SRECHARAN SELVAM

Homework 11

Instructions

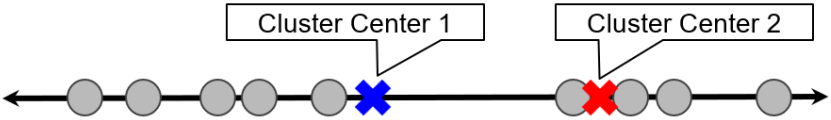
This homework contains **2** concepts and **5** programming questions. In MS word or a similar text editor, write down the problem number and your answer for each problem. Combine all answers for concept questions in a single PDF file. Export/print the Jupyter notebook as a PDF file including the code you implemented and the outputs of the program. Make sure all plots and outputs are visible in the PDF.

Combine all answers into a single PDF named andrewID_hw11.pdf and submit it to Gradescope before the due date. Refer to the syllabus for late homework policy. Please assign each question a page by using the "Assign Questions and Pages" feature in Gradescope.

Here is a breakdown of the points for programming questions:

Name	Points
M11-L1-P1	15
M11-L1-P2	15
M11-L1-P3	15
M11-L2-P1	15
M11-HW1	30

Problem 1 (6 points)



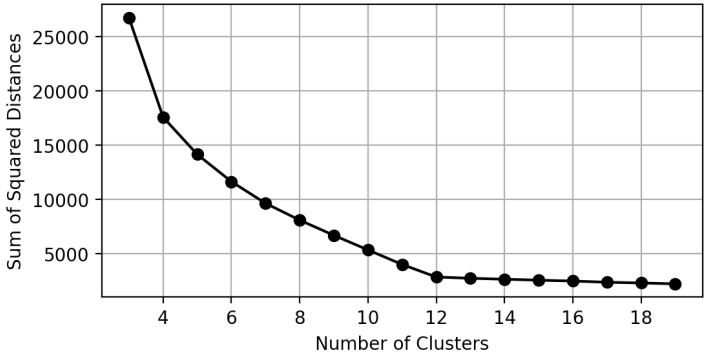
We have randomly placed two cluster centers amongst the data above to initialize the K-Means algorithm.
Q1: Now we will assign a cluster to each of the data points, based which centroid they are closest to. How many points are assigned to each cluster center. (Text Entry)

Q2: Now that we have assigned clusters to each of the data points, we will proceed to the next iteration of K-Means and move the cluster centers to the centroid of their data points. Which direction will each of the cluster centers 1 and 2 move, respectively? (Multiple Choice)

1. Left, Left
2. Left, Right
3. Right, Left
4. Right, Right

Problem 2 (4 Points)

Consider the following elbow method plot generated using the K-Means algorithm. What is the natural number of clusters for this dataset?
(Text Entry)



ANSWERS:-

PROBLEM 1:-

Q1: cluster center 1 ⇒ 5

cluster center 2 ⇒ 4

Q2:

OPTION 2: Left, Right

PROBLEM 2:-

The natural number of clusters for this dataset is 12

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.

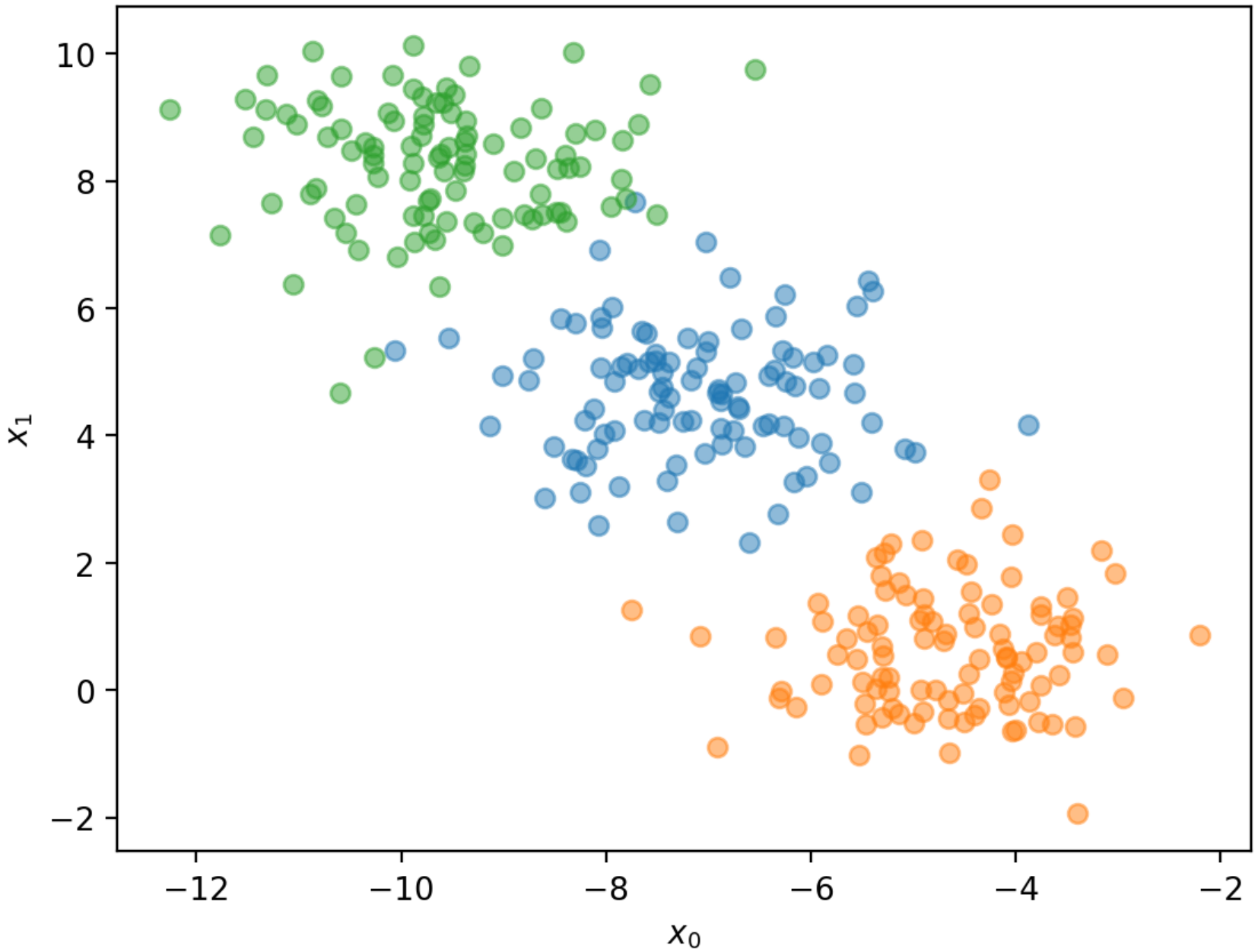
In [31]:
```
import numpy as np
import matplotlib.pyplot as plt

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 = 'Mis
    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.

In [32]:
```
## 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

In [33]:
```
## YOUR CODE GOES HERE
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

In [34]:
```
## FILL IN THE FOLLOWING FUNCTION
def find_cluster(point, centers):
    distances = np.linalg.norm(point - centers, axis=1)
    return np.argmin(distances)
```

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`

In [35]:
```
## FILL IN THE FOLLOWING FUNCTION
def assign_labels(x, centers):
    labels = np.array([find_cluster(point, centers) for point in x])
    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

In [37]:
```
## FILL IN THE FOLLOWING FUNCTION
def update_centers(x, labels):
    num_clusters = len(np.unique(labels))
```

```
    new_centers = np.array([x[labels == i].mean(axis=0) for i in range(num_clusters)])
    return new_centers
```

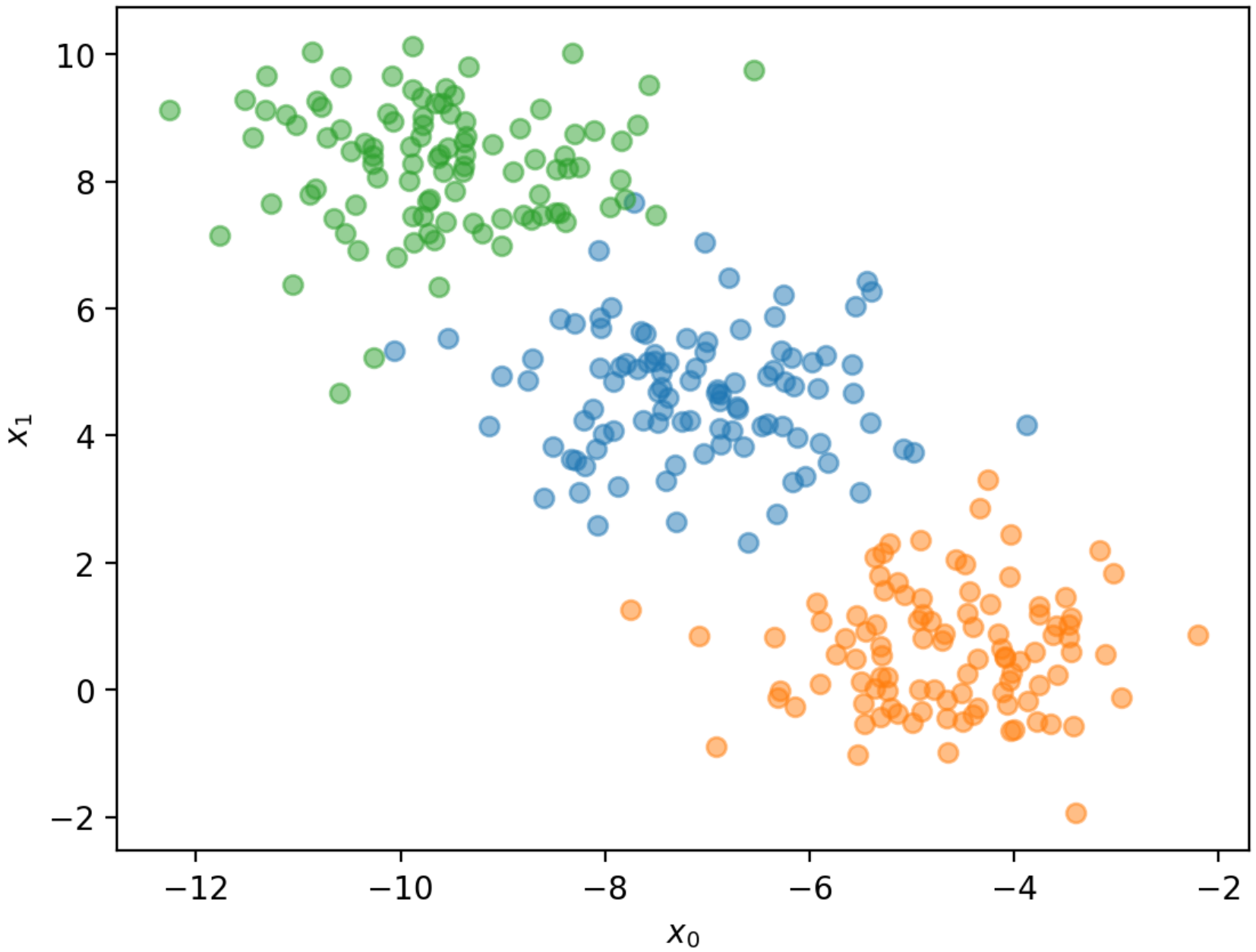
Finally write a function `myKMeans(x, init_centers)` which will run the KMeans algorithm, provided all the points in `x` and the coordinates of the initial cluster centers in `init_centers`. Run the algorithm until there is no change in cluster membership in subsequent iterations. Your function should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

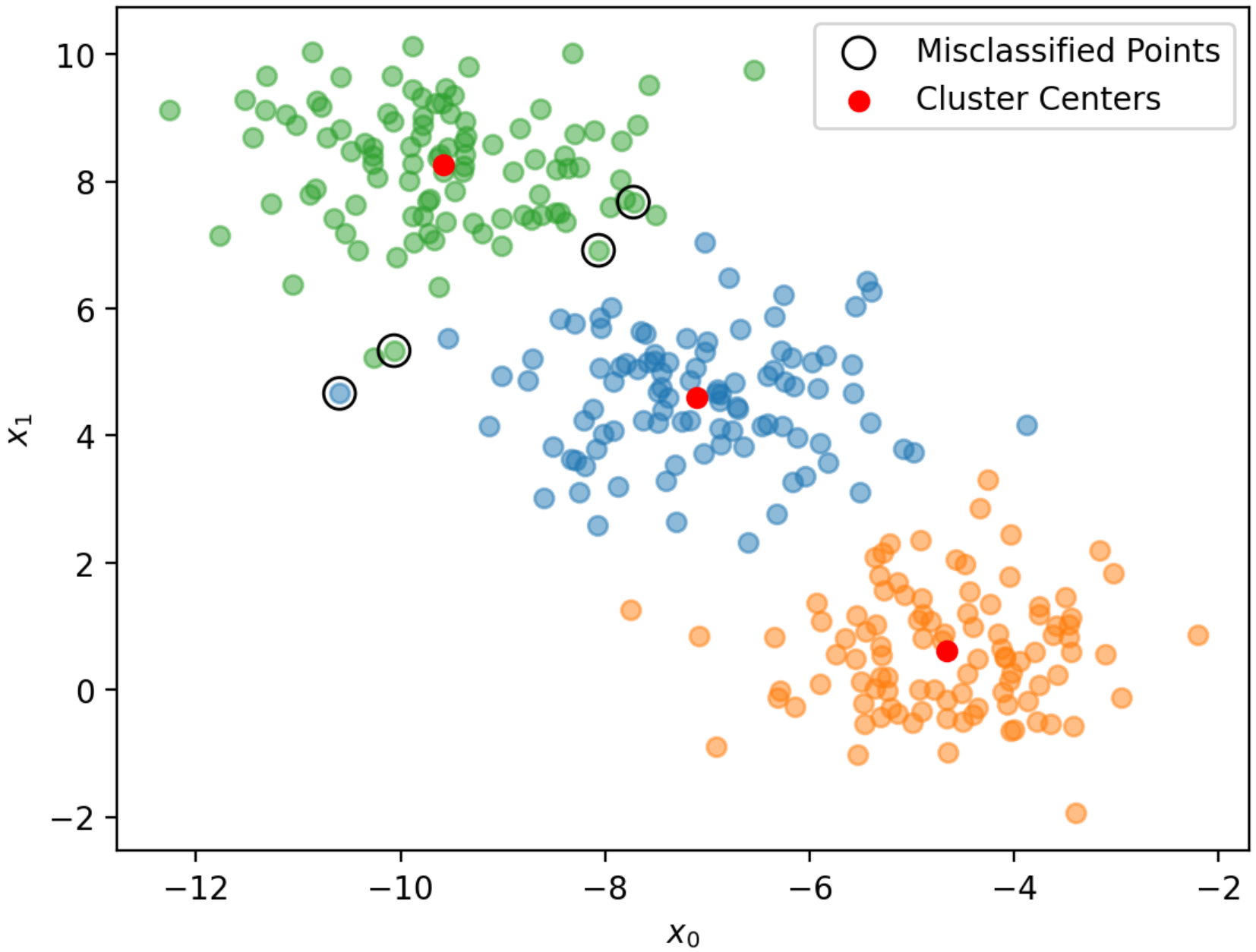
- `x` is a 300×2 numpy array containing the coordinates of all the points in the dataset
- `init_centers` is a 3×2 numpy array containing the coordinates of the three cluster centers provided to you
- **return** `labels` and `centers` as defined above

In [38]:
```
## FILL IN THE FOLLOWING FUNCTION
def myKMeans(x, init_centers):
    centers = init_centers
    while True:
        labels = assign_labels(x, centers)
        new_centers = update_centers(x, labels)
        if np.all(centers == new_centers):
            break
        centers = new_centers
    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.

In [39]:
```
## YOUR CODE GOES HERE
x_blob, y_blob = make_blobs(n_samples=300, n_features=2, random_state=12)
plotter(x_blob, y_blob)

init_centers_blob = np.array([[-5, 5], [0, 0], [-10, 10]])
labels_blob, centers_blob = myKMeans(x_blob, init_centers_blob)
plotter(x_blob, y_blob, labels_blob, centers_blob)
```


10
8
6
4
2
0
−2
x_1
−12
−10
−8
−6
−4
−2
x_0



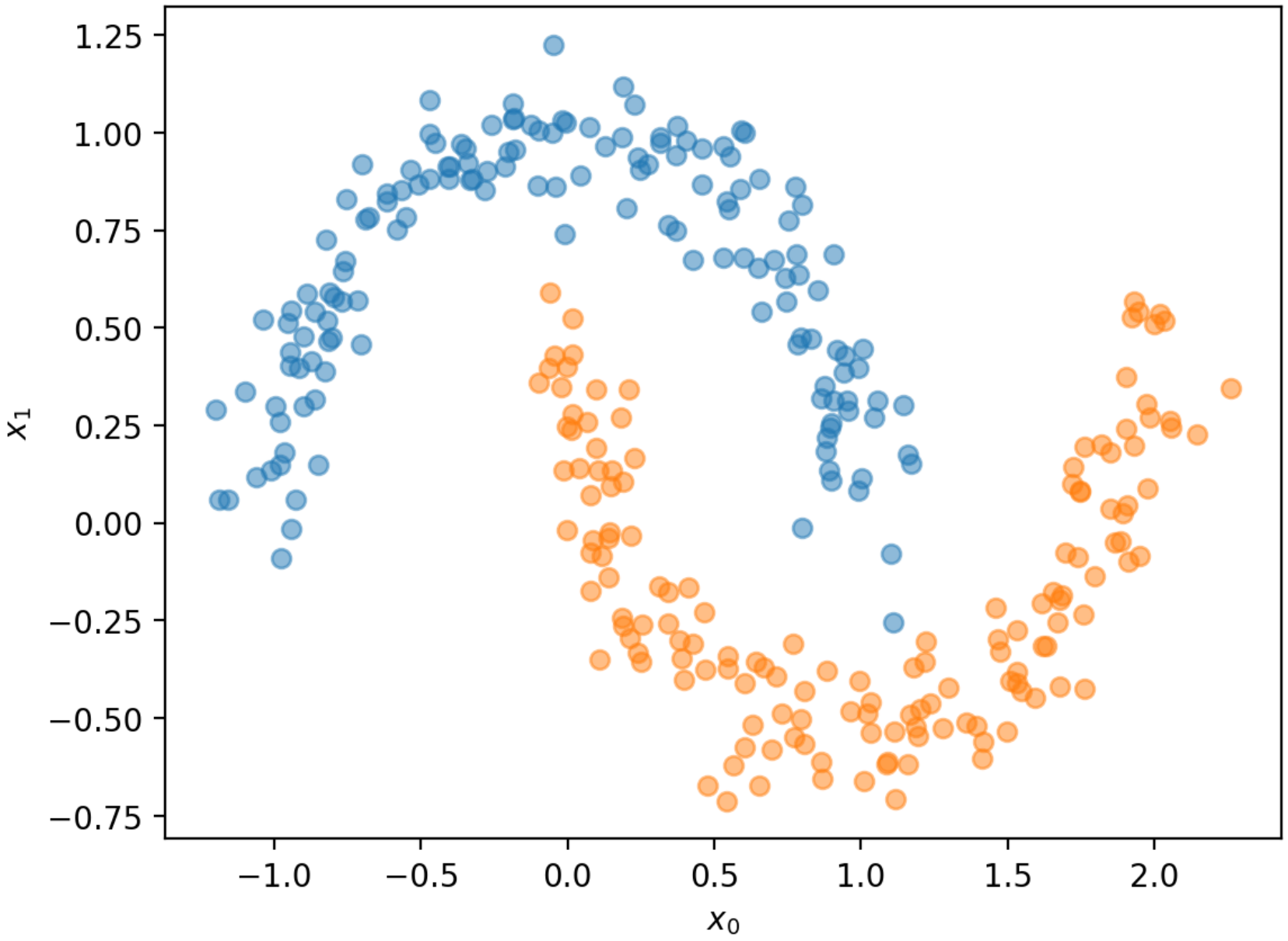
Moon Dataset

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

In [42]:
```
## DO NOT MODIFY
x,y = make_moons(n_samples = 300, noise = 0.1, random_state = 0)
```

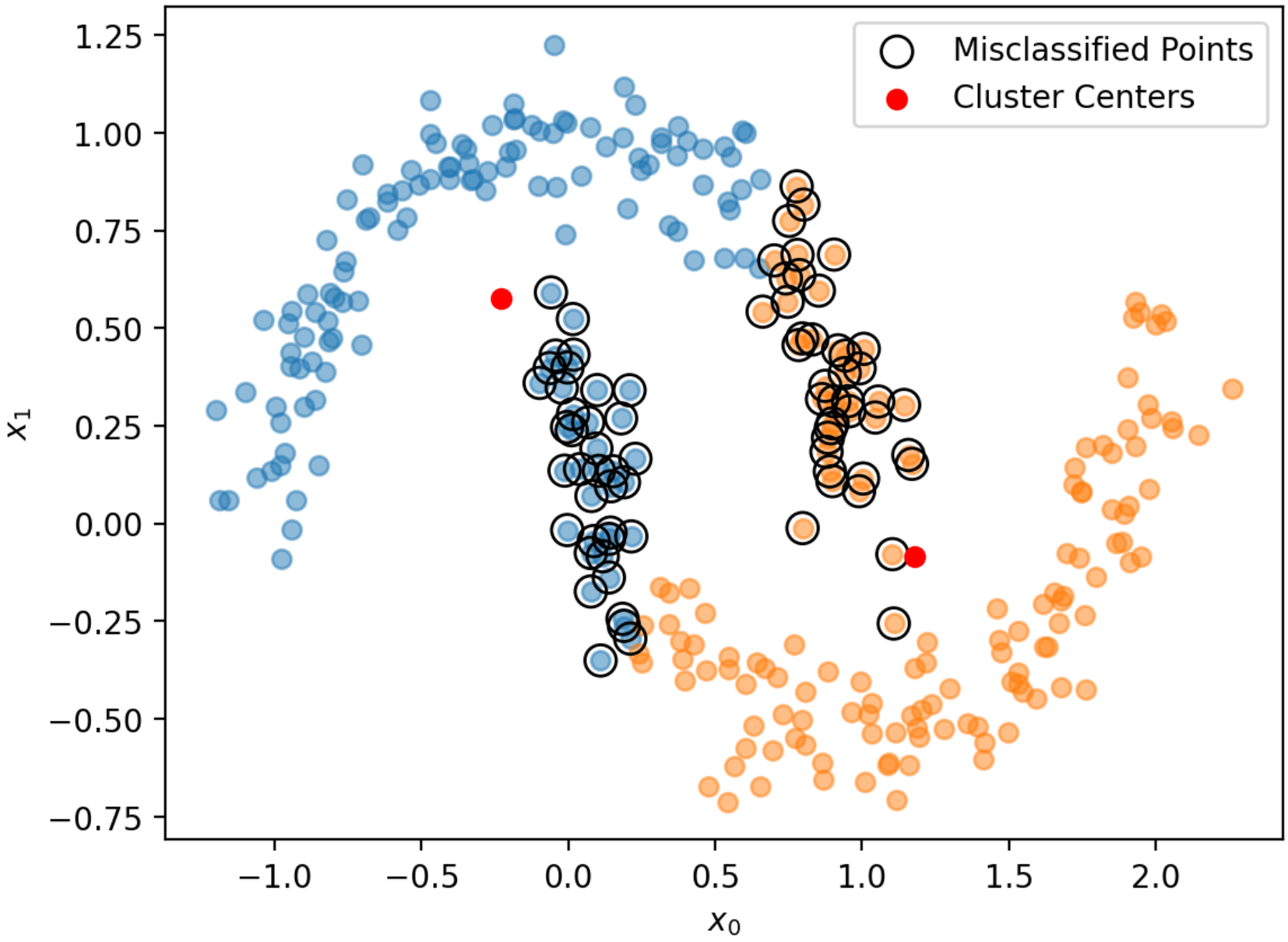
Visualize the data using the `plotter(x,y)` function.

In [43]:
```
## YOUR CODE GOES HERE
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)`.

In [44]:
```
## YOUR CODE GOES HERE
init_centers_moon = np.array([[0, 1], [1, -0.5]])
labels_moon, centers_moon = myKMeans(x_moon, init_centers_moon)
plotter(x_moon, y_moon, labels_moon, centers_moon)
```


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.

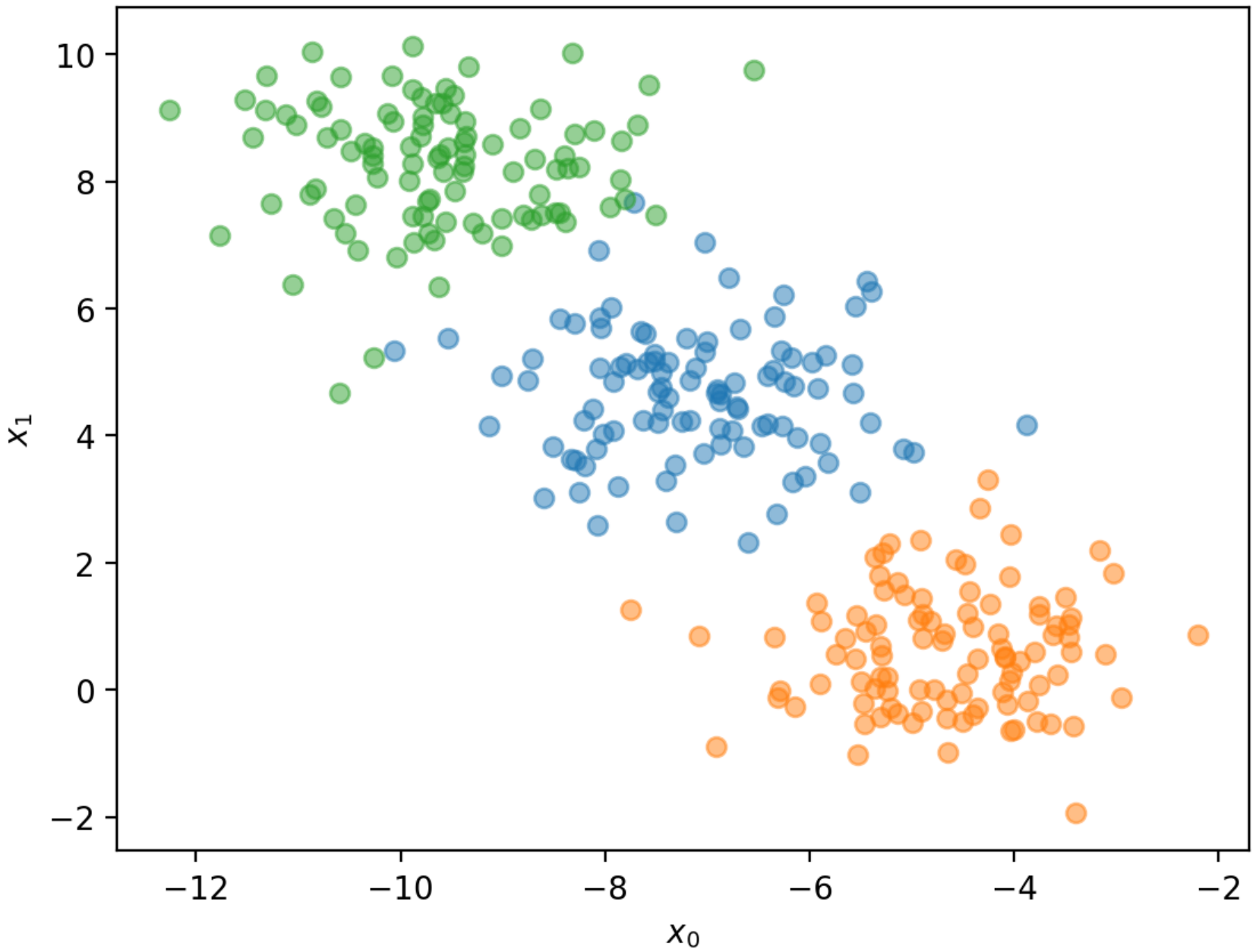
In [17]:
```
import numpy as np
import matplotlib.pyplot as plt

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 = 'Mis
    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.

In [18]:
```
## 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

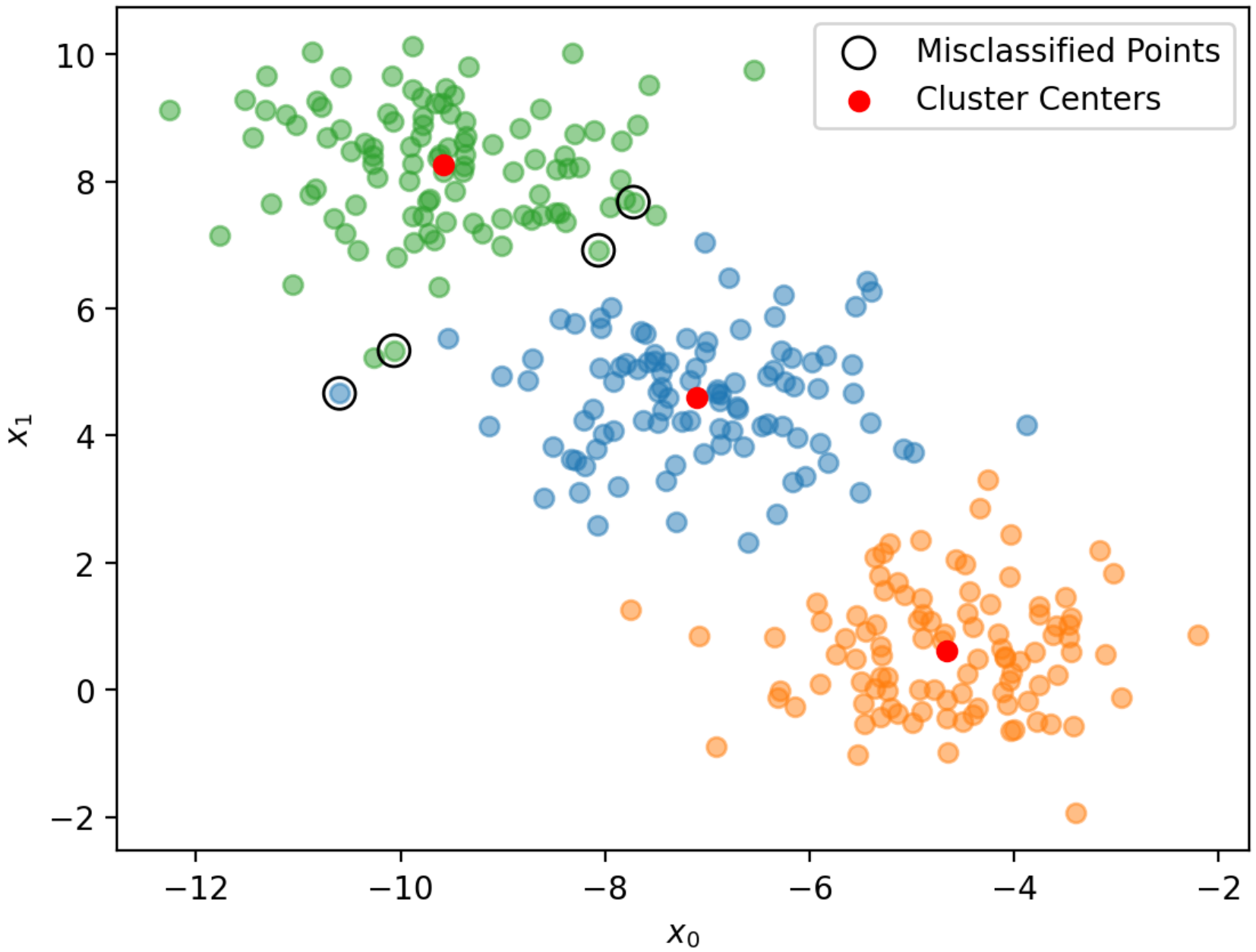
In [19]:
```
## YOUR CODE GOES HERE
plotter(x, y)
```



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.

In [20]:
```
## YOUR CODE GOES HERE
init_centers_blob = np.array([[-5, 5], [0, 0], [-10, 10]])
kmeans_blob = KMeans(n_clusters=3, init=init_centers_blob, n_init=1)
kmeans_blob.fit(x_blob)
labels_blob = kmeans_blob.labels_
centers_blob = kmeans_blob.cluster_centers_

plotter(x_blob, y_blob, labels_blob, centers_blob)
```



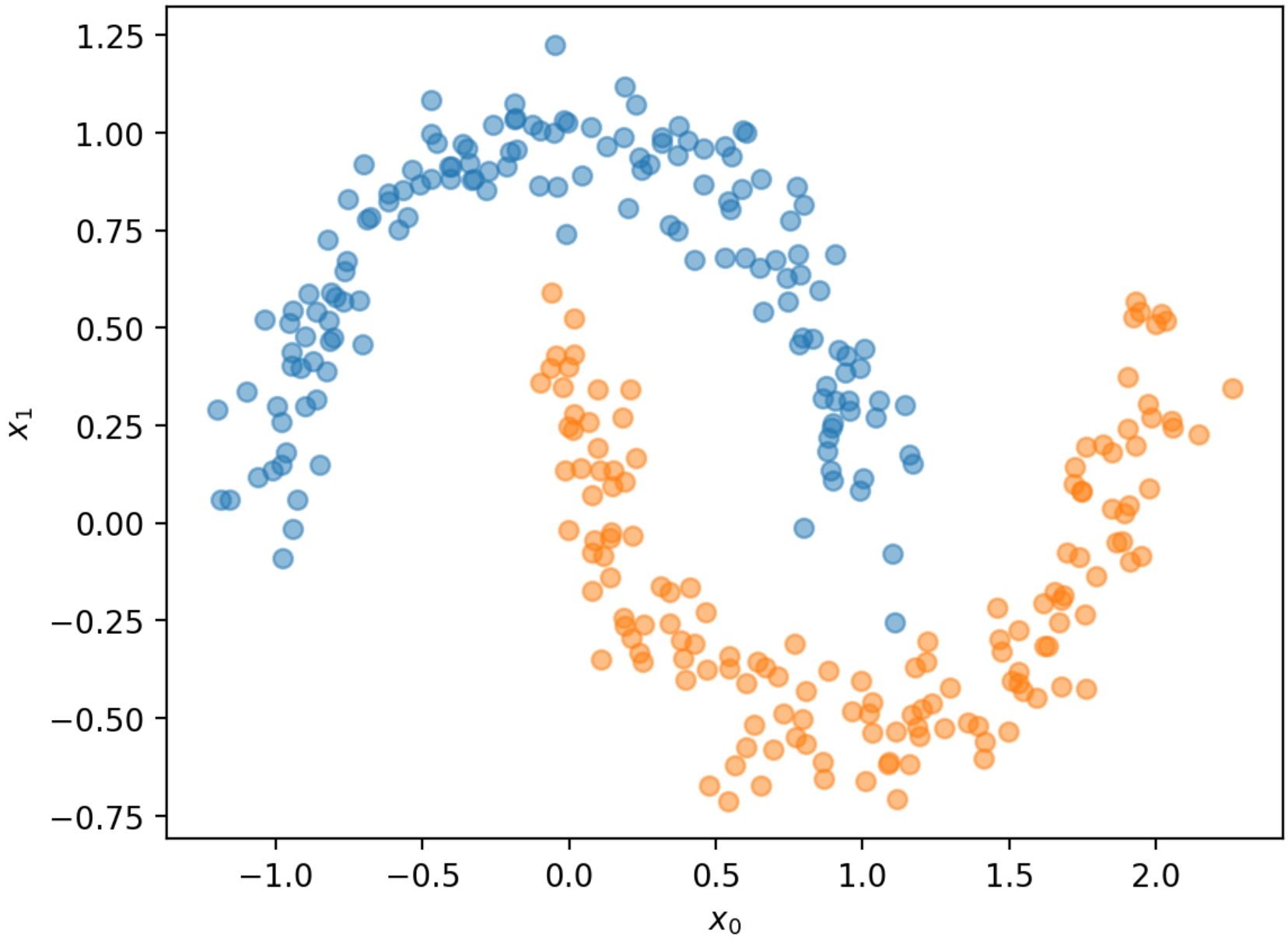
Moon Dataset

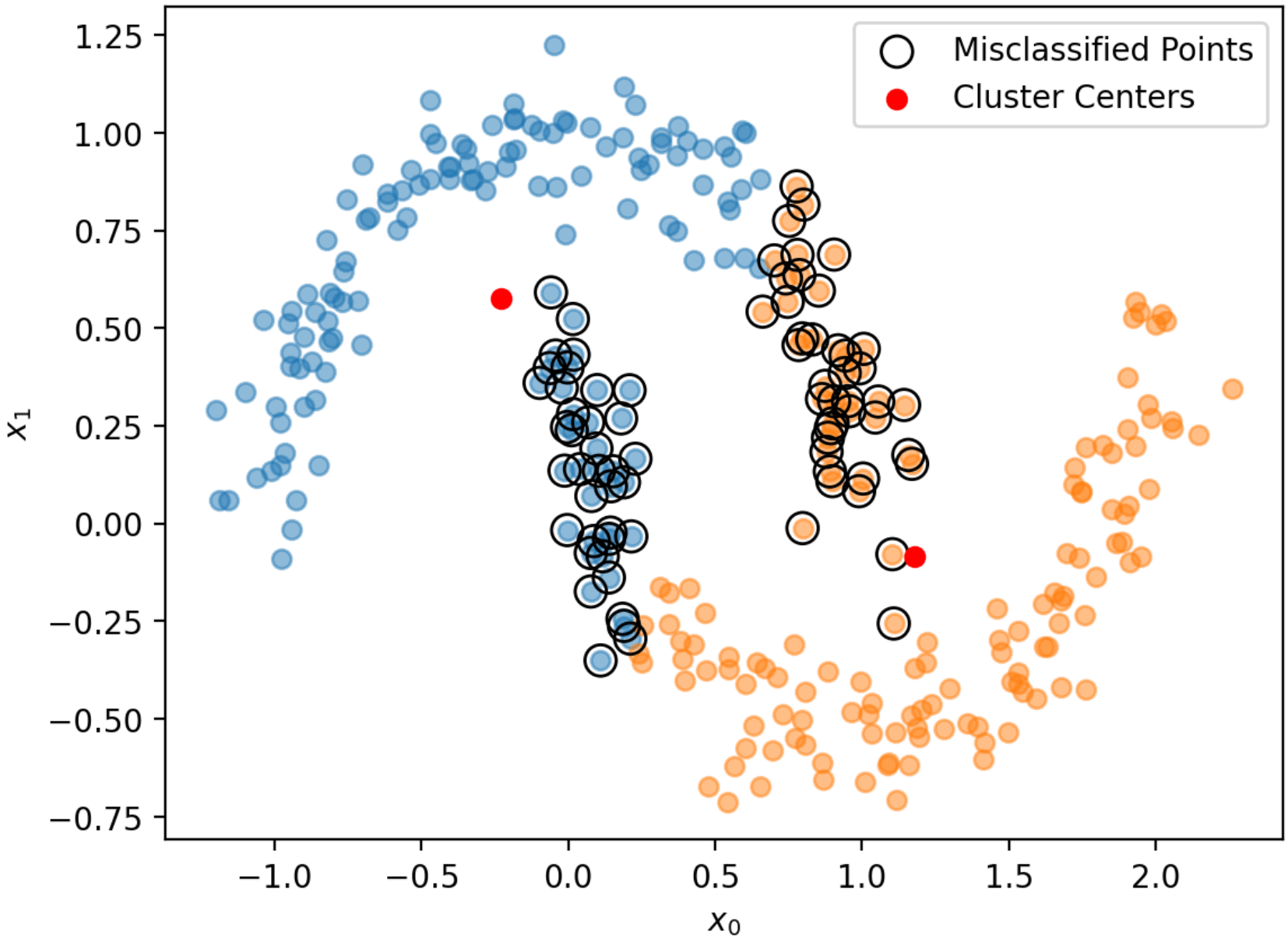
Now we will try using the `sklearn.cluster.KMeans()` function on the moons dataset from problem 1.

In [21]:
```
## 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.

In [15]:
```
## YOUR CODE GOES HERE
plotter(x, y)

init_centers_moon = np.array([[0, 1], [1, -0.5]])
kmeans_moon = KMeans(n_clusters=2, init=init_centers_moon, n_init=1)
kmeans_moon.fit(x_moon)
labels_moon = kmeans_moon.labels_
centers_moon = kmeans_moon.cluster_centers_

plotter(x_moon, y_moon, labels_moon, centers_moon)
```


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.

Your response goes here

The outcomes of the hand-coded K-Means algorithm and its sklearn counterpart are consistent with each other. While sklearn offers various options for initializing clusters, the fundamental process of alternating between updating cluster assignments and centroids until convergence is common to both implementations. Convergence is typically achieved when there is no significant change in centroid positions. The sklearn implementation uses Euclidean distance to measure the proximity between data points and centroids, a method which should also be employed in a hand-coded version to ensure comparable results. In summary, the identical results obtained from both the hand-coded and sklearn implementations of the K-Means algorithm are due to the similarity in their input parameters, algorithmic structure, and initial conditions, leading to the same final outcome.

M11-L1 Problem 3

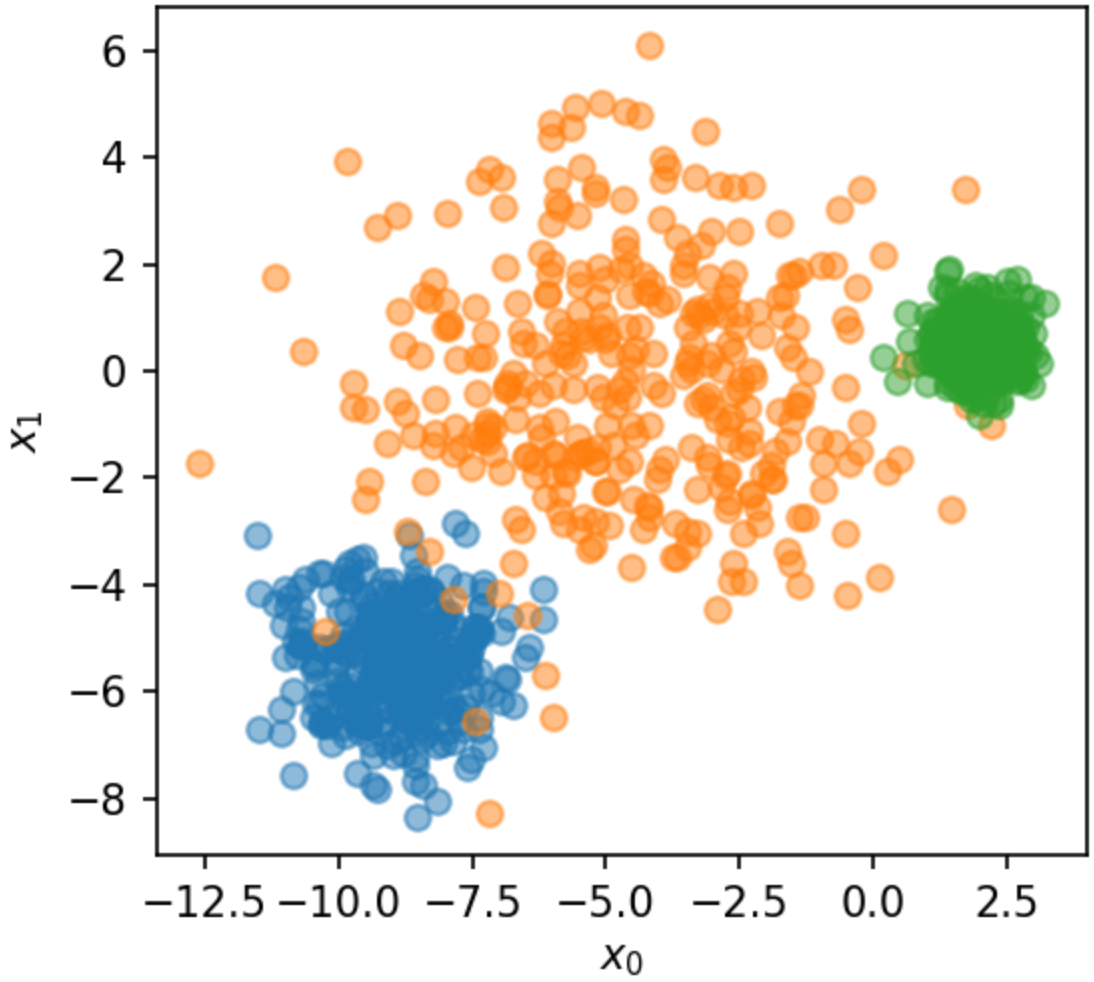
In this problem you will use the `sklearn` implementation of hierarchical clustering with three different linkage criteria `('single', 'complete', 'average')` to clusters two datasets: a "blob" shaped dataset with three classes, and a concentric circle dataset with two classes.

In [17]:
```
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.

In [18]:
```
## DO NOT MODIFY
x, labels = make_blobs(n_samples = 1000, cluster_std=[1.0, 2.5, 0.5], random_state = 170)
```

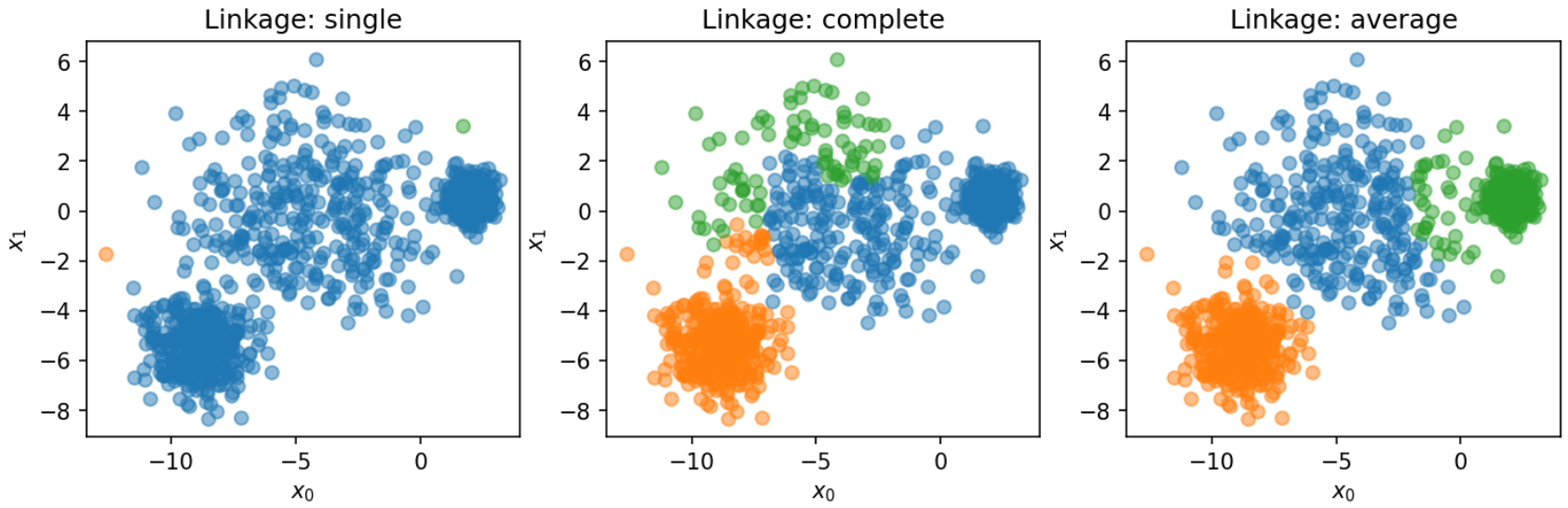
In [19]:
```
## YOUR CODE GOES HERE
plotter(x, labels)
```



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.

In [12]:
```
## YOUR CODE GOES HERE
fig, ax = plt.subplots(1, 3, figsize=(12, 4), dpi=150)
linkage_criteria = ['single', 'complete', 'average']

for i, linkage in enumerate(linkage_criteria):
    clustering = AgglomerativeClustering(n_clusters=3, linkage=linkage)
    labels = clustering.fit_predict(x_blob)
    plotter(x_blob, labels, ax=ax[i], title=f'Linkage: {linkage}')
plt.show()
```

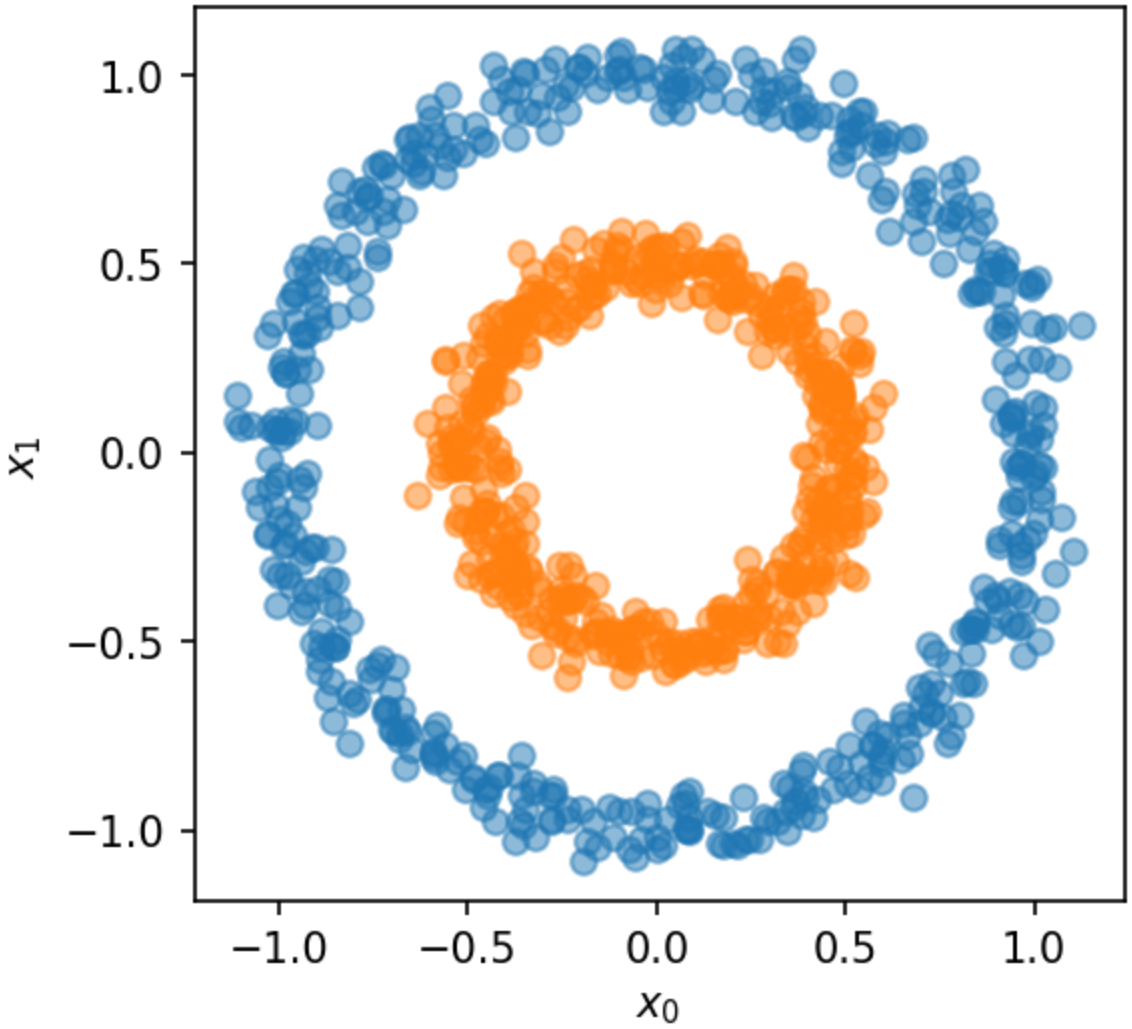


Now we will work on the concentric circle dataset, generated below. Visualize the data using the provided `plotter(x, labels)` function.

In [20]:
```
## DO NOT MODIFY
```

```
x, labels = make_circles(1000, factor = 0.5, noise = 0.05, random_state = 0)
```

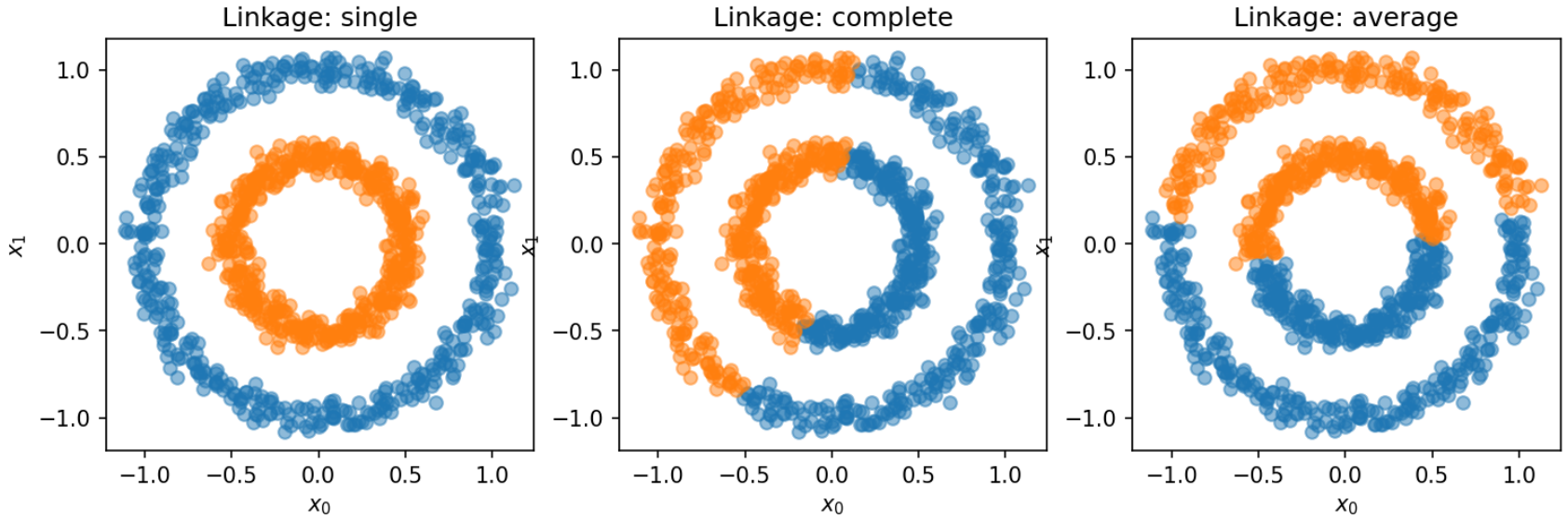
In [21]:
```
## YOUR CODE GOES HERE
x_circles, labels_circles = make_circles(1000, factor=0.5, noise=0.05, random_state=0)
plotter(x_circles, labels_circles)
```



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.

In [22]:
```
## YOUR CODE GOES HERE
fig, ax = plt.subplots(1, 3, figsize=(12, 4), dpi=150)
linkage_criteria = ['single', 'complete', 'average']

for i, linkage in enumerate(linkage_criteria):
    clustering = AgglomerativeClustering(n_clusters=2, linkage=linkage)
    labels = clustering.fit_predict(x_circles)
    plotter(x_circles, 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?

Your response goes here

The 'average' linkage criterion demonstrates effective performance on the 'blob' dataset, while the 'single' and 'complete' linkage criteria exhibit less favorable results.

In contrast, the 'single' linkage criterion excels on the 'concentric circle' dataset, with the 'complete' and 'average' criteria falling behind in performance . This variation in performance is rooted in the inherent assumptions of each linkage criterion. The 'single' linkage assumes clusters are elongated and chain-like, making it suitable for circular structures but less effective for blob-like formations. On the other hand, both 'complete' and 'average' linkage criteria are based on the assumption of compact and uniformly dense clusters, leading to better results with blob shapes but poorer outcomes with ring-like structures.ios.

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.

In [21]:
```
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))
```

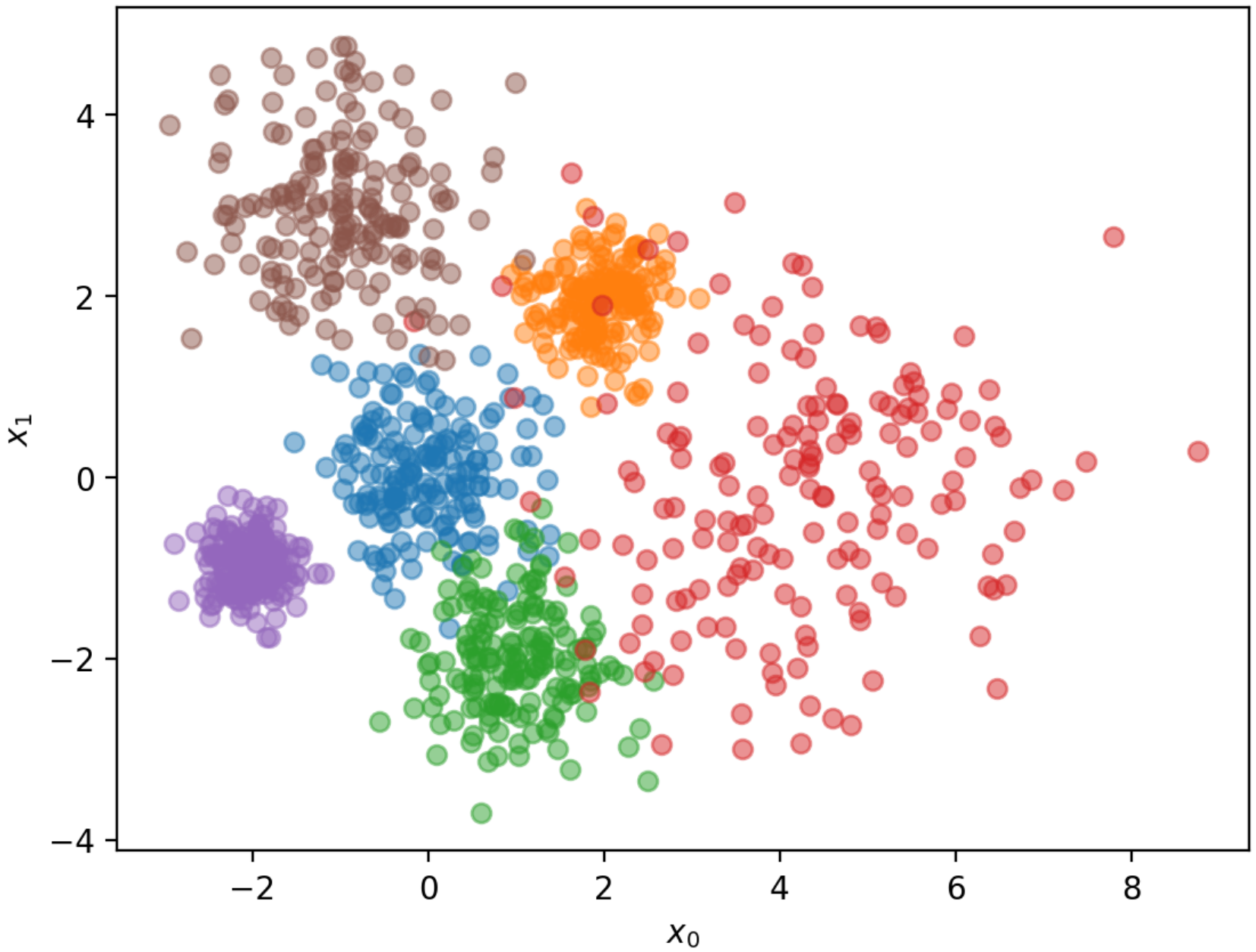
```
    # Loop through the clusters
    for i in range(n_clust):
        # Compute the center of a given label
        center = np.mean(x[labels == i, :], axis = 0)
        # Compute the sum of squared distances between each point and its corresponding cluster center
        loss += np.sum(np.linalg.norm(x[labels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

In [33]:
```
## 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
```

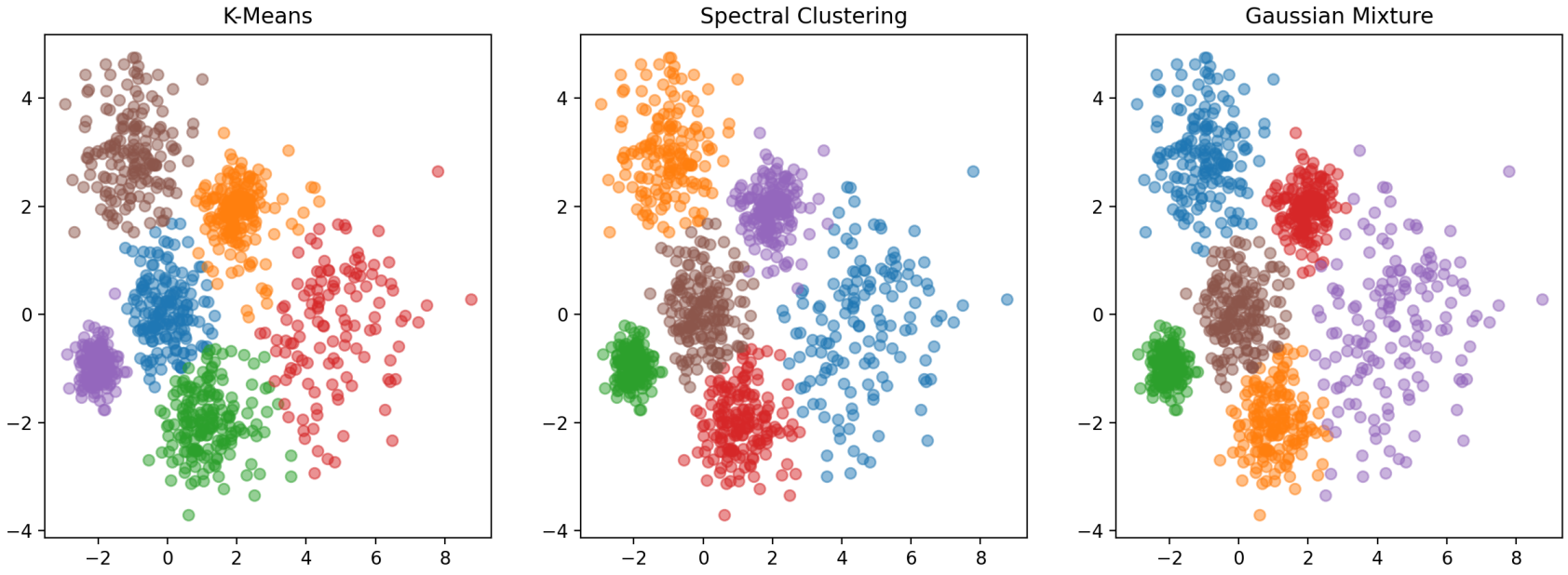
In [34]:
```
## YOUR CODE GOES HERE
def plotter(x, y, labels=None, centers=None):
    fig = plt.figure(dpi=200)
    for i in range(len(np.unique(y))):
        if labels is not None:
            plt.scatter(x[labels == i, 0], x[labels == i, 1], alpha=0.5)
        else:
            plt.scatter(x[y == i, 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='Misclassif
    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()

plotter(x, y)
```



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

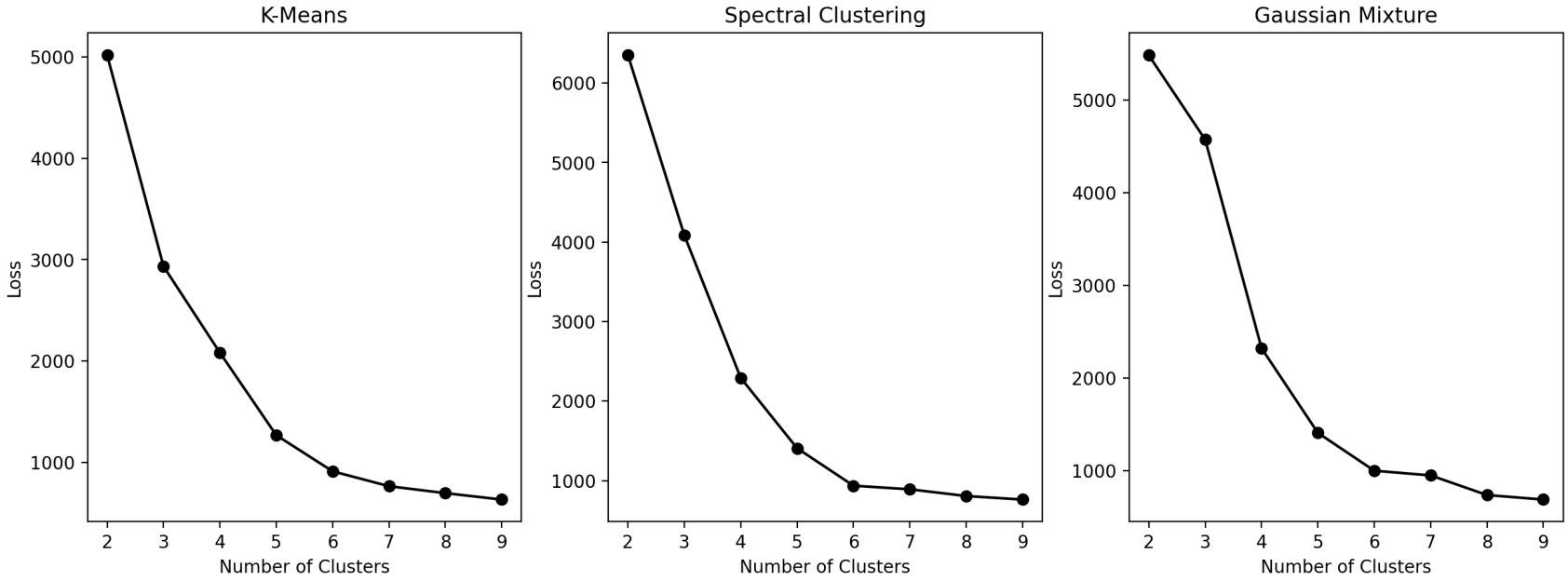
`plt.subplots` and the provided `plot_pred(x, labels, ax, title)` function.

In [35]:
```
## YOUR CODE GOES HERE
mdl = KMeans(n_clusters=6, init=np.array([[0,0], [2,2], [1,-2], [5,0], [-2,-1], [-1,3]]), n_init=1).fit(x)
centroids = mdl.cluster_centers_
labels = mdl.labels_
fig, ax = plt.subplots(1, 3, figsize=(15, 5))
plot_pred(x, labels, ax[0], 'K-Means')
mdl2 = SpectralClustering(n_clusters=6, affinity='nearest_neighbors', n_neighbors=25).fit(x)
labels2 = mdl2.labels_
plot_pred(x, labels2, ax[1], 'Spectral Clustering')
mdl3 = GaussianMixture(n_components=6, covariance_type='full', max_iter=100).fit(x)
plot_pred(x, mdl3.predict(x), ax[2], 'Gaussian Mixture')

plt.show()
```



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_pred(x, labels, ax, title)` function.

In [36]:
```
## YOUR CODE GOES HERE
n_clust = [2, 3, 4, 5, 6, 7, 8, 9]
kmeans_loss = []
spectral_loss = []
gmm_loss = []

for clust in n_clust:
    mdl = KMeans(n_clusters=clust, n_init=10).fit(x)
    labels = mdl.labels_
    kmeans_loss.append(compute_loss(x, labels))
    mdl2 = SpectralClustering(n_clusters=clust, affinity='nearest_neighbors', n_neighbors=25).fit(x)
    labels2 = mdl2.labels_
    spectral_loss.append(compute_loss(x, labels2))
    mdl3 = GaussianMixture(n_components=clust, covariance_type='full', max_iter=100).fit(x)
    gmm_loss.append(compute_loss(x, mdl3.predict(x)))

fig, ax = plt.subplots(1, 3, figsize=(15, 5))
plot_loss(kmeans_loss, ax[0], 'K-Means')
plot_loss(spectral_loss, ax[1], 'Spectral Clustering')
plot_loss(gmm_loss, ax[2], 'Gaussian Mixture')
plt.show()
```

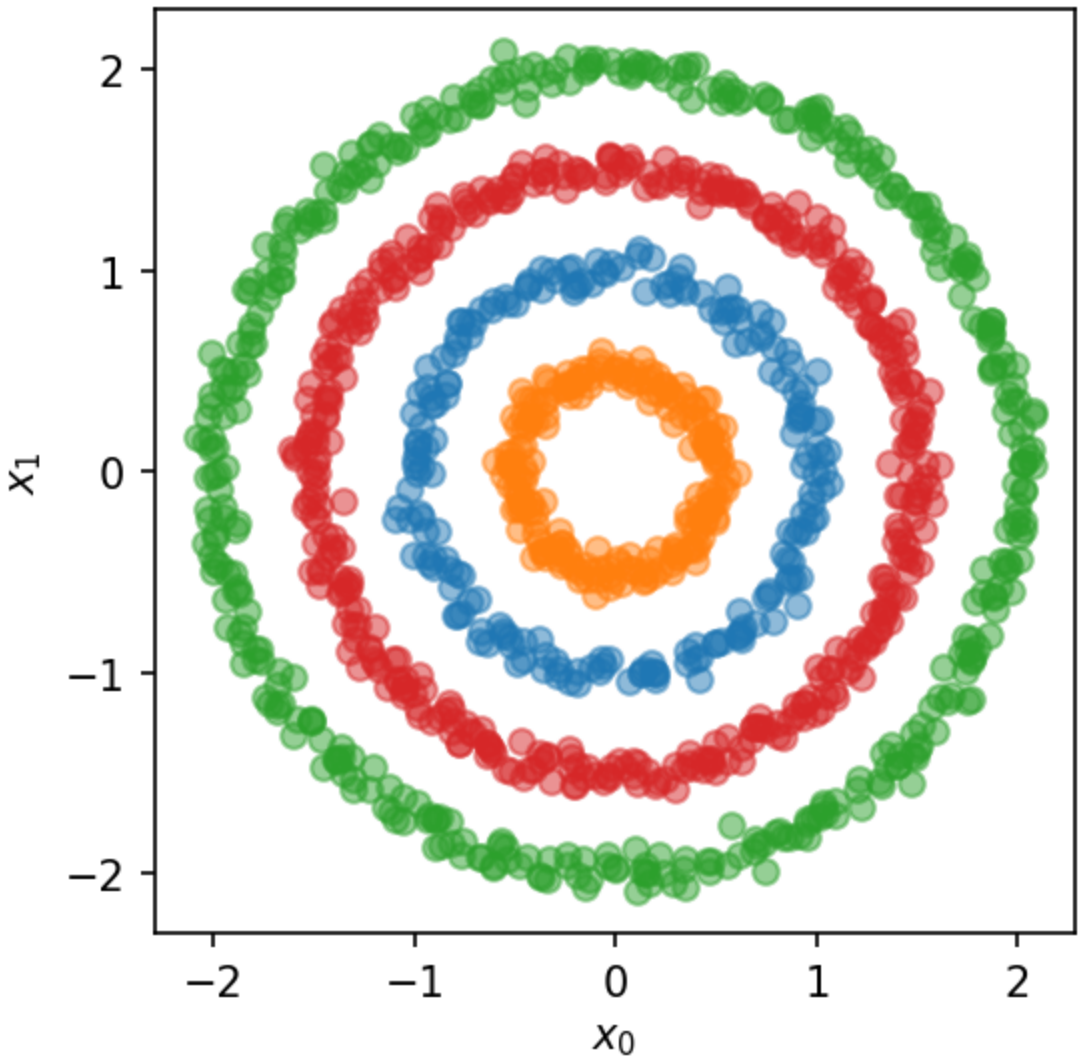


Concentric circles dataset

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`.

In [37]:
```
## 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])
```

In [38]:
```
## YOUR CODE GOES HERE
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)
```

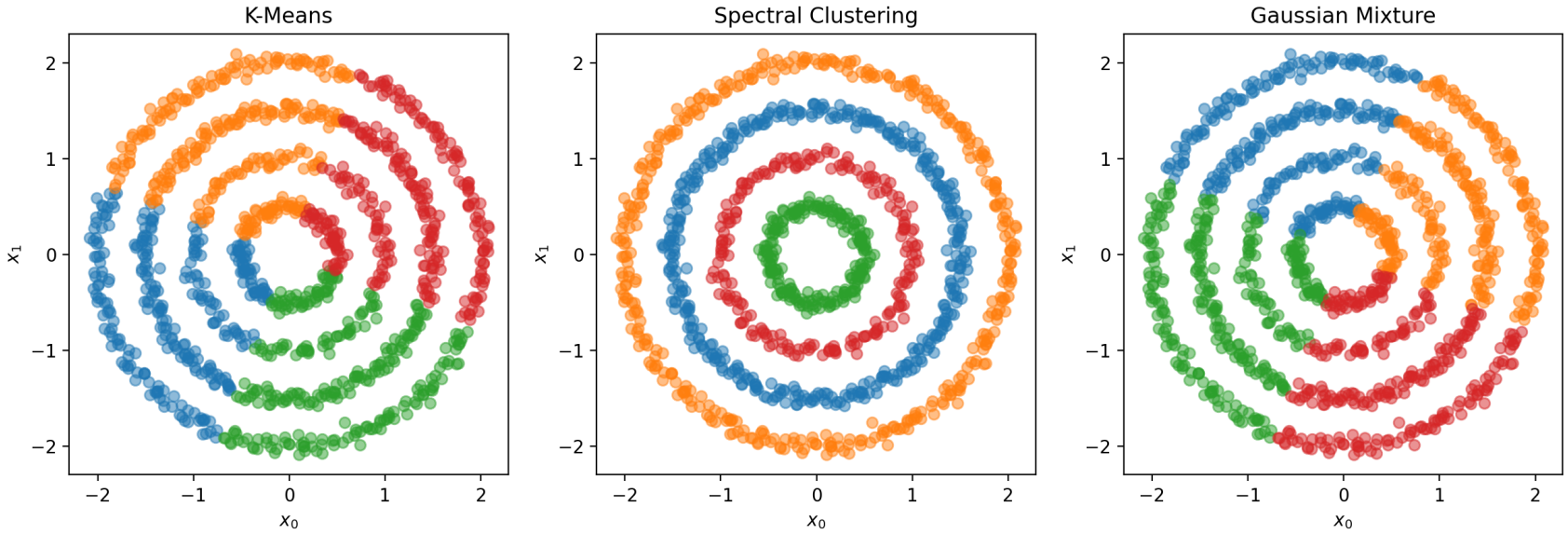


Use the `sklearn` KMeans, Spectral Clustering, and Gaussian Mixture Model functions to cluster the concentric circle data with 4 clusters, and attempt to modify the parameters until you get satisfactory results. Note: you should get good clustering results with Spectral Clustering, but the KMeans and 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.

In [39]:
```
## YOUR CODE GOES HERE
mdl = KMeans(n_clusters=4, n_init=10).fit(x)
```

```
fig, ax = plt.subplots(1, 3, figsize=(15, 5))
plotter(x, mdl.labels_, ax[0], title='K-Means')
mdl2 = SpectralClustering(n_clusters=4, affinity='nearest_neighbors', n_neighbors=10).fit(x)
plotter(x, mdl2.labels_, ax[1], title='Spectral Clustering')
mdl3 = GaussianMixture(n_components=4, covariance_type='full', max_iter=100).fit(x)
plotter(x, mdl3.predict(x), ax[2], title='Gaussian Mixture')

plt.show()
```

```
C:\Users\srech\AppData\Local\Programs\Python\Python311\Lib\site-packages\sklearn\manifold\_spectral_embedding.py:273:
UserWarning: Graph is not fully connected, spectral embedding may not work as expected.
  warnings.warn(
```

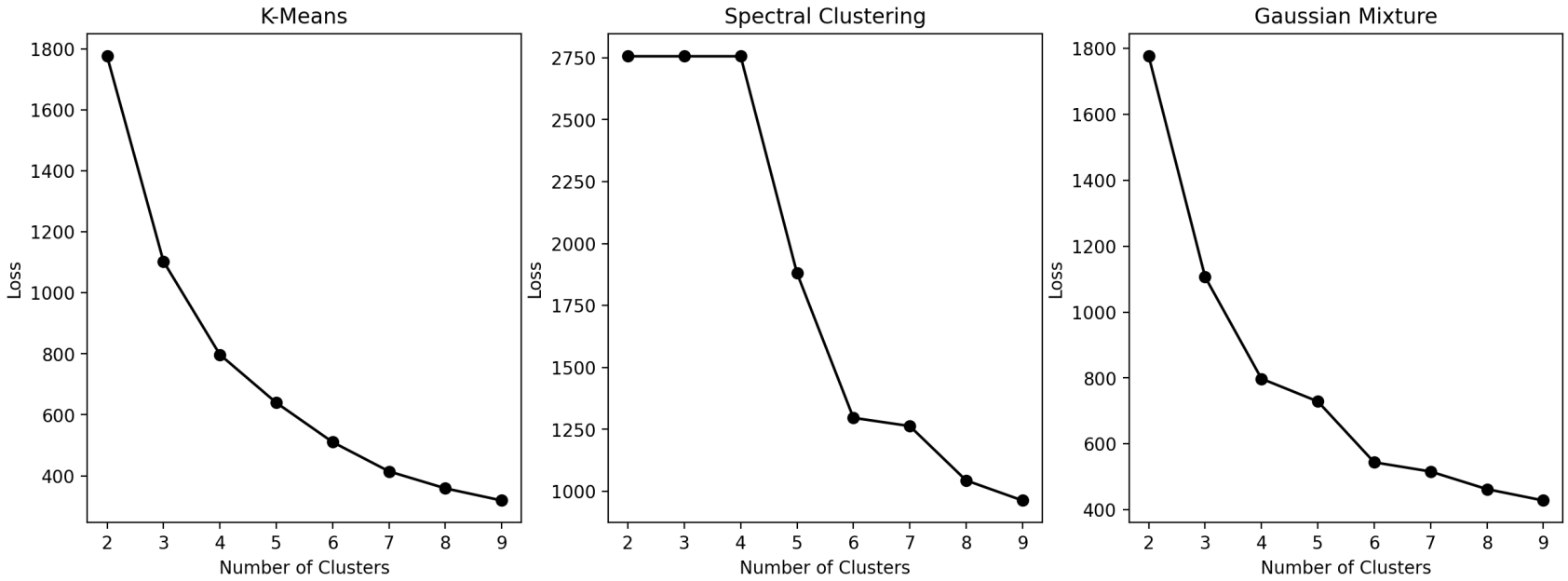


Using the parameters you found for the three models above, run each of the clustering algorithms for `n_clust = [2,3,4,5,6,7,8,9]` and compute the sum of squared distances loss for each case using the provided `compute_loss(x, labels)` function, where labels is the cluster assigned to each point by the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

In [40]:
```
## YOUR CODE GOES HERE
n_clust = [2, 3, 4, 5, 6, 7, 8, 9]
kmeans_loss = []
spectral_loss = []
gmm_loss = []
```

```
for clust in n_clust:
    mdl = KMeans(n_clusters=clust, n_init=10).fit(x)
    labels = mdl.labels_
    kmeans_loss.append(compute_loss(x, labels))
    mdl2 = SpectralClustering(n_clusters=clust, affinity='nearest_neighbors', n_neighbors=10).fit(x)
    labels2 = mdl2.labels_
    spectral_loss.append(compute_loss(x, labels2))
    mdl3 = GaussianMixture(n_components=clust, covariance_type='full', max_iter=100).fit(x)
    gmm_loss.append(compute_loss(x, mdl3.predict(x)))

fig, ax = plt.subplots(1, 3, figsize=(15, 5))
plot_loss(kmeans_loss, ax[0], 'K-Means')
plot_loss(spectral_loss, ax[1], 'Spectral Clustering')
plot_loss(gmm_loss, ax[2], 'Gaussian Mixture')

plt.show()
```

```
C:\Users\srech\AppData\Local\Programs\Python\Python311\Lib\site-packages\sklearn\manifold\_spectral_embedding.py:273:
UserWarning: Graph is not fully connected, spectral embedding may not work as expected.
  warnings.warn(
C:\Users\srech\AppData\Local\Programs\Python\Python311\Lib\site-packages\sklearn\manifold\_spectral_embedding.py:273:
UserWarning: Graph is not fully connected, spectral embedding may not work as expected.
  warnings.warn(
C:\Users\srech\AppData\Local\Programs\Python\Python311\Lib\site-packages\sklearn\manifold\_spectral_embedding.py:273:
UserWarning: Graph is not fully connected, spectral embedding may not work as expected.
  warnings.warn(
C:\Users\srech\AppData\Local\Programs\Python\Python311\Lib\site-packages\sklearn\manifold\_spectral_embedding.py:273:
UserWarning: Graph is not fully connected, spectral embedding may not work as expected.
  warnings.warn(
C:\Users\srech\AppData\Local\Programs\Python\Python311\Lib\site-packages\sklearn\manifold\_spectral_embedding.py:273:
UserWarning: Graph is not fully connected, spectral embedding may not work as expected.
  warnings.warn(
C:\Users\srech\AppData\Local\Programs\Python\Python311\Lib\site-packages\sklearn\manifold\_spectral_embedding.py:273:
UserWarning: Graph is not fully connected, spectral embedding may not work as expected.
  warnings.warn(
C:\Users\srech\AppData\Local\Programs\Python\Python311\Lib\site-packages\sklearn\manifold\_spectral_embedding.py:273:
UserWarning: Graph is not fully connected, spectral embedding may not work as expected.
  warnings.warn(
C:\Users\srech\AppData\Local\Programs\Python\Python311\Lib\site-packages\sklearn\manifold\_spectral_embedding.py:273:
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?

Your response goes here Clustering Performance on the "Blob" Dataset: The Gaussian Mixture Model exhibited the highest performance, followed by Spectral Clustering and then K-Means. With the elbow method, it was observed that Spectral Clustering showed a more distinct elbow compared to K-Means and Gaussian Mixture, indicating a clearer identification of the natural number of clusters.

2. 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?

Your response goes here Clustering Performance on the Concentric Circles Dataset: Spectral Clustering outperformed both K-Means and the Gaussian Mixture Model in this scenario. However, the elbow method was not effective in determining the natural number

of clusters for any of these methods on this particular dataset.

3. 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?

Your response goes here Effectiveness of Sum of Squared Distances as a Loss Function: The sum of squared distances generally served as a good loss function for all three clustering algorithms. However, its effectiveness varied with the type of dataset; particularly, it was less successful for the Spectral Clustering method when applied to the concentric circles dataset. This reflects the limitations of this loss function in certain complex clustering scenarios.

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:

In [20]:
```
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')
```

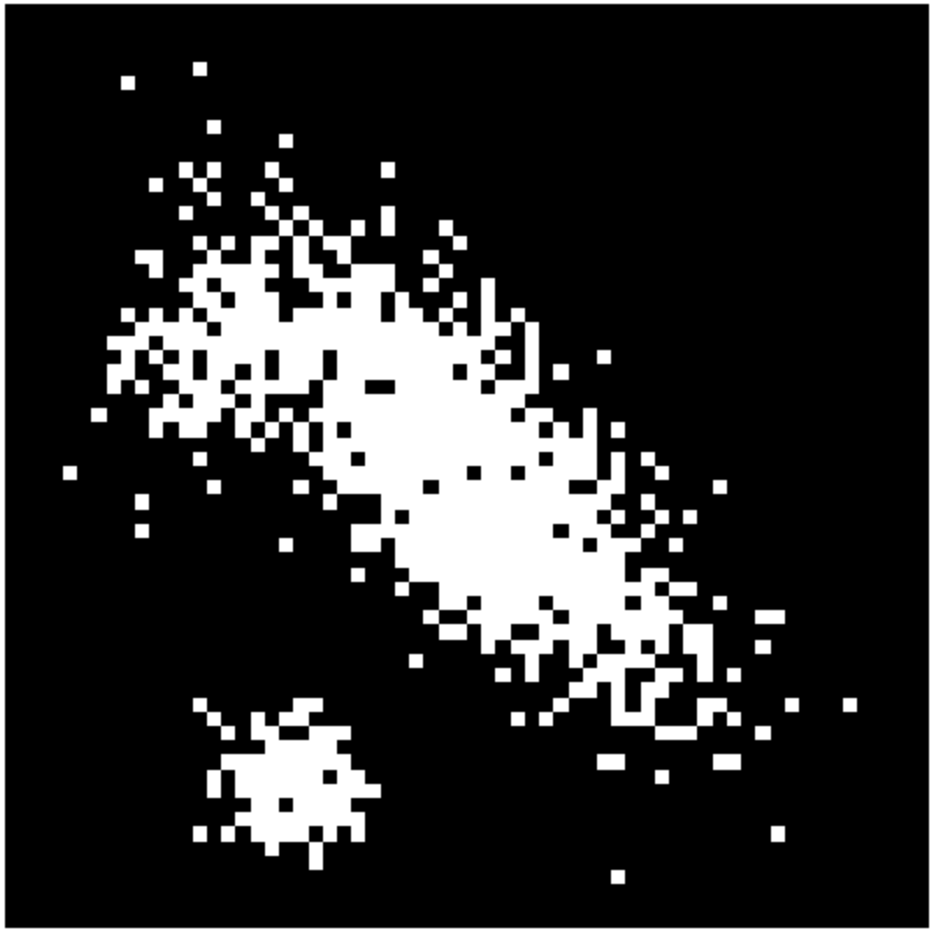
```
    ax.axis('off')

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.

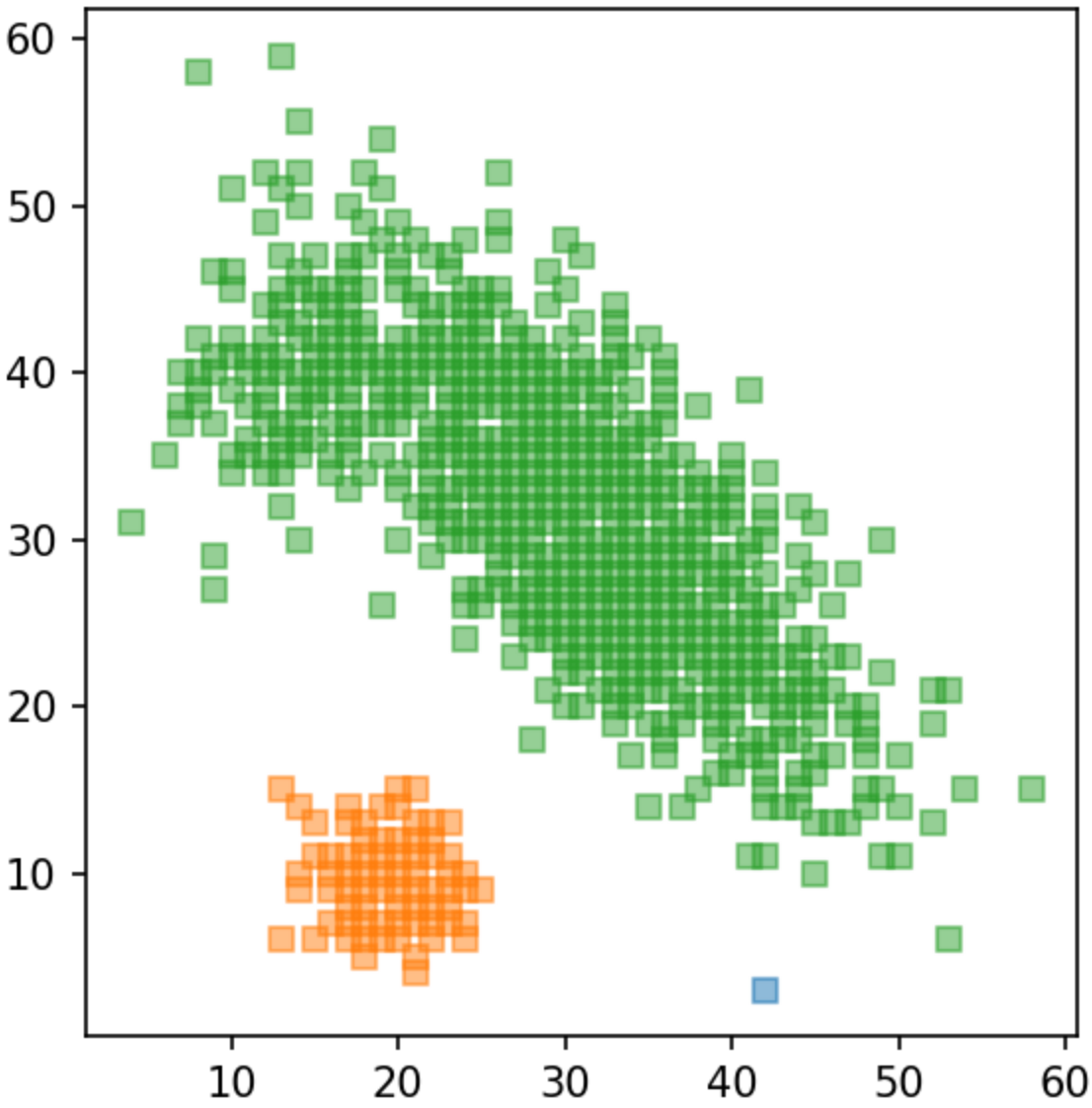
Note: you will use the integer coordinates for clustering, the bitmap is just for visualizing the data.

In [21]:
```
## YOUR CODE GOES HERE
data1 = np.loadtxt(r"C:\Users\srech\Downloads\m11-hw1-data1.txt", dtype=int)
bitmap1 = points_to_bitmap(data1)
plot_bitmap(bitmap1)
```



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

In [23]:
```
## YOUR CODE GOES HERE
dbscan1 = DBSCAN(eps=7, min_samples=11)
labels1 = dbscan1.fit_predict(data1)
plot_points(data1, labels1)
```



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.

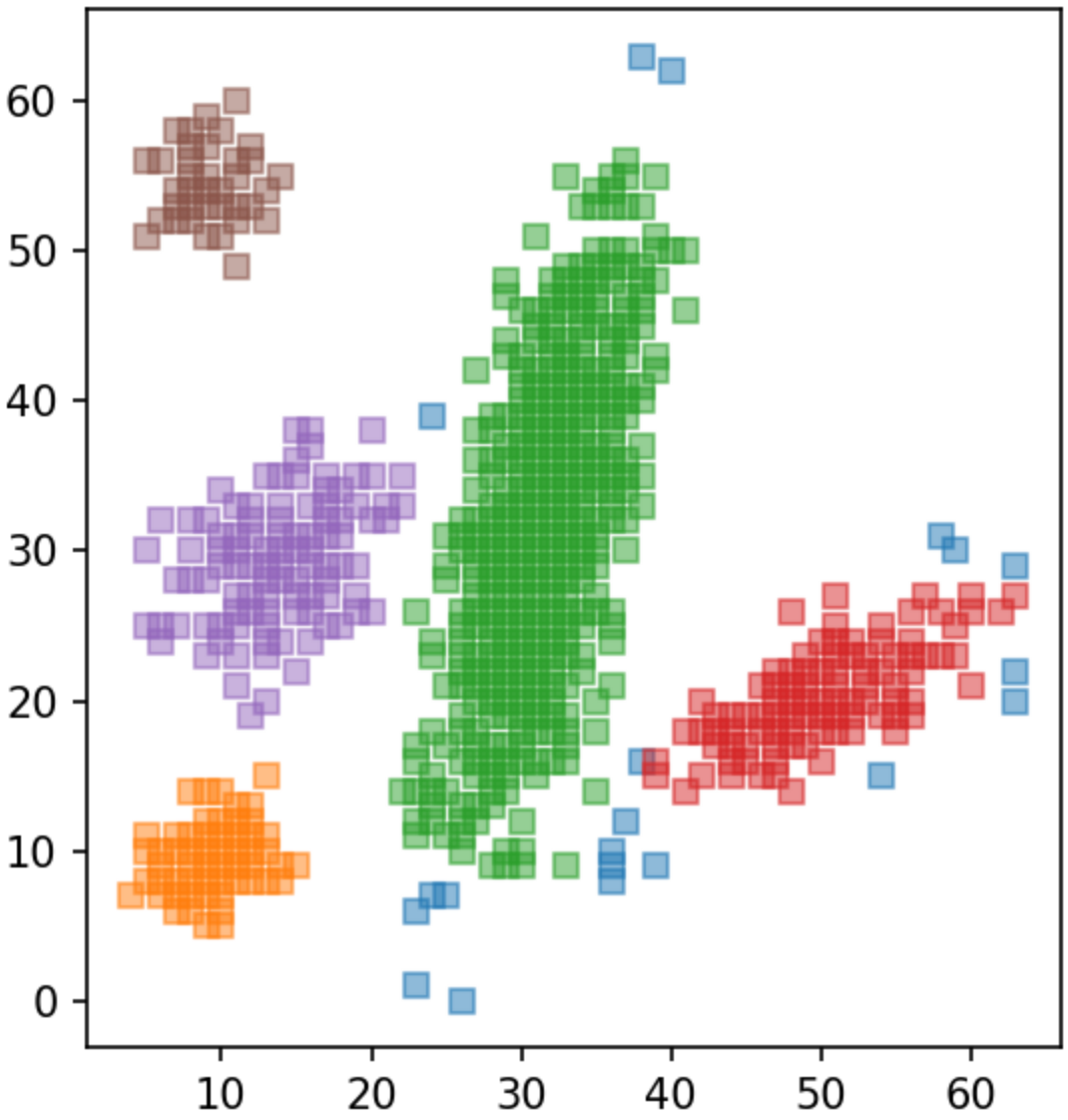
In [24]:
```
## YOUR CODE GOES HERE
data2 = np.loadtxt(r"C:\Users\srech\Downloads\m11-hw1-data2.txt", dtype=int)
bitmap2 = points_to_bitmap(data2)
plot_bitmap(bitmap2)
```

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.

In [25]:
```
## YOUR CODE GOES HERE
dbscan2 = DBSCAN(eps=3, min_samples=6)
labels2 = dbscan2.fit_predict(data2)
plot_points(data2, labels2)
```



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?

Your response goes here

Parameter Tuning Process: The tuning of eps and min_samples parameters for both models involved a methodical process of trial and error, followed by a careful visual inspection to confirm the effectiveness of chosen values. This hands-on approach allowed for an intuitive understanding of the impact of these parameters on the clustering results.

Number of Clusters Identified: In the clustering process, the first melt pool image revealed the presence of 2 distinct clusters, while the second image demonstrated a more complex structure with 5 clusters. This difference in cluster count reflects the varying complexities and characteristics of each image.

Variability in Parameter Ranges for Successful Clustering: The first melt pool image, with a simpler cluster arrangement, allowed for a broader range of eps and min_samples values to successfully identify clusters. In contrast, the second melt pool image, characterized by clusters that were closer to each other, necessitated a more precise and constrained range of these parameters. The proximity of clusters in the second image meant that a wider range could potentially merge distinct clusters or fail to differentiate closely situated ones, thus requiring more careful tuning to accurately distinguish each cluster.