7   Cash and Cash Equivalents                       1781 non-null   float64
 8   Changes in Inventories                          1781 non-null   float64
 9   Common Stocks                                   1781 non-null   float64
 10  Cost of Revenue                                 1781 non-null   float64
 11  Current Ratio                                   1482 non-null   float64
 12  Deferred Asset Charges                          1781 non-null   float64
 13  Deferred Liability Charges                      1781 non-null   float64
 14  Depreciation                                    1781 non-null   float64
 15  Earnings Before Interest and Tax                1781 non-null   float64
 16  Earnings Before Tax                             1781 non-null   float64
 17  Effect of Exchange Rate                         1781 non-null   float64
 18  Equity Earnings/Loss Unconsolidated Subsidiary  1781 non-null   float64
 19  Fixed Assets                                    1781 non-null   float64
 20  Goodwill                                        1781 non-null   float64
 21  Gross Margin                                    1781 non-null   float64
 22  Gross Profit                                    1781 non-null   float64
 23  Income Tax                                      1781 non-null   float64
 24  Intangible Assets                               1781 non-null   float64
 25  Interest Expense                                1781 non-null   float64
 26  Inventory                                       1781 non-null   float64
 27  Investments                                     1781 non-null   float64
 28  Liabilities                                     1781 non-null   float64
 29  Long-Term Debt                                  1781 non-null   float64
 30  Long-Term Investments                           1781 non-null   float64
 31  Minority Interest                               1781 non-null   float64
 32  Misc. Stocks                                    1781 non-null   float64
 33  Net Borrowings                                  1781 non-null   float64
 34  Net Cash Flow                                   1781 non-null   float64
 35  Net Cash Flow-Operating                         1781 non-null   float64
 36  Net Cash Flows-Financing                        1781 non-null   float64
 37  Net Cash Flows-Investing                        1781 non-null   float64
 38  Net Income                                      1781 non-null   float64
 39  Net Income Adjustments                          1781 non-null   float64
 40  Net Income Applicable to Common Shareholders    1781 non-null   float64
 41  Net Income-Cont. Operations                     1781 non-null   float64
 42  Net Receivables                                 1781 non-null   float64
 43  Non-Recurring Items                             1781 non-null   float64
 44  Operating Income                                1781 non-null   float64
```

```
 45  Operating Margin                                        1781 non-null   float64
 46  Other Assets                                            1781 non-null   float64
 47  Other Current Assets                                    1781 non-null   float64
 48  Other Current Liabilities                               1781 non-null   float64
 49  Other Equity                                            1781 non-null   float64
 50  Other Financing Activities                              1781 non-null   float64
 51  Other Investing Activities                              1781 non-null   float64
 52  Other Liabilities                                       1781 non-null   float64
 53  Other Operating Activities                              1781 non-null   float64
 54  Other Operating Items                                   1781 non-null   float64
 55  Pre-Tax Margin                                          1781 non-null   float64
 56  Pre-Tax ROE                                             1781 non-null   float64
 57  Profit Margin                                           1781 non-null   float64
 58  Quick Ratio                                             1482 non-null   float64
 59  Research and Development                                1781 non-null   float64
 60  Retained Earnings                                       1781 non-null   float64
 61  Sale and Purchase of Stock                              1781 non-null   float64
 62  Sales, General and Admin.                               1781 non-null   float64
 63  Short-Term Debt / Current Portion of Long-Term Debt     1781 non-null   float64
 64  Short-Term Investments                                  1781 non-null   float64
 65  Total Assets                                            1781 non-null   float64
 66  Total Current Assets                                    1781 non-null   float64
 67  Total Current Liabilities                               1781 non-null   float64
 68  Total Equity                                            1781 non-null   float64
 69  Total Liabilities                                       1781 non-null   float64
 70  Total Liabilities & Equity                              1781 non-null   float64
 71  Total Revenue                                           1781 non-null   float64
 72  Treasury Stock                                          1781 non-null   float64
 73  Earnings Per Share                                      1562 non-null   float64
 74  Estimated Shares Outstanding                            1562 non-null   float64
dtypes: float64(75)
memory usage: 1.0 MB
```

In [ ]:
```
# only keeping features that report ratios as they are a combination of other features
df_feat_reduced = df_copy[['Cash Ratio', 'Current Ratio', 'Earnings Per Share', 'Quick Ratio', 'Earnings Per Share', 'Profit Margi
df_feat_reduced.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 1781 entries, 0 to 1780
Data columns (total 10 columns):
 #   Column              Non-Null Count  Dtype  
---  ------              --------------  -----  
 0   Cash Ratio          1482 non-null   float64
 1   Current Ratio       1482 non-null   float64
 2   Earnings Per Share  1562 non-null   float64
 3   Quick Ratio         1482 non-null   float64
 4   Earnings Per Share  1562 non-null   float64
 5   Profit Margin       1781 non-null   float64
 6   Pre-Tax ROE         1781 non-null   float64
 7   Pre-Tax Margin      1781 non-null   float64
 8   Operating Margin    1781 non-null   float64
 9   After Tax ROE       1781 non-null   float64
dtypes: float64(10)
memory usage: 153.1 KB
```

In [ ]:
```
df_feat_reduced.describe()
```

Out[ ]:

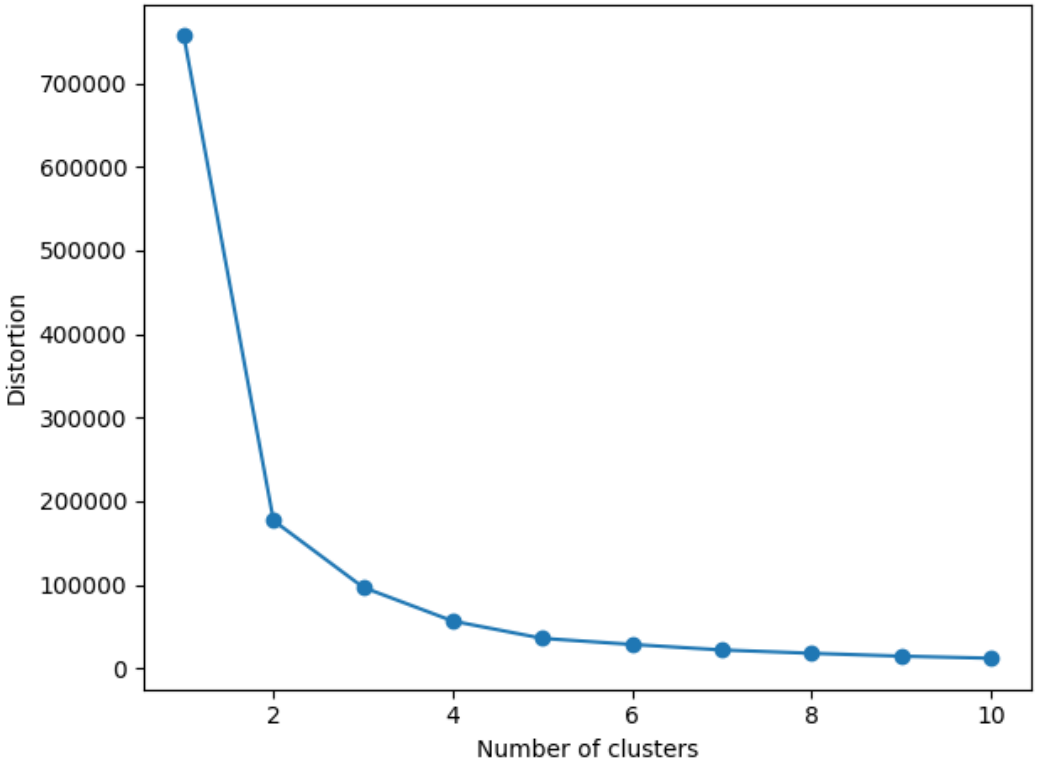
	Cash Ratio	Current Ratio	Earnings Per Share	Quick Ratio	Earnings Per Share	Profit Margin	Pre-Tax ROE	Pre-Tax Margin	Operating Margin	After Tax ROE
count	1482.000000	1482.000000	1562.000000	1482.000000	1562.000000	1781.000000	1781.000000	1781.000000	1781.000000	1781.000000
mean	74.457490	186.771255	3.353707	146.952767	3.353707	13.957889	59.644582	17.754632	18.177990	43.601348
std	102.298374	128.066801	4.695896	118.625127	4.695896	17.559655	330.447431	21.379605	20.504159	233.924028
min	0.000000	17.000000	-61.200000	10.000000	-61.200000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	17.000000	109.000000	1.590000	77.250000	1.590000	6.000000	13.000000	8.000000	9.000000	10.000000
50%	41.000000	152.000000	2.810000	115.000000	2.810000	10.000000	22.000000	14.000000	15.000000	16.000000
75%	90.000000	226.000000	4.590000	180.000000	4.590000	17.000000	36.000000	22.000000	23.000000	26.000000
max	1041.000000	1197.000000	50.090000	1197.000000	50.090000	369.000000	9089.000000	442.000000	437.000000	5789.000000

In [ ]:
```
# fill null values with 0
df_feat_reduced.fillna(0, inplace=True)
```

```
/var/folders/hw/f8pnpzm163q0j3yww182vmgm0000gn/T/ipykernel_83350/3824769268.py:2: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  df_feat_reduced.fillna(0, inplace=True)
```

In [ ]:
```
# set up the pipeline and plot distortions to choose k values
def plot_distortions(data):
```

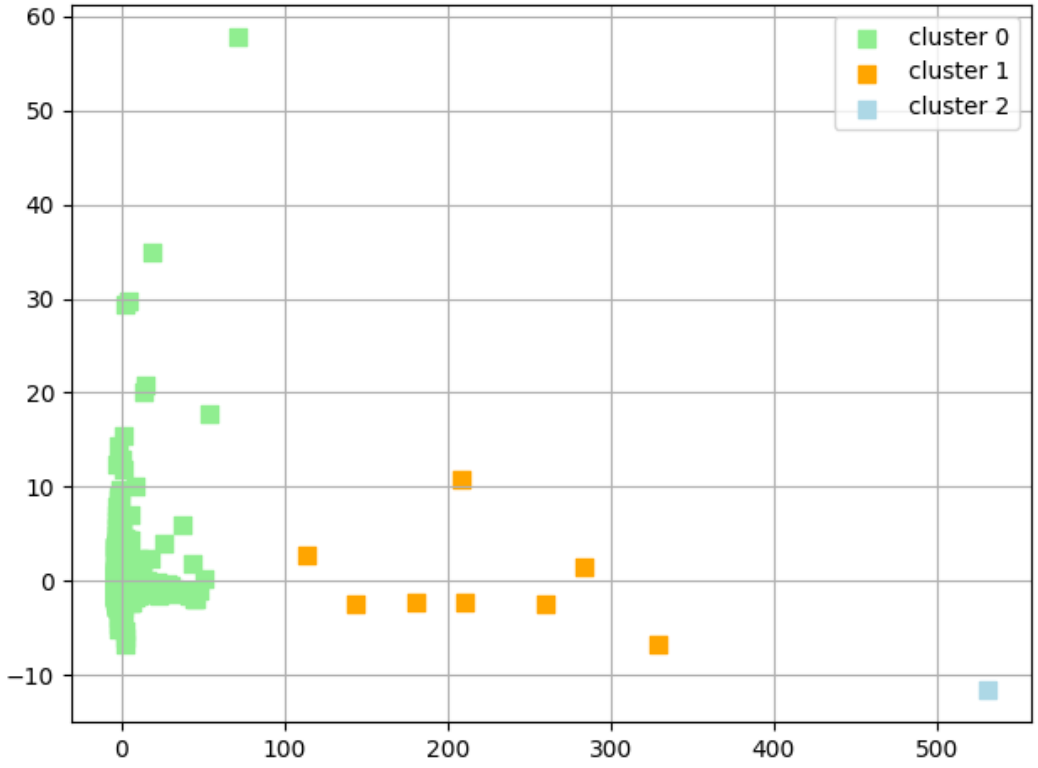
```
    distortions = []
    for i in range(1, 11):
        pipe_km = make_pipeline(
            RobustScaler(),
            PCA(n_components=2),
            KMeans(n_clusters=i,
                init='k-means++',
                n_init=10,
                max_iter=300,
                random_state=0)
        )
        pipe_km.fit(data)
        kmeans_step = pipe_km.named_steps['kmeans']
        distortions.append(kmeans_step.inertia_)
        # distortions.append(pipe_km.inertia_)
    plt.plot(range(1, 11), distortions, marker='o')
    plt.xlabel('Number of clusters')
    plt.ylabel('Distortion')
    plt.tight_layout()
    plt.show()

plot_distortions(df_feat_reduced)
```



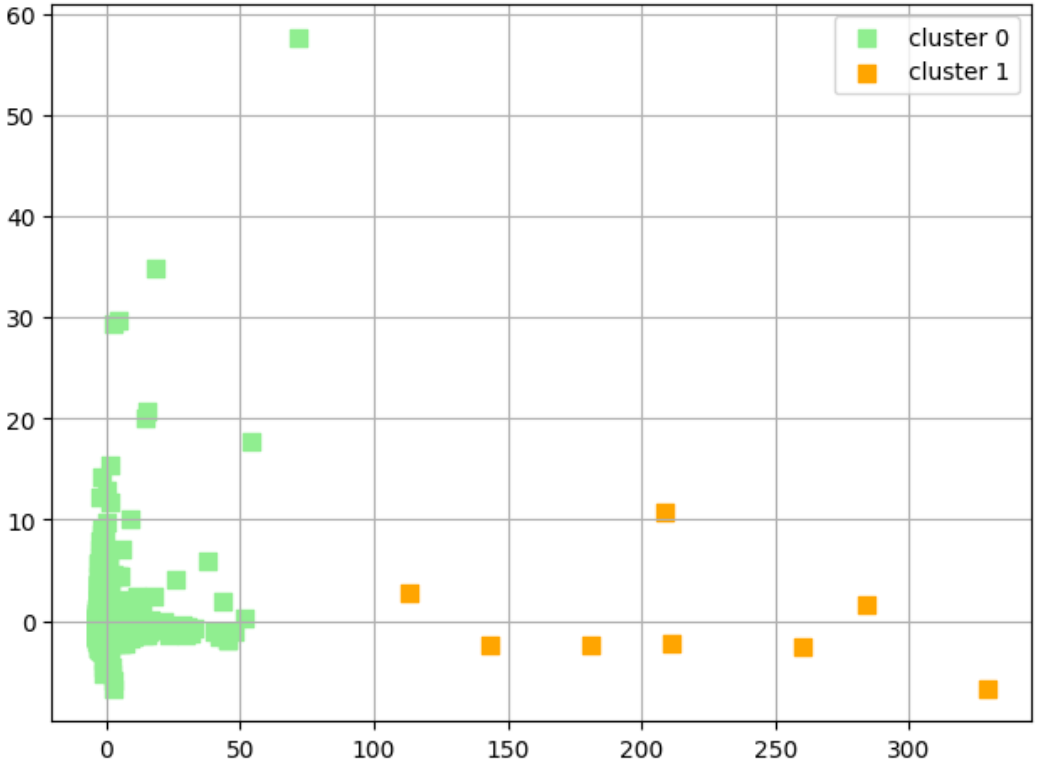
The lowest distortions seem to be closer to 2 or 3 clusters. Lets visualize the clusters to be sure

In [ ]:
```
def print_cluster(model, n_clusters, X):
    y_km = model.fit_predict(X)
    color_list = ['lightgreen', 'orange', 'lightblue', 'red', 'yellow', 'brown', 'cyan']
    for i in range(n_clusters):
        plt.scatter(X[y_km == i, 0],
        X[y_km == i, 1],
        s=50,
        c=color_list[i],
        marker='s',
        label='cluster ' + str(i))
    plt.legend()
    plt.grid()
    plt.tight_layout()
    plt.show()

rscaler = RobustScaler()
df_scaled = rscaler.fit_transform(df_feat_reduced)
pca = PCA(n_components=2)
```

```
df_pca = pca.fit_transform(df_scaled)

km = KMeans(
    n_clusters=3,
    init='k-means++',
    n_init=10,
    max_iter=300,
    random_state=0
)

print_cluster(km, 3, df_pca)
```



In [ ]:
```
km2 = KMeans(
    n_clusters=2,
    init='k-means++',
    n_init=10,
    max_iter=300,
    random_state=0
)

print_cluster(km, 2, df_pca)
```



k=2 seems to make the most sense in this data. Next steps would include a deeper look into these data points to determine what features make the greatest difference among the other features. It may be that a hierarchical clustering approach may be more suitable for this dataset as well. However, it may even make more sense to look at normalizing the data across the rows rather than accross the columns.