24-787: Machine Learning and Artificial Intelligence for Engineers
Ryan Wu
ID: weihuanw
Homework 11
Due: April 13, 2024

Concept Questions:

Problem 1
Q1: 5 points for Cluster Center 1, and 4 points for Cluster Center 2.

Q2: 2. Left, Right.

Problem 2
The natural number of clusters for this dataset is around 12. We can observe an elbow point where increasing the number of clusters is not informative or meaningful. The decrease in the sum of squared distances becomes relatively constant.

M10-L1 Problem 1

In this problem you will implement the K-Means algorithm from scratch, and use it to cluster two datasets: a "blob" shaped dataset with three classes, and a "moon" shaped dataset with two classes.

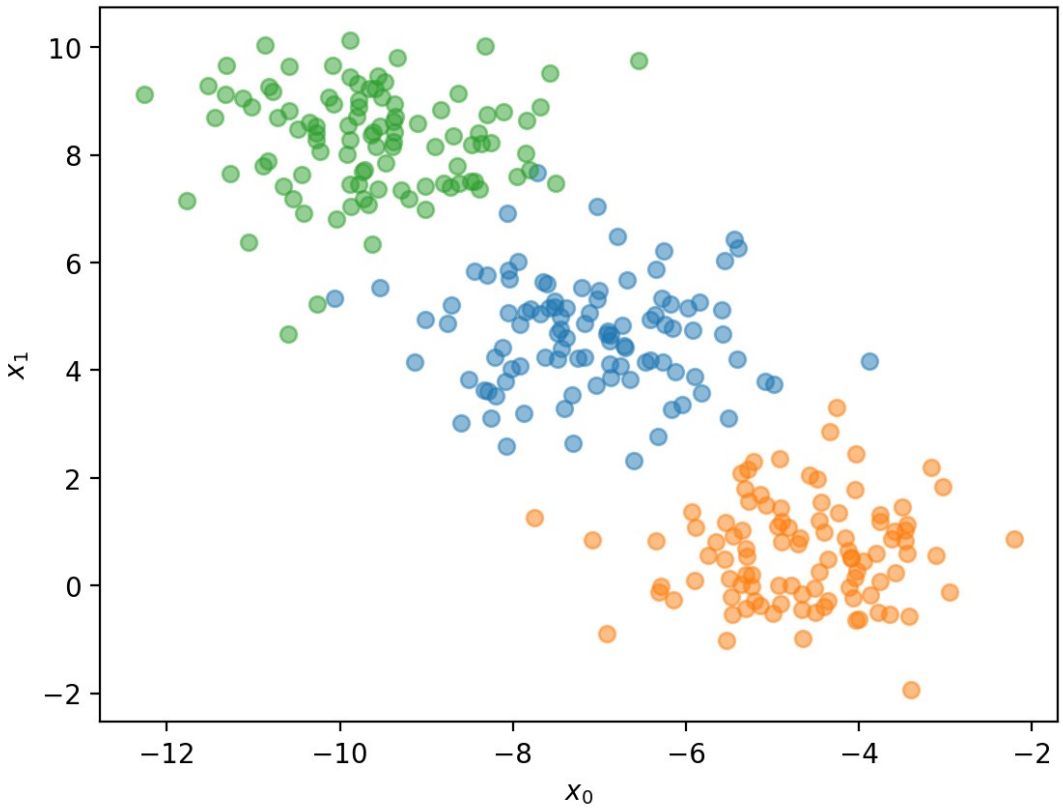
```
import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import make_blobs, make_moons

## DO NOT MODIFY
def plotter(x, y, labels = None, centers = None):
    fig = plt.figure(dpi = 200)
    for i in range(len(np.unique(y))):
        if labels is not None:
            plt.scatter(x[labels == i, 0], x[labels == i, 1], alpha =
0.5)
        else:
            plt.scatter(x[y == i, 0], x[y == i, 1], alpha = 0.5)
    if labels is not None:
        if (labels != y).any():
            plt.scatter(x[labels != y, 0], x[labels != y, 1], s = 100,
c = 'None', edgecolors = 'black', label = 'Misclassified Points')
    if centers is not None:
        plt.scatter(centers[:,0], centers[:,1], c = 'red', label =
'Cluster Centers')
    plt.xlabel('$x_0$')
    plt.ylabel('$x_1$')
    if labels is not None or centers is not None:
        plt.legend()
    plt.show()
```

We will use `sklearn.datasets.make_blobs()` to generate the dataset. The `random_state = 12` argument is used to ensure all students have the same data.

```
## DO NOT MODIFY
x, y = make_blobs(n_samples = 300, n_features = 2, random_state = 12)
```

Visualize the data using the `plotter(x,y)` function. You do not need to pass the `labels` or `centers` arguments

```
## YOUR CODE GOES HERE
# visualize the data
plotter(x, y)
```



Now we will begin to create our own K-Means function.

First you will write a function `find_cluster(point, centers)` which returns the index of the cluster center closest to the given point.

- `point` is a one dimensional numpy array containing the x_0 and x_1 coordinates of a single data point
- `centers` is a 3×2 numpy array containing the coordinates of the three cluster centers at any given iteration
- **return** the index of the closest cluster center

```
## FILL IN THE FOLLOWING FUNCTION
def find_cluster(point, centers):
    closet_center = np.argmin(np.linalg.norm(centers - point, axis =
1))
    return closet_center
```

Next, write a function `assign_labels(x, centers)` which will loop through all the points in `x` and use the `find_cluster()` function we just wrote to assign the label of the closest cluster center. Your function should return the labels

- `x` is a 300×2 numpy array containing the coordinates of all the points in the dataset
- `centers` is a 3×2 numpy array containing the coordinates of the three cluster centers at any given iteration

- **return** a one dimensional numpy array of length 300 containing the corresponding label for each point in `x`

```
## FILL IN THE FOLLOWING FUNCTION
def assign_labels(x, centers):
    labels = np.zeros(x.shape[0])
    for i in range(x.shape[0]):
        labels[i] = find_cluster(x[i], centers)
    return labels
```

Next, write a function `update_centers(x, labels)` which will compute the new cluster centers using the centroid of each cluster, provided all the points in `x` and their corresponding labels

- `x` is a 300×2 numpy array containing the coordinates of all the points in the dataset
- `labels` is a one dimensional numpy array of length 300 containing the corresponding label for each point in `x`
- **return** a 3×2 numpy array containing the coordinates of the three cluster centers

```
## FILL IN THE FOLLOWING FUNCTION
def update_centers(x, labels):
    cluster_centers = np.zeros((len(np.unique(labels)), x.shape[1]))
    for i in range(len(np.unique(labels))):
        cluster_centers[i] = np.mean(x[labels == i], axis = 0)
    return cluster_centers
```

Finally write a function `myKMeans(x, init_centers)` which will run the KMeans algorithm, provided all the points in `x` and the coordinates of the initial cluster centers in `init_centers`. Run the algorithm until there is no change in cluster membership in subsequent iterations. Your function should return both the `labels`, the labels of each point in x, and `centers`, the final coordinates of each of the cluster centers.

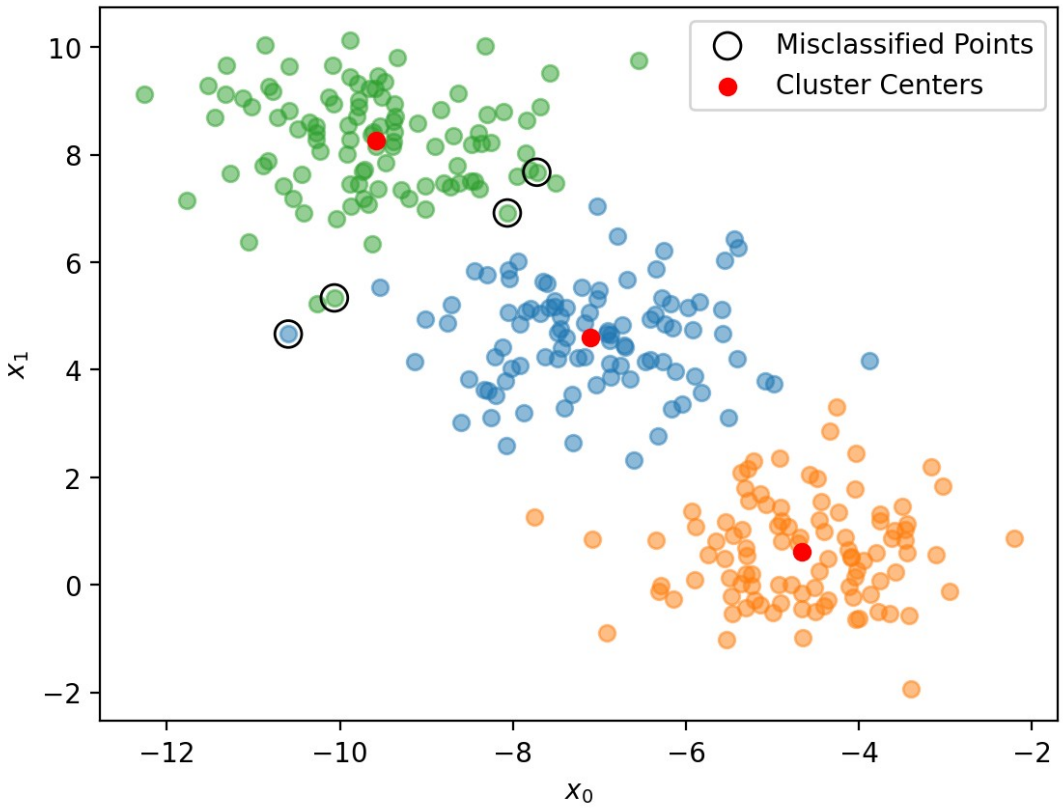
- `x` is a 300×2 numpy array containing the coordinates of all the points in the dataset
- `init_centers` is a 3×2 numpy array containing the coordinates of the three cluster centers provided to you
- **return** `labels` and `centers` as defined above

```
## FILL IN THE FOLLOWING FUNCTION
def myKMeans(x, init_centers):
    centers = init_centers
    labels = assign_labels(x, centers)
    new_centers = update_centers(x, labels)
    while not np.allclose(centers, new_centers):
        centers = new_centers
        labels = assign_labels(x, centers)
        new_centers = update_centers(x, labels)
    return labels, centers
```

Now use your `myKMeans()` function to cluster the provided data points `x` and set the initial cluster centers as `init_centers = np.array([[-5,5],[0,0],[-10,10]])`. Then use

the provided plotting function, `plotter(x,y,labels,centers)` to visualize your model's clustering.

```
## YOUR CODE GOES HERE
# visualize the model's clustering
init_centers = np.array([[-5,5], [0,0], [-10,10]])
labels, centers = myKMeans(x, init_centers)
plotter(x, y, labels, centers)
```



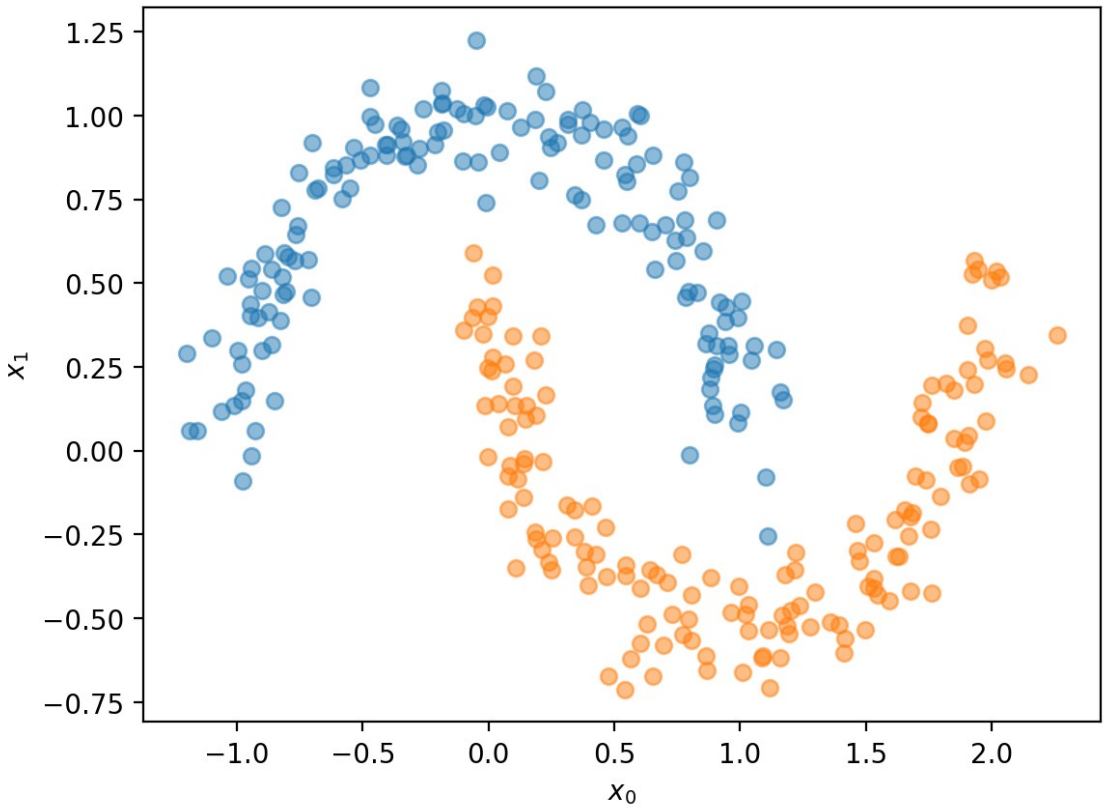
Moon Dataset

Now we will try using our `myKMeans()` function on a more challenging dataset, as generated below.

```
## DO NOT MODIFY
x,y = make_moons(n_samples = 300, noise = 0.1, random_state = 0)
```

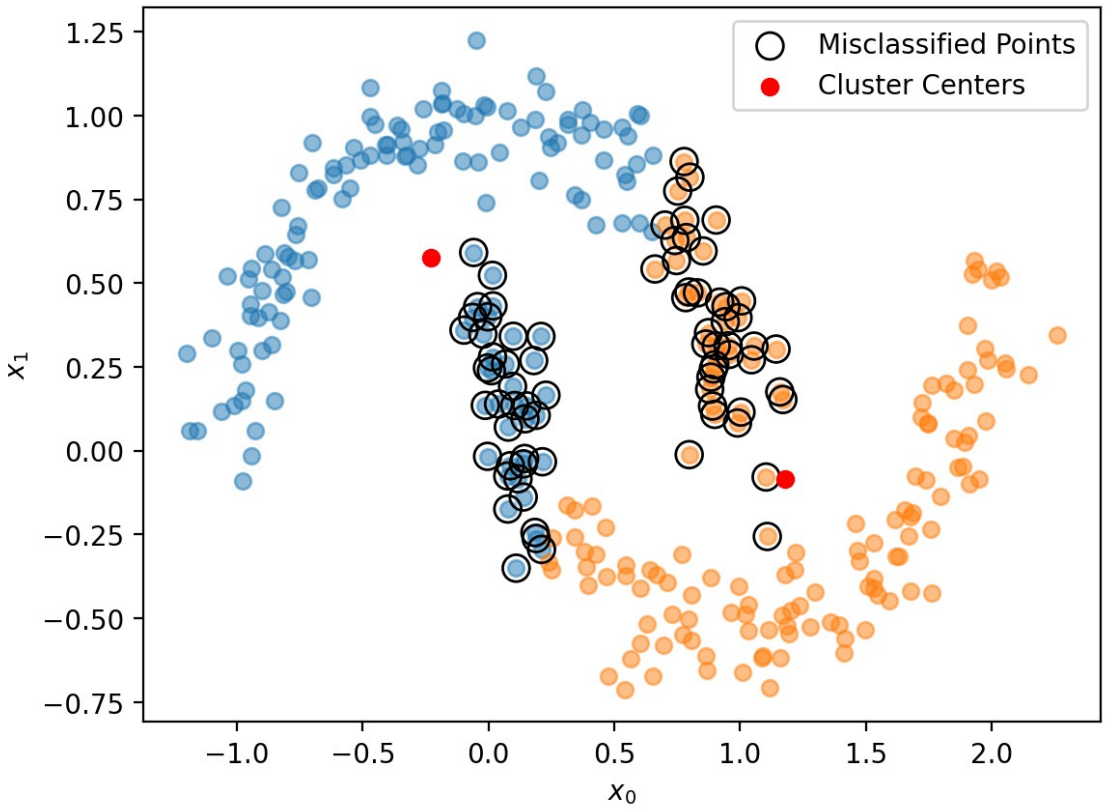
Visualize the data using the `plotter(x,y)` function.

```
## YOUR CODE GOES HERE
# visualize the data
plotter(x, y)
```



Using your `myKMeans()` function and `init_centers = np.array([[0,1],[1,-0.5]])` cluster the data, and visualize the results using `plotter(x,y,labels,centers)`.

```
## YOUR CODE GOES HERE
# visualize the model's clustering
init_centers = np.array([[0,1], [1,-0.5]])
labels, centers = myKMeans(x, init_centers)
plotter(x, y, labels, centers)
```


Misclassified Points
Cluster Centers
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
x_1
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
x_0

M10-L1 Problem 2: Solution

In this problem you will use the `sklearn` implementation of the K-Means algorithm to cluster the same two datasets from problem 1.

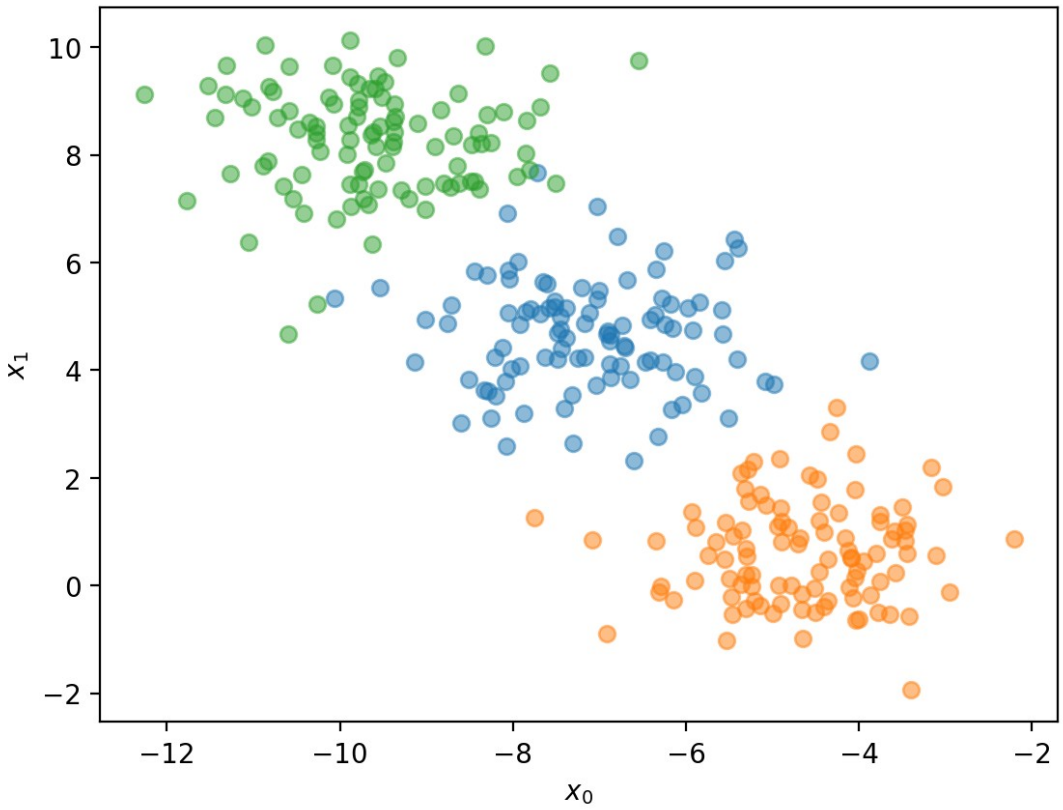
```
import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import make_blobs, make_moons
from sklearn.cluster import KMeans

## DO NOT MODIFY
def plotter(x, y, labels = None, centers = None):
    fig = plt.figure(dpi = 200)
    for i in range(len(np.unique(y))):
        if labels is not None:
            plt.scatter(x[labels == i, 0], x[labels == i, 1], alpha =
0.5)
        else:
            plt.scatter(x[y == i, 0], x[y == i, 1], alpha = 0.5)
    if labels is not None:
        if (labels != y).any():
            plt.scatter(x[labels != y, 0], x[labels != y, 1], s = 100,
c = 'None', edgecolors = 'black', label = 'Misclassified Points')
    if centers is not None:
        plt.scatter(centers[:,0], centers[:,1], c = 'red', label =
'Cluster Centers')
    plt.xlabel('$x_0$')
    plt.ylabel('$x_1$')
    if labels is not None or centers is not None:
        plt.legend()
    plt.show()
```

We will use `sklearn.datasets.make_blobs()` to generate the dataset. The `random_state = 12` argument is used to ensure all students have the same data.

```
## DO NOT MODIFY
x, y = make_blobs(n_samples = 300, n_features = 2, random_state = 12)
```

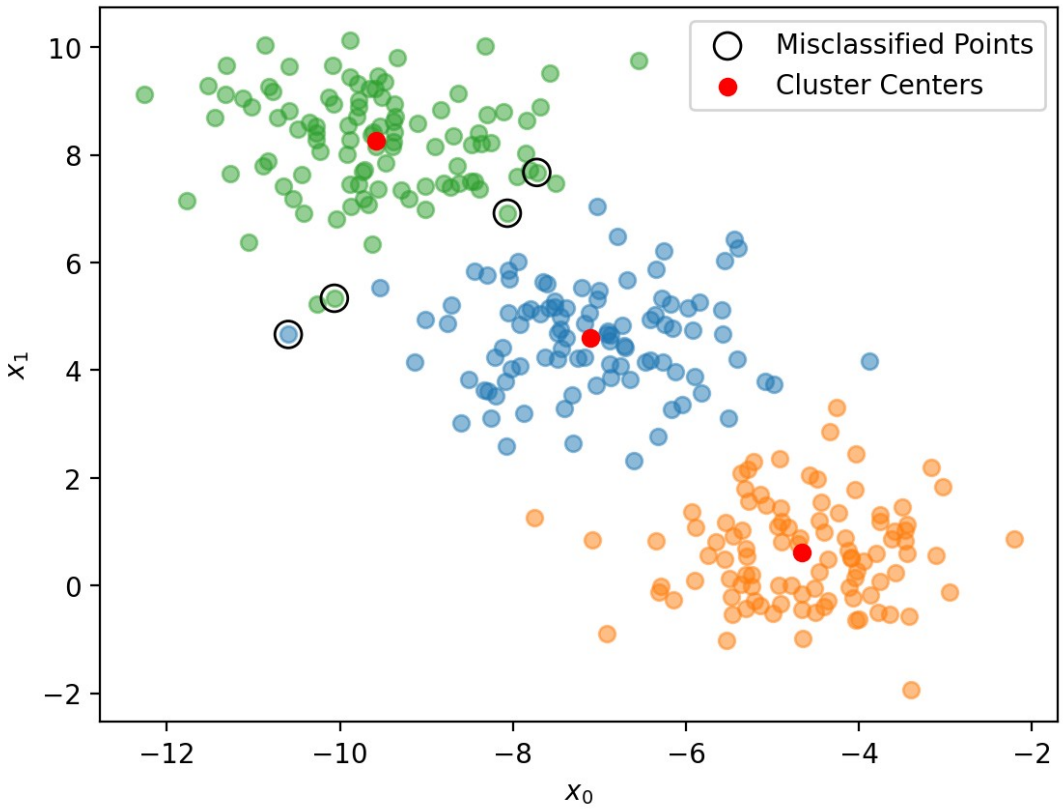
Visualize the data using the `plotter(x,y)` function. You do not need to pass the `labels` or `centers` arguments

```
## YOUR CODE GOES HERE
# visualize the data
plotter(x, y)
```



Now you will use `sklearn.cluster.KMeans()` to cluster the provided data points `x`. For the `KMeans()` function to perform identically to our implementation, we need to provide the same initial clusters with the `init` argument. The cluster centers should be initialized as `np.array([[-5,5],[0,0],[-10,10]])`, and you can additionally pass in the `n_init = 1` argument to silence a runtime warning that comes from passing explicit initial cluster centers. Then plot the results using the provided `plotter(x,y,labels,centers)` function.

```
## YOUR CODE GOES HERE
# visualize the model's clustering
init_center = np.array([[-5, 5], [0, 0], [-10, 10]])
kmeans = KMeans(n_clusters = 3, init = init_center, random_state = 12)
kmeans.fit(x)
plotter(x, y, kmeans.labels_, kmeans.cluster_centers_)
```



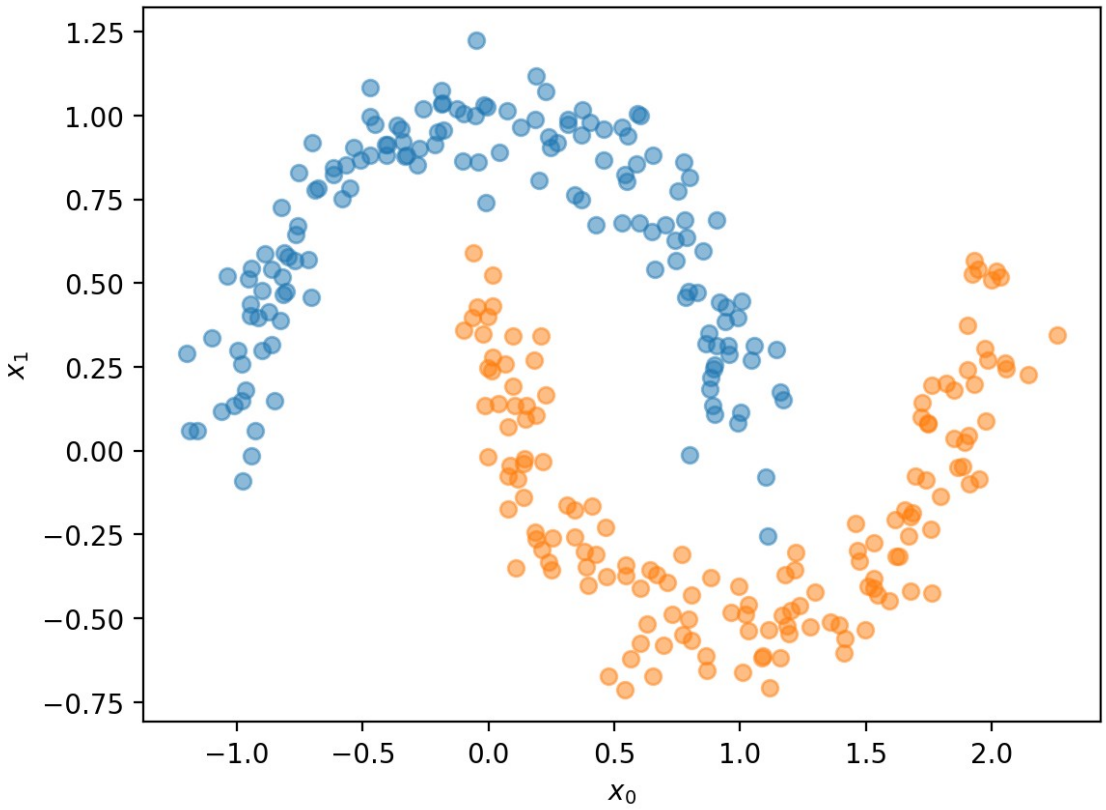
Moon Dataset

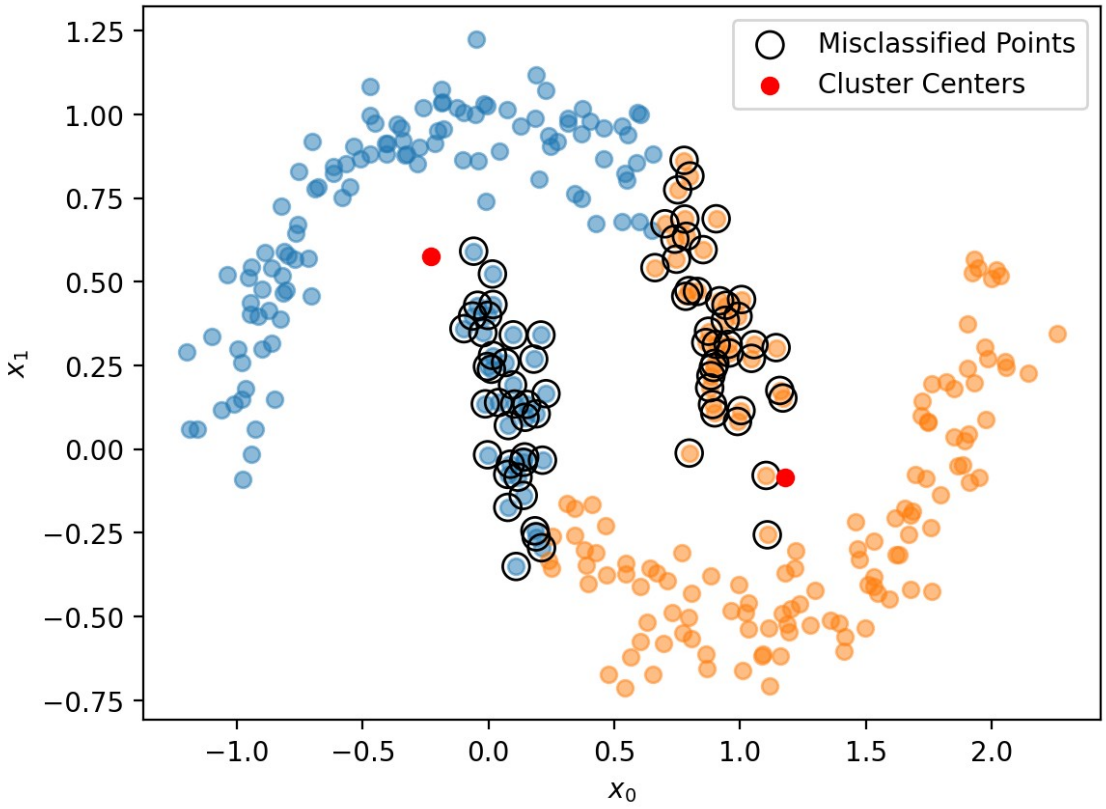
Now we will try using the `sklearn.cluster.KMeans()` function on the moons dataset from problem 1.

```
## DO NOT MODIFY
x,y = make_moons(n_samples = 300, noise = 0.1, random_state = 0)
```

Using the same initial cluster centers from problem 1, namely, `np.array([[0,1],[1,-0.5]])`, cluster the moons datasets and plot the results using the provided `plotter(x,y,labels,centers)` function.

```
## YOUR CODE GOES HERE
# visualize the orginal data
plotter(x, y)

# visualize the model's clustering
init_center = np.array([[0, 1], [1, -0.5]])
kmeans = KMeans(n_clusters = 2, init = init_center, random_state = 12)
kmeans.fit(x)
plotter(x, y, kmeans.labels_, kmeans.cluster_centers_)
```


1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
x_1
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
x_0


Misclassified Points
Cluster Centers
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
x_1
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
x_0

Discussion

How do the results of your hand coded implementation of the K-Means algorithm compare to the `sklearn` implementation? If there is any discrepancy between the results, provide your reasoning why.

From the results, my hand-coded implementation and the sklearn implementation have no discrepancy between the results.

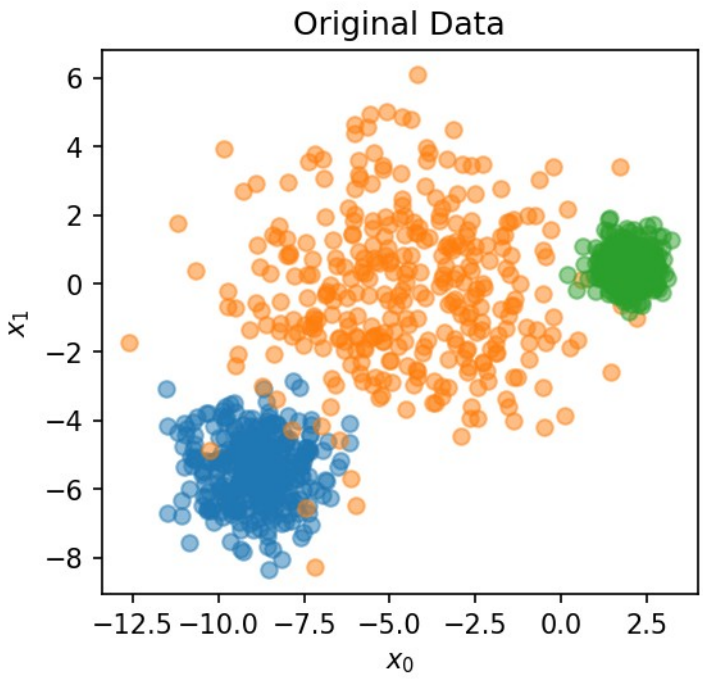
M11-L1 Problem 3

In this problem you will use the `sklearn` implementation of hierarchical clustering with three different linkage criteria `('single', 'complete', 'average')` to clusters two datasets: a "blob" shaped dataset with three classes, and a concentric circle dataset with two classes.

```
import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import make_blobs, make_circles
from sklearn.cluster import AgglomerativeClustering

## DO NOT MODIFY
def plotter(x, labels = None, ax = None, title = None):
    if ax is None:
        _, ax = plt.subplots(dpi = 150, figsize = (4,4))
        flag = True
    else:
        flag = False
    for i in range(len(np.unique(labels))):
        ax.scatter(x[labels == i, 0], x[labels == i, 1], alpha = 0.5)
    ax.set_xlabel('$x_0$')
    ax.set_ylabel('$x_1$')
    ax.set_aspect('equal')
    if title is not None:
        ax.set_title(title)
    if flag:
        plt.show()
    else:
        return ax
```

First we will consider the "blob" dataset, generated below. Visualize the data using the provided `plotter(x, labels)` function.

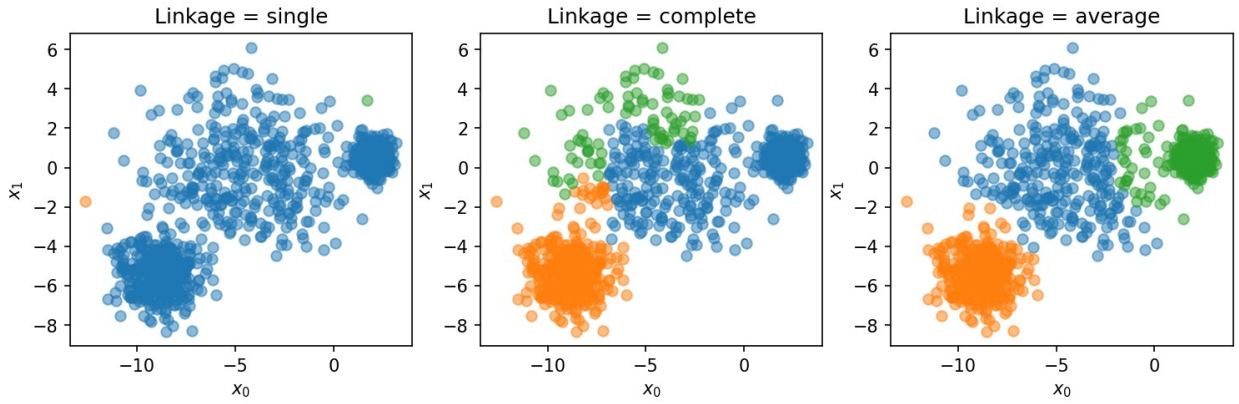
```
## DO NOT MODIFY
x, labels = make_blobs(n_samples = 1000, cluster_std=[1.0, 2.5, 0.5],
random_state = 170)

## YOUR CODE GOES HERE
# visualize the data
plotter(x, labels, title = 'Original Data')
```



Using the `AgglomerativeClustering()` function, generate 3 side-by-side plots using `plt.subplots()` and the provided `plotter(x, labels, ax, title)` function to visualize the results of the following three linkage criteria `['single', 'complete', 'average']`.

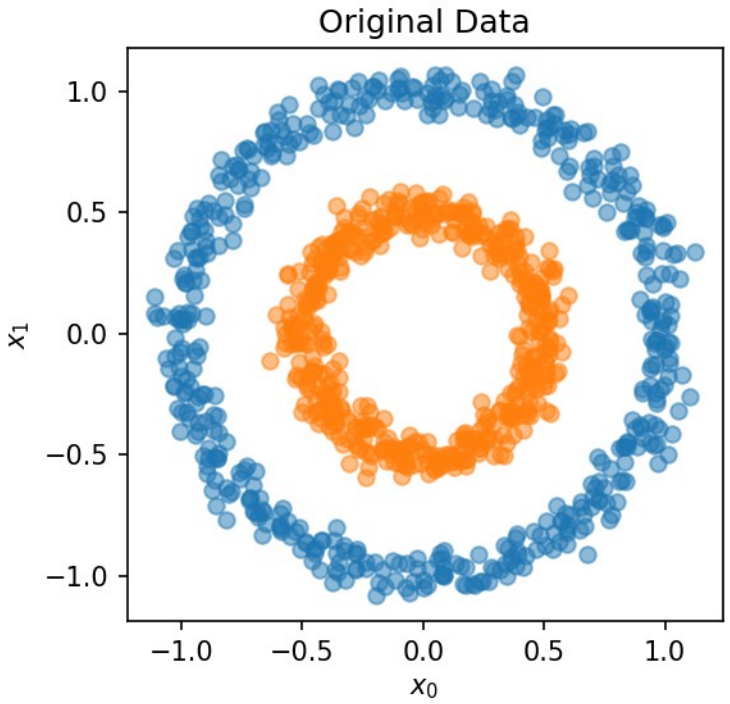
Note: the `plt.subplots()` function will return `fig, ax`, where `ax` is an array of all the subplot axes in the figure. Each individual subplot can be accessed with `ax[i]` which you can then pass to the `plotter()` function's `ax` argument.

```
## YOUR CODE GOES HERE
# 3 subplot using agglomerative clustering for 3 linkage criteria
(single, complete, average)
fig, ax = plt.subplots(1, 3, dpi = 150, figsize = (12,5))
for i, linkage in enumerate(['single', 'complete', 'average']):
    model = AgglomerativeClustering(n_clusters = 3, linkage = linkage)
    model.fit(x)
    plotter(x, model.labels_, ax = ax[i], title = f'Linkage =
{linkage}')
plt.show()
```



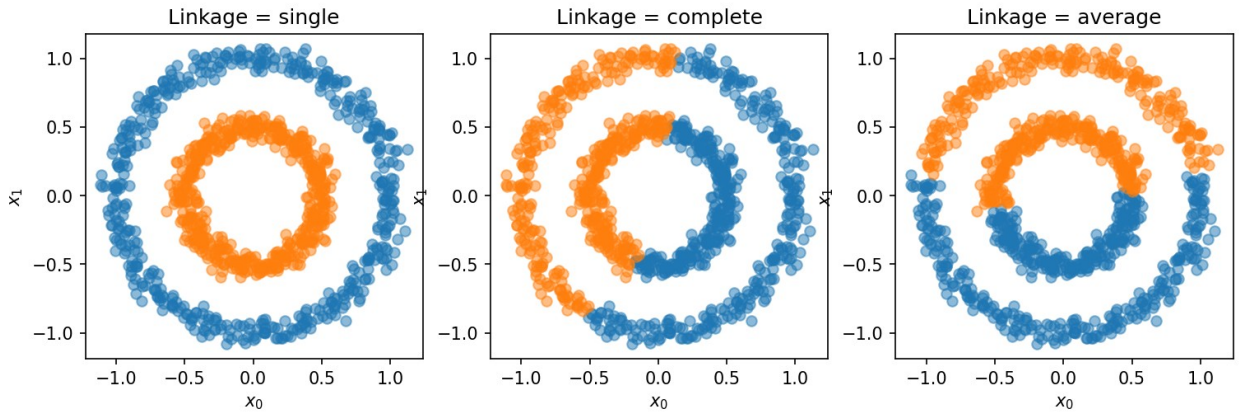
Now we will work on the concentric circle dataset, generated below. Visualize the data using the provided `plotter(x, labels)` function.

```
## DO NOT MODIFY
x, labels = make_circles(1000, factor = 0.5, noise = 0.05,
random_state = 0)

## YOUR CODE GOES HERE
# visualize the data
plotter(x, labels, title = 'Original Data')
```



Again, use the `AgglomerativeClustering()` function to generate 3 side-by-side plots using `plt.subplots()` and the provided `plotter(x, labels, ax, title)` function to visualize the results of the following three linkage criteria `['single', 'complete', 'average']` for the concentric circle dataset.

```
## YOUR CODE GOES HERE
# 3 subplot using agglomerative clustering for 3 linkage criteria
(single, complete, average)
fig, ax = plt.subplots(1, 3, dpi = 150, figsize = (12,5))
for i, linkage in enumerate(['single', 'complete', 'average']):
    model = AgglomerativeClustering(n_clusters = 2, linkage = linkage)
    model.fit(x)
    plotter(x, model.labels_, ax = ax[i], title = f'Linkage =
{linkage}')
plt.show()
```



Discussion

Discuss the performance of the three different linkage criteria on the "blob" dataset, and then on the concentric circle dataset. Why do some linkage criteria perform better on one dataset, but worse on others?

In the blob dataset, the average linkage criteria performed the best in data clustering. The complete linkage criteria did classify some clusters but the outcome is not satisfactory compared to the ground truth dataset. The single linkage criteria performed the worst in data clustering.

In the concentric circle dataset, the single linkage criteria performed the best in data clustering. On the other hand, the complete and average linkage criteria did classify the dataset but the outcome is not correct when compared to the ground truth dataset.

The performance of each linkage criterion will depend on the shape and distribution of the cluster in the given dataset. The single linkage criteria perform better on elongated clusters, while the complete linkage criteria perform better on compact clusters. These unique properties depend on the cluster distance calculations.

M11-L2 Problem 1

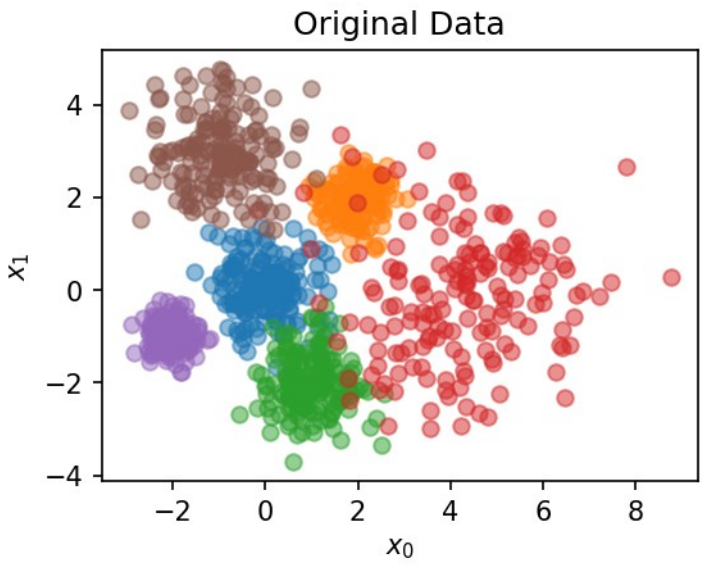
In this problem you will implement the elbow method using three different sklearn clustering algorithms: (`KMeans`, `SpectralClustering`, `GaussianMixture`). You will use the algorithms to find the number of natural clusters for two different datasets, one "blob" shaped dataset, and one concetric circle dataset.

```
import numpy as np
import matplotlib.pyplot as plt
plt.rcParams['figure.dpi'] = 200

from sklearn.datasets import make_blobs, make_circles
from sklearn.cluster import KMeans, SpectralClustering
from sklearn.mixture import GaussianMixture

def plot_loss(loss, ax = None, title = None):
    if ax is None:
        ax = plt.gca()
    ax.plot(np.arange(2, len(loss)+2), loss, 'k-o')
    ax.set_xlabel('Number of Clusters')
    ax.set_ylabel('Loss')
    if title:
        ax.set_title(title)
    return ax

def plot_pred(x, labels, ax = None, title = None):
    if ax is None:
        ax = plt.gca()
    n_clust = len(np.unique(labels))
    for i in range(n_clust):
        ax.scatter(x[labels == i,0], x[labels == i,1], alpha = 0.5)
    ax.set_title(title)
    return ax

def compute_loss(x, labels):
    # Initialize loss
    loss = 0
    # Number of clusters
    n_clust = len(np.unique(labels))
    # Loop through the clusters
    for i in range(n_clust):
        # Compute the center of a given label
        center = np.mean(x[labels == i, :], axis = 0)
        # Compute the sum of squared distances between each point and
its corresponding cluster center
        loss += np.sum(np.linalg.norm(x[labels == i, :] - center, axis
= 1)**2)
    return loss
```

Blob dataset

Visualize the "blob" dataset generated below, using a unique color for each cluster of points, where `y` contains the label of each corresponding point in `x`.

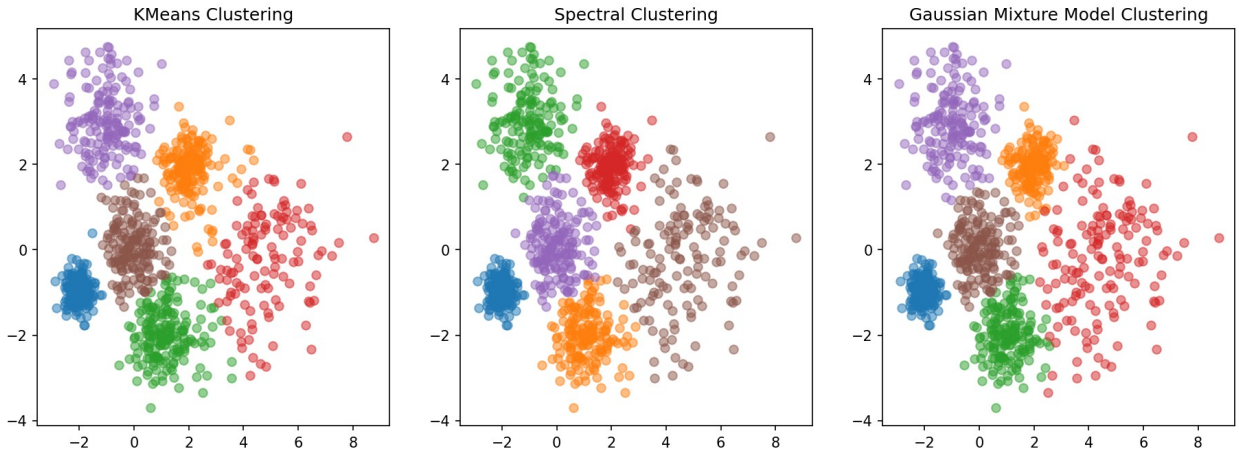
```
## DO NOT MODIFY
x, y = make_blobs(n_samples = 1000, n_features = 2, centers = [[0,0],
[2,2],[1,-2],[4,0],[-2,-1],[-1,3]], cluster_std =
[0.6,0.4,0.6,1.5,0.3,0.8], random_state = 0)

## YOUR CODE GOES HERE
# visualize the data

# plotter fucntion
def plotter(x, labels = None, ax = None, title = None):
    if ax is None:
        _, ax = plt.subplots(dpi = 150, figsize = (4,4))
        flag = True
    else:
        flag = False
    for i in range(len(np.unique(labels))):
        ax.scatter(x[labels == i, 0], x[labels == i, 1], alpha = 0.5)
    ax.set_xlabel('$x_0$')
    ax.set_ylabel('$x_1$')
    ax.set_aspect('equal')
    if title is not None:
        ax.set_title(title)
    if flag:
        plt.show()
    else:
        return ax

plotter(x, y, title = 'Original Data')
```



Use the `sklearn` KMeans, Spectral Clustering, and Gaussian Mixture Model functions to cluster the "blob" data with 6 clusters, and modify the parameters until you get satisfactory results. Plot the results of your three models side-by-side using `plt.subplots` and the provided `plot_pred(x, labels, ax, title)` function.

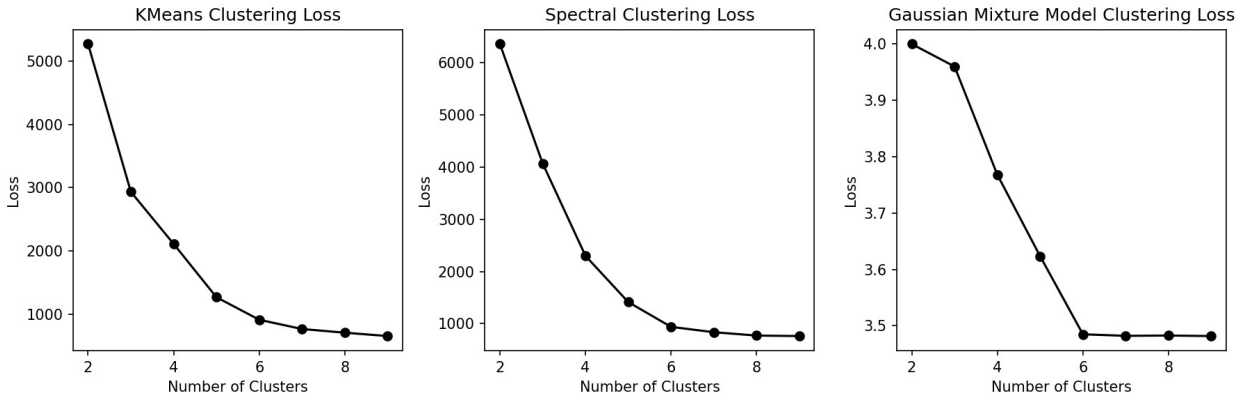
```
## YOUR CODE GOES HERE
# KMeans, Spectral Clustering, Gaussian Mixture Model
n_clust = 6
kmeans = KMeans(n_clusters = n_clust, random_state = 0).fit(x)
spectral = SpectralClustering(n_clusters = n_clust, random_state = 0,
affinity='nearest_neighbors').fit(x)
gmm = GaussianMixture(n_components = n_clust, random_state = 0).fit(x)

 # plot the results
fig, ax = plt.subplots(1, 3, figsize = (15,5), dpi = 150)
plot_pred(x, kmeans.labels_, ax = ax[0], title = 'KMeans Clustering')
plot_pred(x, spectral.labels_, ax = ax[1], title = 'Spectral
Clustering')
plot_pred(x, gmm.predict(x), ax = ax[2], title = 'Gaussian Mixture
Model Clustering')
```

```
<Axes: title={'center': 'Gaussian Mixture Model Clustering'}>
```



Using the parameters you found for the three models above, run each of the clustering algorithms for `n_clust = [2,3,4,5,6,7,8,9]` and compute the sum of squared distances loss for each case using the provided `compute_loss(x, labels)` function, where labels is the cluster assigned to each point by the algorithm. Plot loss versus number of cluster for each your three models in side-by-side subplots using the provided `plot_loss(x, labels, ax, title)` function.

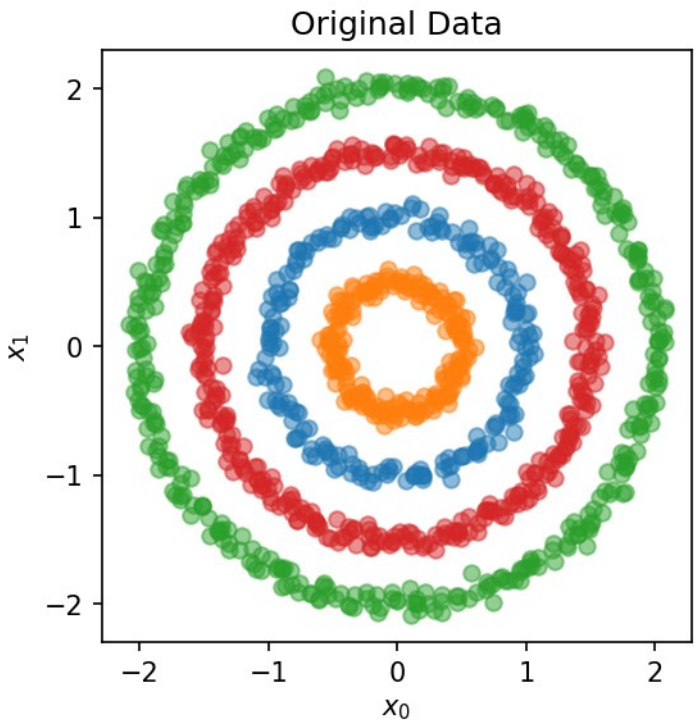
```
## YOUR CODE GOES HERE
# KMeans, Spectral Clustering, and Gaussian Mixture Model
n_clust = [2, 3, 4, 5, 6, 7, 8, 9]
loss_kmeans = []
loss_spectral = []
loss_gmm = []
for i in n_clust:
    kmeans = KMeans(n_clusters = i, random_state = 0).fit(x)
    loss_kmeans.append(compute_loss(x, kmeans.labels_))
    spectral = SpectralClustering(n_clusters = i, random_state = 0,
affinity='nearest_neighbors').fit(x)
    loss_spectral.append(compute_loss(x, spectral.labels_))
    gmm = GaussianMixture(n_components = i, random_state = 0).fit(x)
    loss_gmm.append(-gmm.score(x))

# plot the loss subplot
fig, ax = plt.subplots(1, 3, figsize = (12, 4), dpi = 150)
plot_loss(loss_kmeans, ax = ax[0], title = 'KMeans Clustering Loss')
plot_loss(loss_spectral, ax = ax[1], title = 'Spectral Clustering
Loss')
plot_loss(loss_gmm, ax = ax[2], title = 'Gaussian Mixture Model
Clustering Loss')
plt.tight_layout()
plt.show()
```



Concentric circles dataset

Visualize the "blob" dataset generated below, using a unique color for each cluster of points, where `y` contains the label of each corresponding point in `x`.

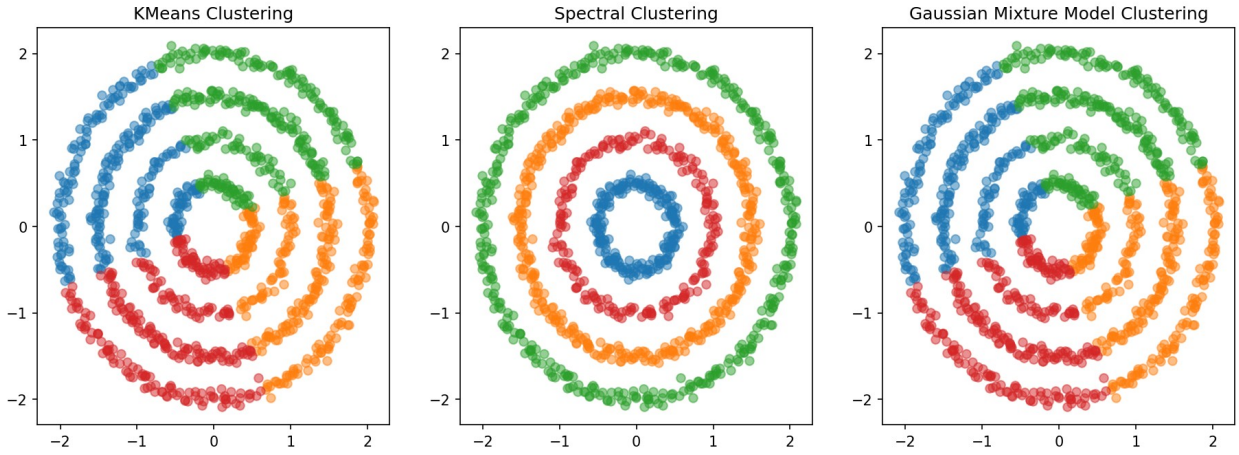
```
## DO NOT MODIFY
x1, y1 = make_circles(n_samples = 400, noise = 0.05, factor = 0.5,
random_state = 0)
x2, y2 = make_circles(n_samples = 800, noise = 0.025, factor = 0.75,
random_state = 1)

x = np.vstack([x1, x2*2])
y = np.hstack([y1, y2+2])

## YOUR CODE GOES HERE
# visualize the data
plotter(x, y, title = 'Original Data')
```



Use the `sklearn` KMeans, Spectral Clustering, and Gaussian Mixture Model functions to cluster the concentric circle data with 4 clusters, and attempt to modify the parameters until you get satisfactory results. Note: you should get good clustering results with Spectral Clustering, but the KMeans and GMM models will struggle to cluster this dataset well. Plot the results of your three models side-by-side using `plt.subplots` and the provided `plot_pred(x, labels, ax, title)` function.

```
## YOUR CODE GOES HERE
# KMeans, Spectral Clustering, Gaussian Mixture Model
n_clust = 4

kmeans = KMeans(n_clusters = n_clust, random_state = 0).fit(x)
spectral = SpectralClustering(n_clusters = n_clust, random_state = 0,
affinity='nearest_neighbors').fit(x)
gmm = GaussianMixture(n_components = n_clust, random_state = 0).fit(x)

 # plot the results
fig, ax = plt.subplots(1, 3, figsize = (15,5), dpi = 150)
plot_pred(x, kmeans.labels_, ax = ax[0], title = 'KMeans Clustering')
plot_pred(x, spectral.labels_, ax = ax[1], title = 'Spectral
Clustering')
plot_pred(x, gmm.predict(x), ax = ax[2], title = 'Gaussian Mixture
Model Clustering')
plt.show()

/Users/ryanwu/Documents/CMU/Spring
24/24-787-AIML/.venv/lib/python3.9/site-packages/sklearn/manifold/
```

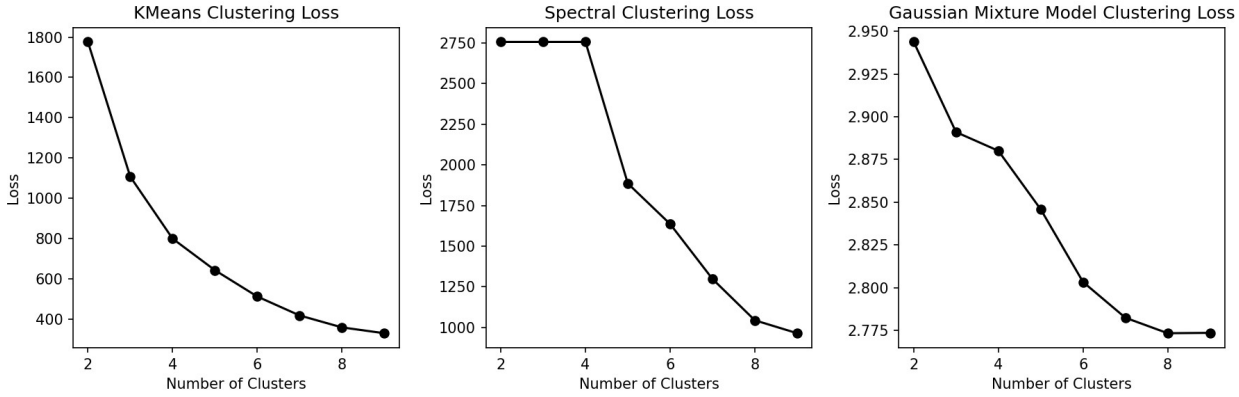
```
_spectral_embedding.py:285: UserWarning: Graph is not fully connected,
spectral embedding may not work as expected.
  warnings.warn(
```



Using the parameters you found for the three models above, run each of the clustering algorithms for `n_clust = [2,3,4,5,6,7,8,9]` and compute the sum of squared distances loss for each case using the provided `compute_loss(x, labels)` function, where labels is the cluster assigned to each point by the algorithm. Plot loss versus number of cluster for each your three models in side-by-side subplots using the provided `plot_loss(x, labels, ax, title)` function.

```
## YOUR CODE GOES HERE
# KMeans, Spectral Clustering, and Gaussian Mixture Model
n_clust = [2, 3, 4, 5, 6, 7, 8, 9]
loss_kmeans = []
loss_spectral = []
loss_gmm = []
for i in n_clust:
    kmeans = KMeans(n_clusters = i, random_state = 0).fit(x)
    loss_kmeans.append(compute_loss(x, kmeans.labels_))
    spectral = SpectralClustering(n_clusters = i, random_state = 0,
affinity='nearest_neighbors').fit(x)
    loss_spectral.append(compute_loss(x, spectral.labels_))
    gmm = GaussianMixture(n_components = i, random_state = 0).fit(x)
    loss_gmm.append(-gmm.score(x))

# plot the loss subplot
fig, ax = plt.subplots(1, 3, figsize = (12, 4), dpi = 150)
plot_loss(loss_kmeans, ax = ax[0], title = 'KMeans Clustering Loss')
plot_loss(loss_spectral, ax = ax[1], title = 'Spectral Clustering
Loss')
plot_loss(loss_gmm, ax = ax[2], title = 'Gaussian Mixture Model
Clustering Loss')
plt.tight_layout()
plt.show()
```

```
/Users/ryanwu/Documents/CMU/Spring
24/24-787-AIML/.venv/lib/python3.9/site-packages/sklearn/manifold/
_spectral_embedding.py:285: UserWarning: Graph is not fully connected,
spectral embedding may not work as expected.
  warnings.warn(
/Users/ryanwu/Documents/CMU/Spring
24/24-787-AIML/.venv/lib/python3.9/site-packages/sklearn/manifold/
_spectral_embedding.py:285: UserWarning: Graph is not fully connected,
spectral embedding may not work as expected.
  warnings.warn(
/Users/ryanwu/Documents/CMU/Spring
24/24-787-AIML/.venv/lib/python3.9/site-packages/sklearn/manifold/
_spectral_embedding.py:285: UserWarning: Graph is not fully connected,
spectral embedding may not work as expected.
  warnings.warn(
/Users/ryanwu/Documents/CMU/Spring
24/24-787-AIML/.venv/lib/python3.9/site-packages/sklearn/manifold/
_spectral_embedding.py:285: UserWarning: Graph is not fully connected,
spectral embedding may not work as expected.
  warnings.warn(
/Users/ryanwu/Documents/CMU/Spring
24/24-787-AIML/.venv/lib/python3.9/site-packages/sklearn/manifold/
_spectral_embedding.py:285: UserWarning: Graph is not fully connected,
spectral embedding may not work as expected.
  warnings.warn(
/Users/ryanwu/Documents/CMU/Spring
24/24-787-AIML/.venv/lib/python3.9/site-packages/sklearn/manifold/
_spectral_embedding.py:285: UserWarning: Graph is not fully connected,
spectral embedding may not work as expected.
  warnings.warn(
/Users/ryanwu/Documents/CMU/Spring
24/24-787-AIML/.venv/lib/python3.9/site-packages/sklearn/manifold/
_spectral_embedding.py:285: UserWarning: Graph is not fully connected,
spectral embedding may not work as expected.
  warnings.warn(
/Users/ryanwu/Documents/CMU/Spring
24/24-787-AIML/.venv/lib/python3.9/site-packages/sklearn/manifold/
_spectral_embedding.py:285: UserWarning: Graph is not fully connected,
spectral embedding may not work as expected.
  warnings.warn(
```



Discussion

1. Discuss the performance of the clustering algorithms on the "blob" dataset. Using the elbow method, were you able to identify the number of natural clusters in the dataset for each of the methods? Does the elbow method work better for some algorithms versus others?

For the blob dataset, the KMeans and the Gaussian Mixture Model clustering methods perform the best in classifying clusters. On the other hand, the Spectral clustering method did not perform well in classifying clusters when compared to the ground truth dataset. From the clustering loss, the number of natural clusters is around 6 to 7 for all clustering methods. The elbow method works well for all the algorithms in this use case.

1. Discuss the performance of the clustering algorithms on the concentric circles dataset. Using the elbow method, were you able to identify the number of natural clusters in the dataset for each of the methods?

For the concentric circles dataset, the spectral clustering method performed the best in classifying clusters. On the other hand, the KMeans and the Gaussian Mixture Model clustering methods were unable to correctly classify clusters. From the clustering loss, the number of natural clusters is hard to tell for KMeans and Spectral clustering methods. For the Gaussian Mixture Model, the number of natural clusters is around 8.

1. Does the sum of squared distances work well as a loss function for each of the three clustering algorithms we implemented? Does the sum of squared distance fail on certain types of clusters?

The sum of squared distances works well as a loss function for the KMeans clustering method. However, the sum of squared distances fails for Spectral and Gaussian Mixture clustering methods.

Problem 1

Problem Description

In this problem you will use DBSCAN to cluster two melt pool images from a powder bed fusion metal 3D printer. Often times during printing there can be spatter around the main melt pool, which is undesirable. If we can successfully train a model to identify images with large amounts of spatter, we can automatically monitor the printing process.

Fill out the notebook as instructed, making the requested plots and printing necessary values.

You are welcome to use any of the code provided in the lecture activities.

Summary of deliverables:

- Bitmap visualization of melt pool images 1 and 2
- Visualization of final DBSCAN clustering result for both melt pool images
- Discussion of tuning, final number of clusters, and the sensitivity of the model parameters for the two images.

Imports and Utility Functions:

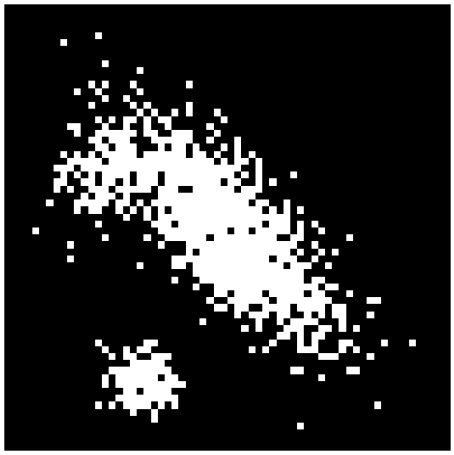
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn.cluster import DBSCAN

def points_to_bitmap(x):
    bitmap = np.zeros((64, 64), dtype=int)
    cols, rows = x[:, 0], x[:, 1]
    bitmap[rows, cols] = 1
    return bitmap

def plot_bitmap(bitmap):
    _, ax = plt.subplots(figsize=(3,3), dpi = 200)
    colors = ListedColormap(['black', 'white'])
    ax.imshow(bitmap, cmap = colors, origin = 'lower')
    ax.axis('off')

def plot_points(x, labels):
    fig = plt.figure(figsize = (5,4), dpi = 150)
    for i in range(min(labels),max(labels)+1):
        plt.scatter(x[labels == i, 0], x[labels == i, 1], alpha = 0.5,
marker = 's')
    plt.gca().set_aspect('equal')
    plt.tight_layout()
    plt.show()
```

Melt Pool Image #1

Load the first meltpool scan from the `m11-hw1-data1.txt` file using `np.loadtxt()`, and pass the `dtype = int` argument to ensure all values are loaded with their integer coordinates. You can convert these points to a binary bitmap using the provided `points_to_bitmap()` function, and then visualize the image using the provided `plot_bitmap()` function.

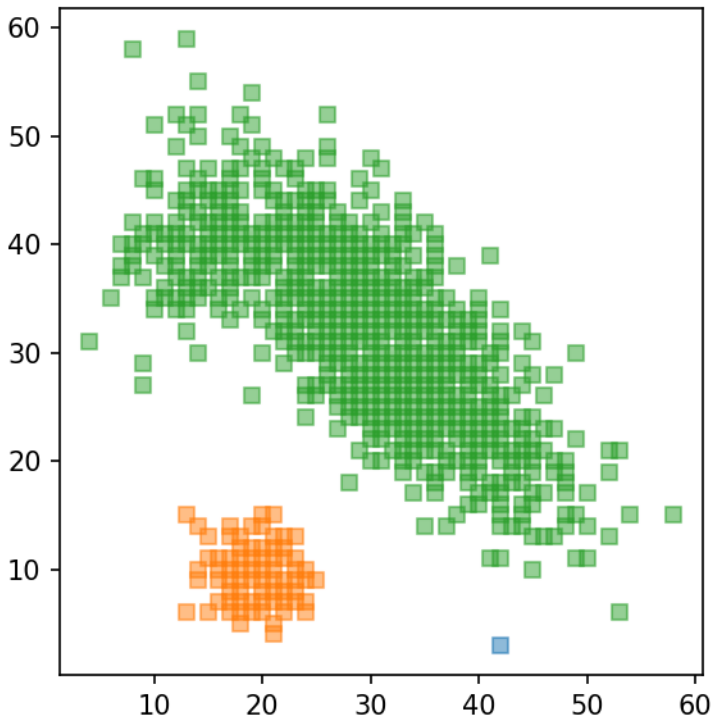
Note: you will use the integer coordinates for clustering, the bitmap is just for visualizing the data.

```
## YOUR CODE GOES HERE
# load the given data
data = np.loadtxt("data/m11-hw1-data1.txt", dtype=int)
bitmap = points_to_bitmap(data)
plot_bitmap(bitmap)
```



Using the `sklearn.cluster.DBSCAN()` function, cluster the melt pool until you get well defined clusters. You will have to modify the `eps` and `min_samples` parameters to get satisfactory results. You can visualize the clustering with the provided `plot_points(x, labels)` function, where `x` is the integer coordinates of all the points, and labels are the labels assigned by `DBSCAN`. Plot the results of your final clustering using the `plot_points()` function

```
## YOUR CODE GOES HERE
# cluster the data using DBSCAN
dbscan = DBSCAN(eps=7, min_samples=3)
labels = dbscan.fit_predict(data)
plot_points(data, labels)
```



Melt Pool Image #2

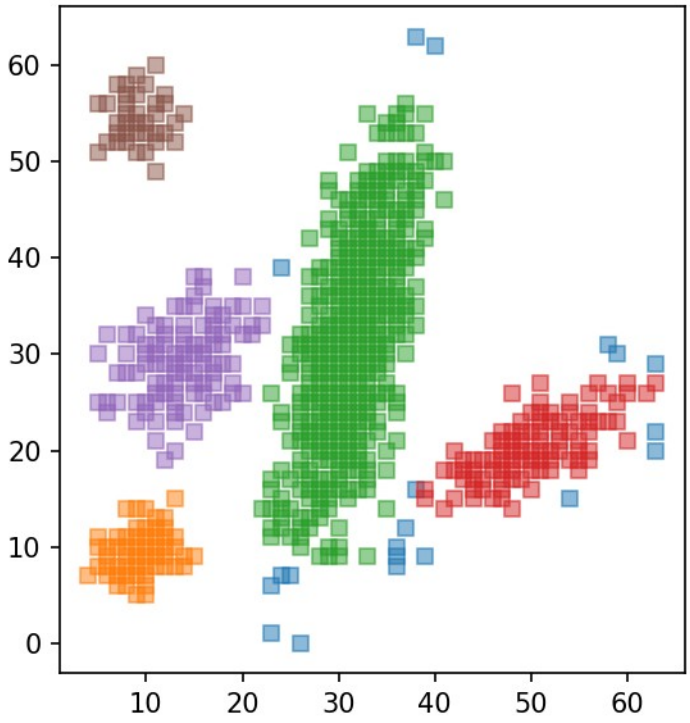
Now load the second melt pool scan from the `m11-hw1-data2.txt` file using `np.loadtxt()`, and the `dtype = int` argument to ensure all values are loaded with their integer coordinates. Again, convert the points to a binary bitmap, and visualize the bitmap using the provided functions.

```
## YOUR CODE GOES HERE
# load the given data
data = np.loadtxt("data/m11-hw1-data2.txt", dtype=int)
bitmap = points_to_bitmap(data)
plot_bitmap(bitmap)
```

Using the `sklearn.cluster.DBSCAN()` function, cluster the meltpool until you get well defined clusters. You will have to modify the `eps` and `min_samples` parameters to get satisfactory results. You can visualize the clustering with the provided `plot_points(x, labels)` function, where `x` is the integer coordinates of all the points, and labels are the labels assigned by `DBSCAN`. Plot the results of your final clustering using the `plot_points()` function.

Note: this melt pool is significantly noisier than the last and therefore requires more sensitive tuning of the two DBSCAN parameters.

```
## YOUR CODE GOES HERE
# cluster the data using DBSCAN
dbscan = DBSCAN(eps=3, min_samples=6)
labels = dbscan.fit_predict(data)
plot_points(data, labels)
```



Discussion

Discuss how you tuned the `eps` and `min_samples` parameters for the two models. How many clusters did you end up finding in each image? Why does a wider range of `eps` and `min_samples` values successfully cluster melt pool image #1 compared to melt pool image #2?

I used the trial-and-error method when tuning the eps and min_samples parameters.

For image number 1, the eps is set to 7 and the min_sample is set to 3. The model identified 3 clusters but I believe the ground truth should be only 2 clusters with blue points being false classification.

For image number 2, the eps is set to 3 and the min_sample is set to 6. The model identified 6 clusters but I believe the ground truth should be only 5 with blue points being false classification.

A wider range of EPS and min_samples values can successfully classify clusters for image number 1 compared to image number 2. This is because the given ground truth dataset for image 1 has more distance cluster characteristics with less overlapping, which gives more flexibility in parameter tuning. On the other hand, the ground truth dataset for image 2 is more complete with more closely distributed clusters, which requires a more definite parameter declaration.