Name: Barathkrishna Satheeshkumar
Andrew ID: bsathees

Problem 1

Q1

Cluster Center 1: 5 points
Cluster Center 2: 4 points

Q2

2: Left, Right

Problem 2

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.

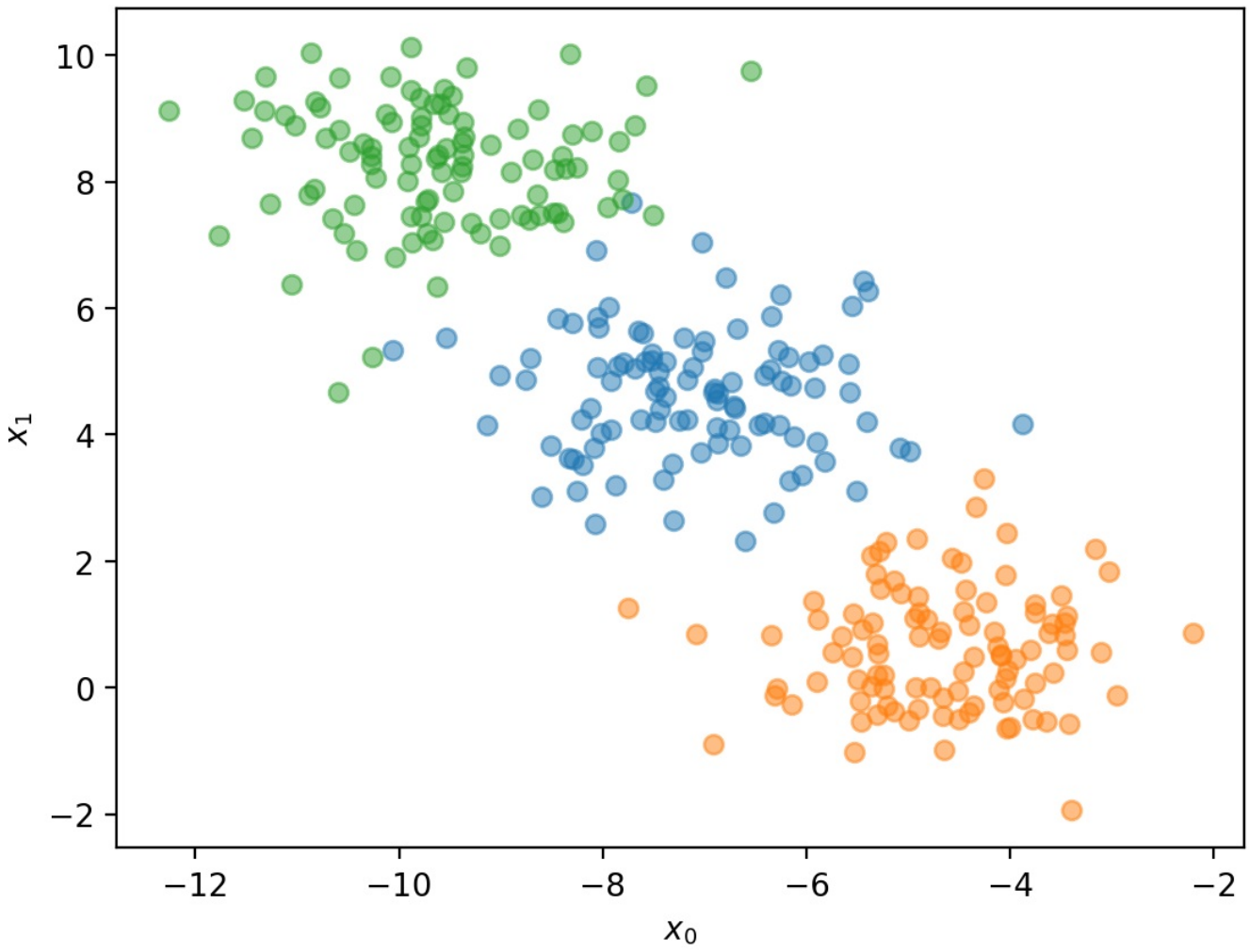
In [1]:
```
import numpy as np
import matplotlib.pyplot as plt

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 =
    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 [2]:
```
## 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 [3]:
```
## 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 [4]:
```
## FILL IN THE FOLLOWING FUNCTION
def find_cluster(point, centers):
    distances = np.linalg.norm(centers - point, 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 [5]:
```
## FILL IN THE FOLLOWING FUNCTION
def assign_labels(x, centers):
    labels = np.zeros(x.shape[0], dtype=int)
    for i, point in enumerate(x):
        labels[i] = find_cluster(point, 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

In [6]:
```
## FILL IN THE FOLLOWING FUNCTION
def update_centers(x, labels):
    num_clusters = len(np.unique(labels))
```

```
    new_centers = np.zeros((num_clusters, x.shape[1]))

    for k in range(num_clusters):
        cluster_points = x[labels == k]
        if len(cluster_points) > 0:
            new_centers[k] = cluster_points.mean(axis=0)

    return new_centers
```

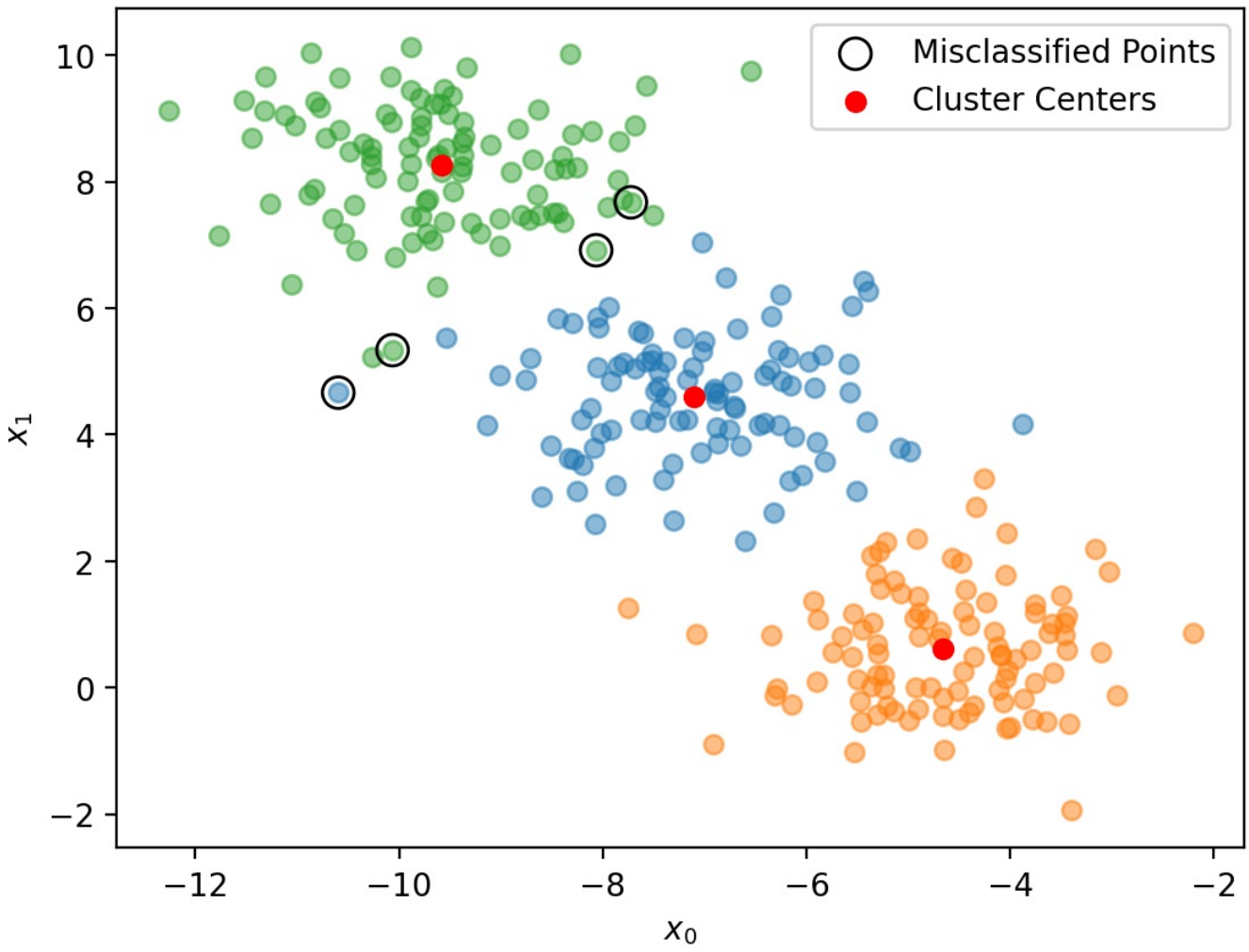
Finally write a function `myKMeans(x, init_centers)` which will run the KMeans algorithm, provided all the points in `x` and the coordinates of the initial cluster centers in `init_centers`. Run the algorithm until there is no change in cluster membership in subsequent iterations. Your function should 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 [7]:
```
## FILL IN THE FOLLOWING FUNCTION
def myKMeans(x, init_centers):
    centers = init_centers
    labels = np.zeros(x.shape[0], dtype=int)

    while True:
        new_labels = assign_labels(x, centers)
        if np.array_equal(labels, new_labels):
            break
        labels = new_labels
        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.

In [9]:
```
## YOUR CODE GOES HERE
init_centers = np.array([[-5, 5], [0, 0], [-10, 10]])
labels, centers = myKMeans(x, init_centers)

plotter(x, y, labels, centers)
```



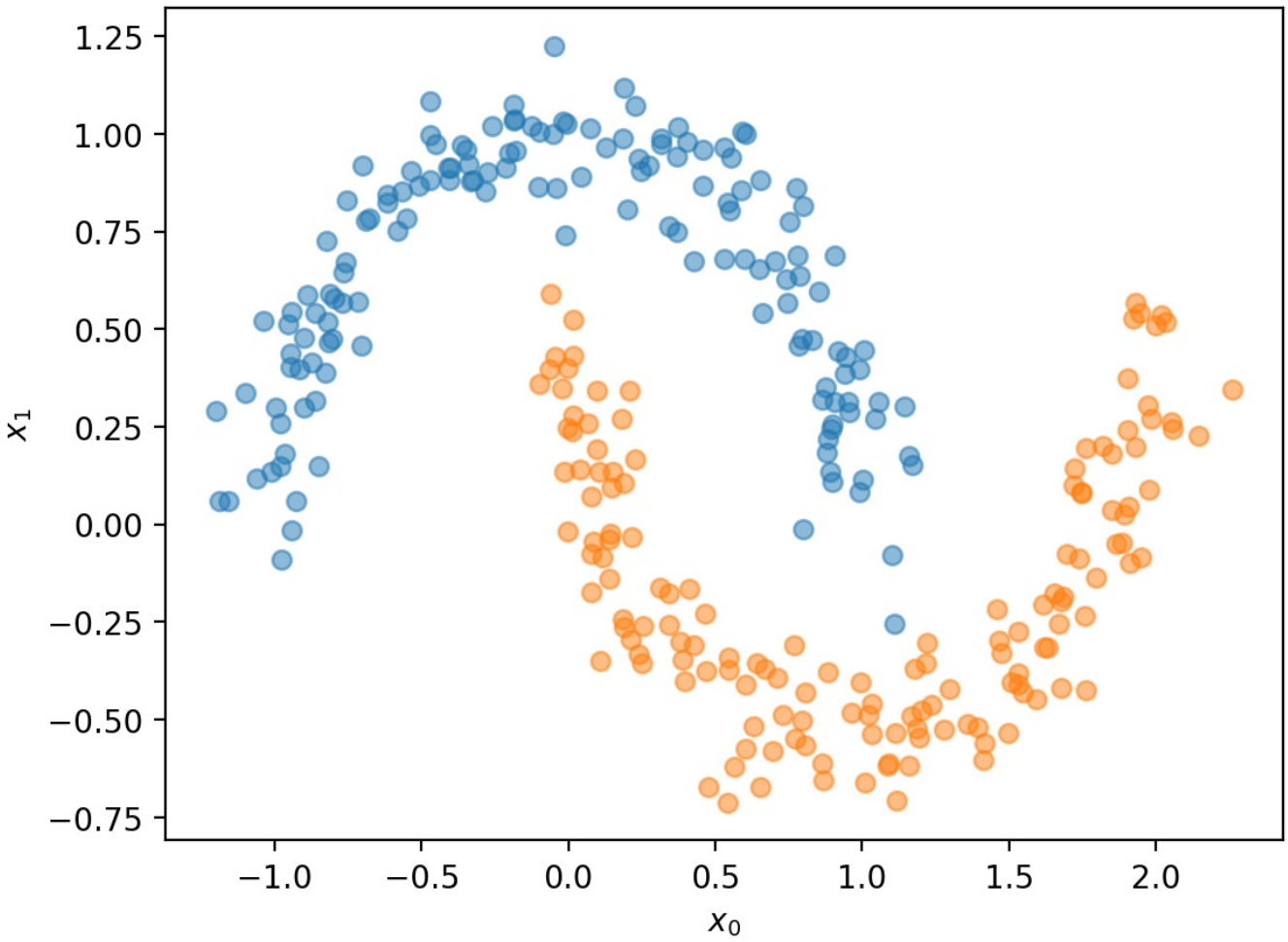
Moon Dataset

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

In [10]:
```
## DO NOT MODIFY
x,y = make_moons(n_samples = 300, noise = 0.1, random_state = 0)
```

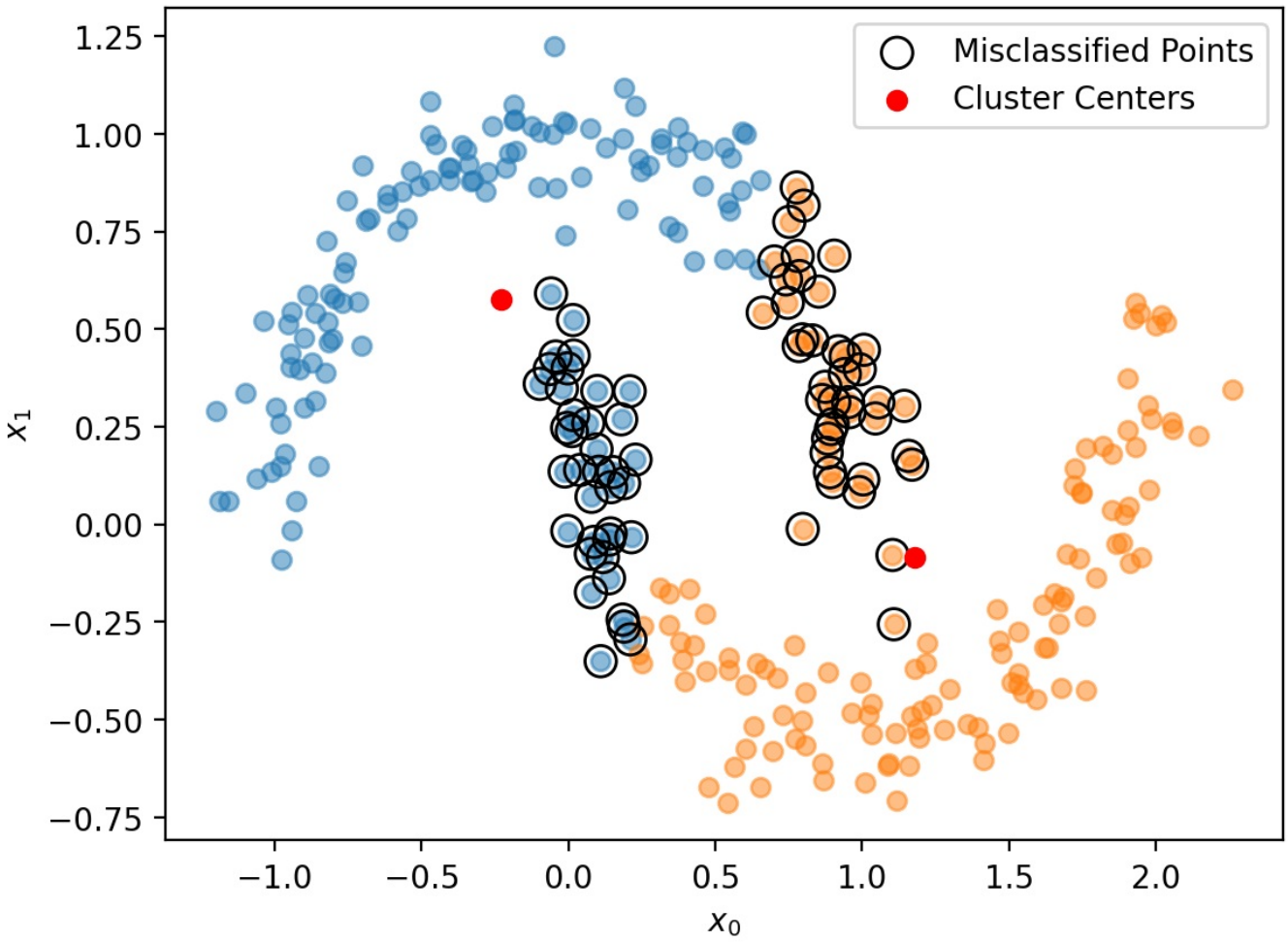
Visualize the data using the `plotter(x,y)` function.

In [11]:
```
## 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 [12]:
```
## YOUR CODE GOES HERE
init_centers = np.array([[0,1],[1,-0.5]])
labels, centers = myKMeans(x, init_centers)

plotter(x, y, labels, centers)
```



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.

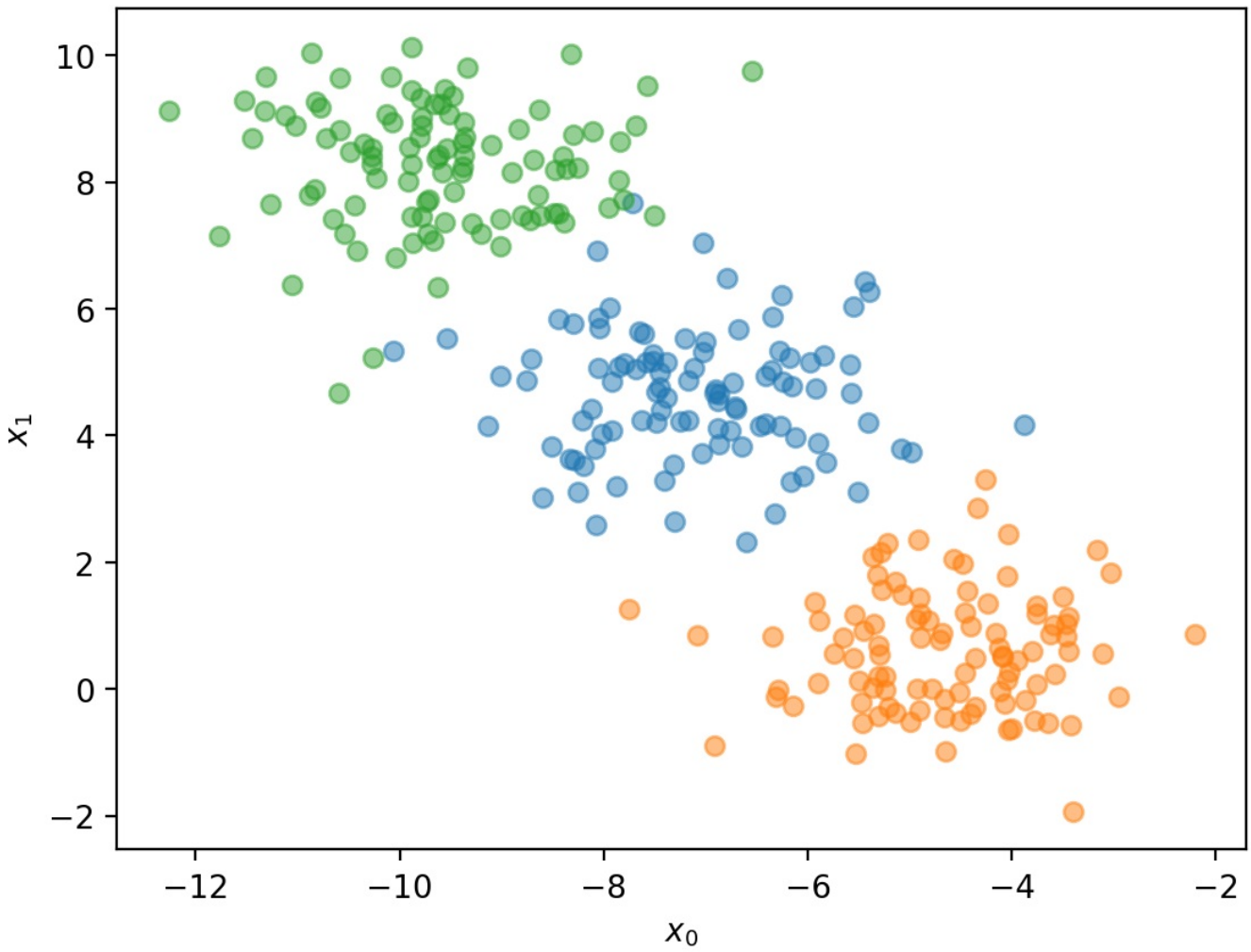
In [1]:
```
import numpy as np
import matplotlib.pyplot as plt

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 =
    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 [8]:
```
## DO NOT MODIFY
x, y = make_blobs(n_samples = 300, n_features = 2, random_state = 12)
```

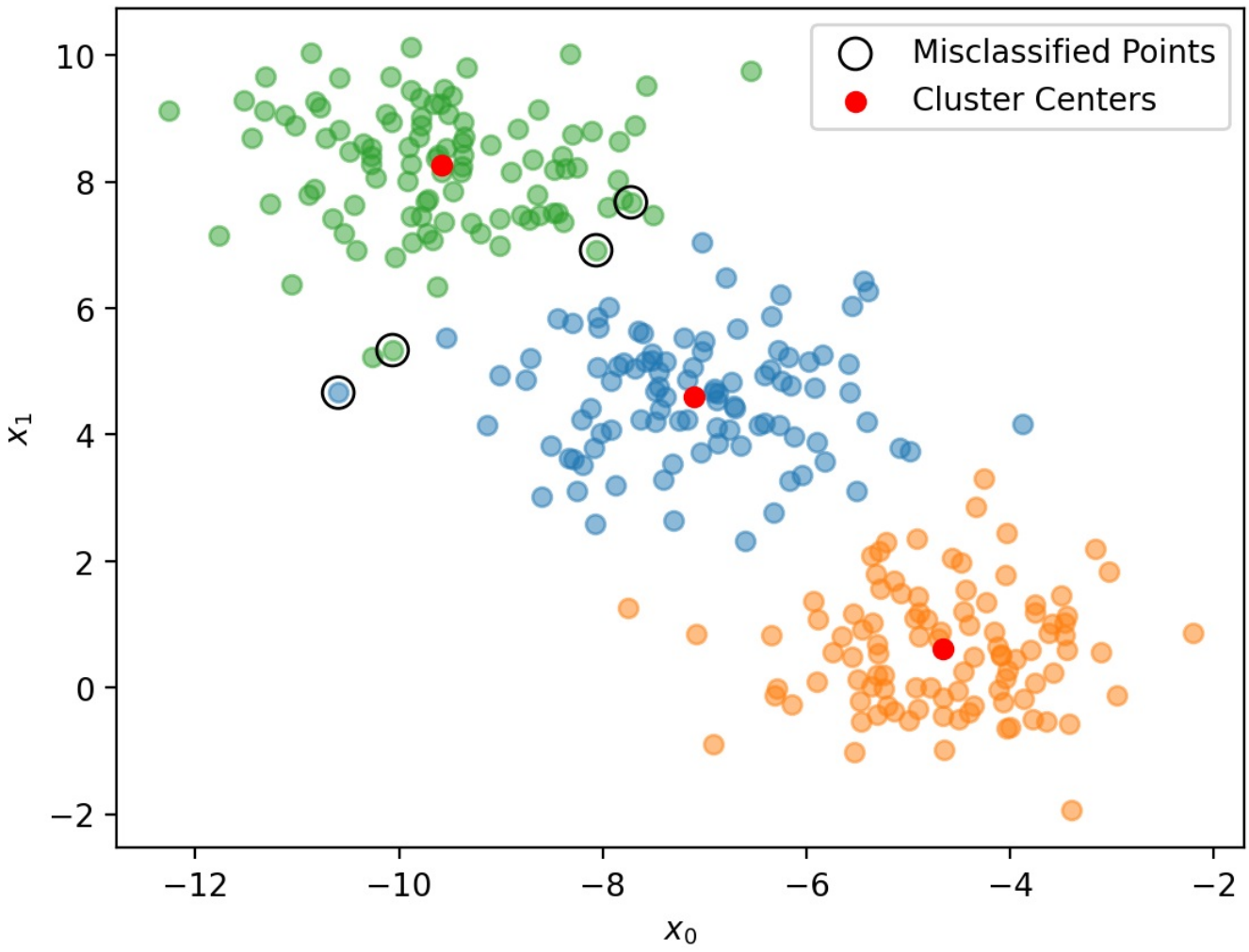
Visualize the data using the `plotter(x,y)` function. You do not need to pass the `labels` or `centers` arguments

In [9]:
```
## 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 [10]:
```
## YOUR CODE GOES HERE
init_centers = np.array([[-5, 5], [0, 0], [-10, 10]])

kmeans = KMeans(n_clusters = 3, init = init_centers, n_init = 1)
kmeans.fit(x)

labels = kmeans.labels_
centers = kmeans.cluster_centers_

plotter(x, y, labels, centers)
```



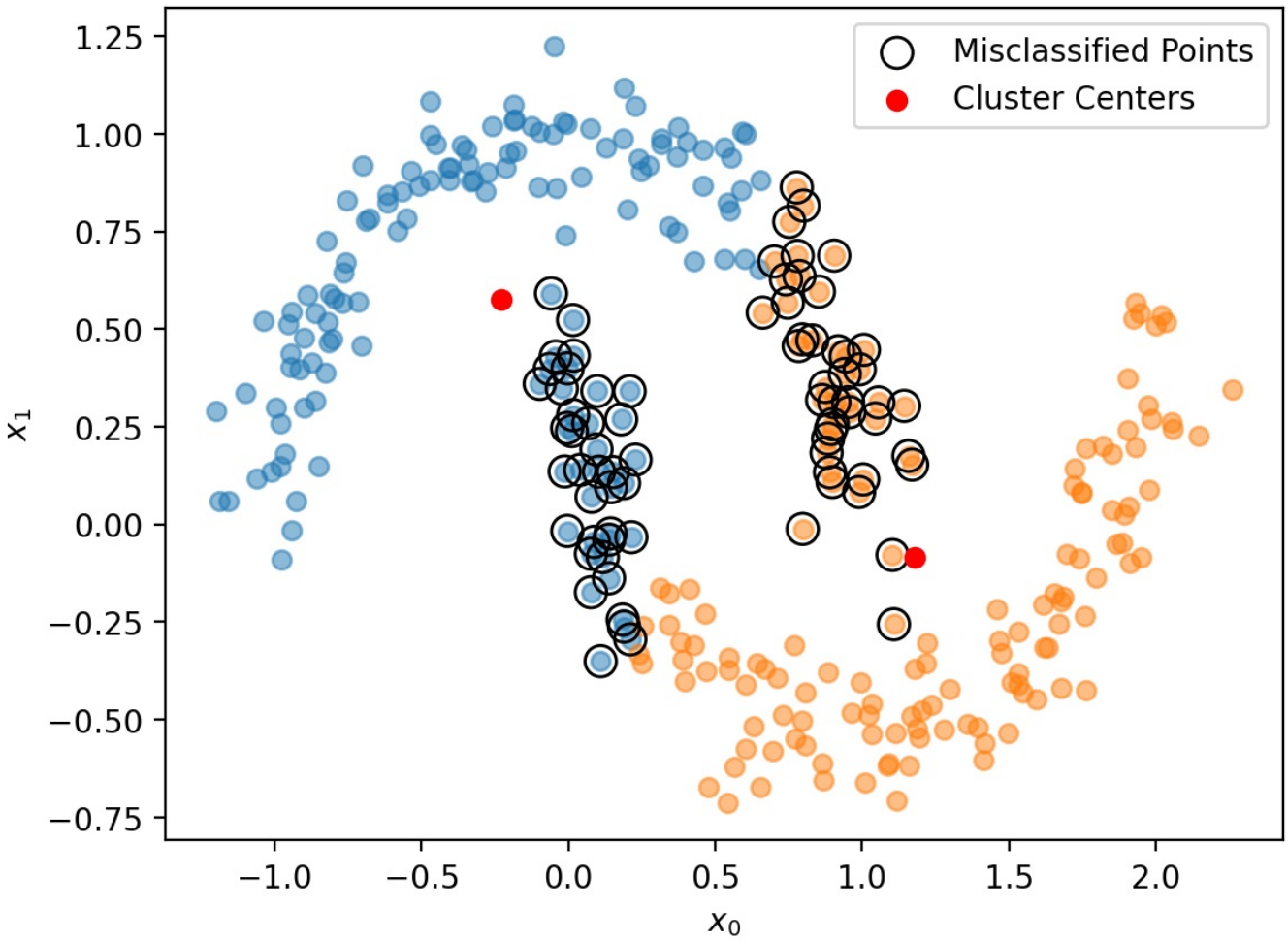
Moon Dataset

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

In [11]:
```
## 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 [12]:
```
## YOUR CODE GOES HERE
init_centers = np.array([[0,1],[1,-0.5]])

kmeans = KMeans(n_clusters = 2, init = init_centers, n_init = 1, random_state=42)
kmeans.fit(x)

labels = kmeans.labels_
centers = kmeans.cluster_centers_

plotter(x, y, labels, centers)
```



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.

The results of the hand-coded K-Means implementation and the sklearn implementation seems to be identical. The reasons could be the following:

1. Initial cluster centres are the same for both implementations
2. Both implementations stop iterating when there is no change in cluster membership, which is the default behavior in sklearn and was implemented in the hand-coded version.
3. Both implementations use the Euclidean distance, which is standard for K-Means.
4. We used n_init=1 in the sklearn implementation to avoid multiple runs with random initializations and ensure direct comparison.

M11-L1 Problem 3

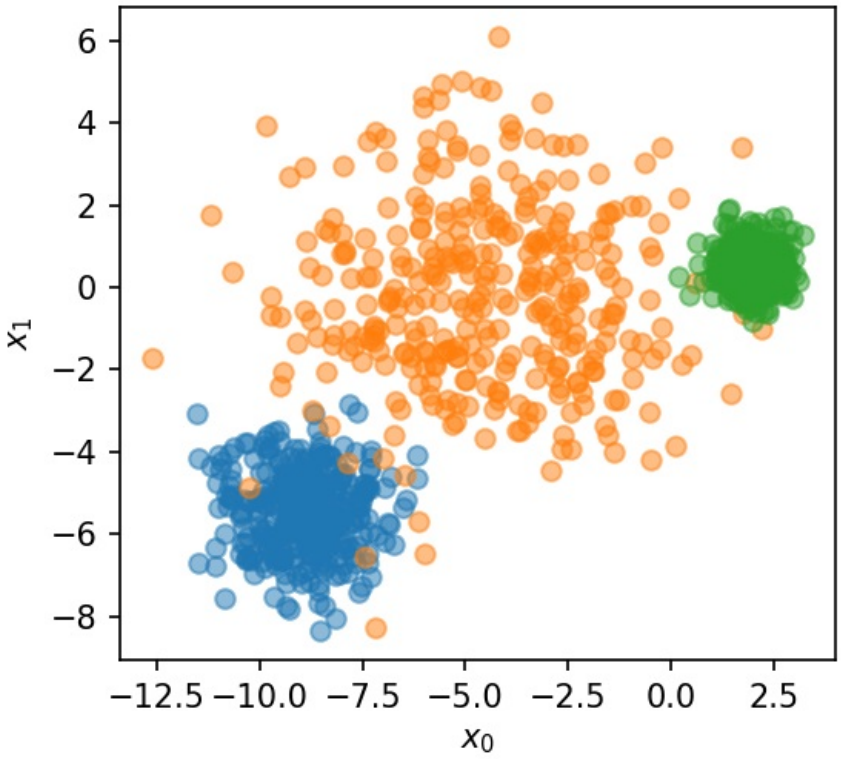
In this problem you will use the `sklearn` implementation of hierarchical clustering with three different linkage criteria `('single', 'complete', 'average')` to clusters two datasets: a "blob" shaped dataset with three classes, and a concentric circle dataset with two classes.

In [1]:
```
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 [4]:
```
## DO NOT MODIFY
x, labels = make_blobs(n_samples = 1000, cluster_std=[1.0, 2.5, 0.5], random_state = 170)
```

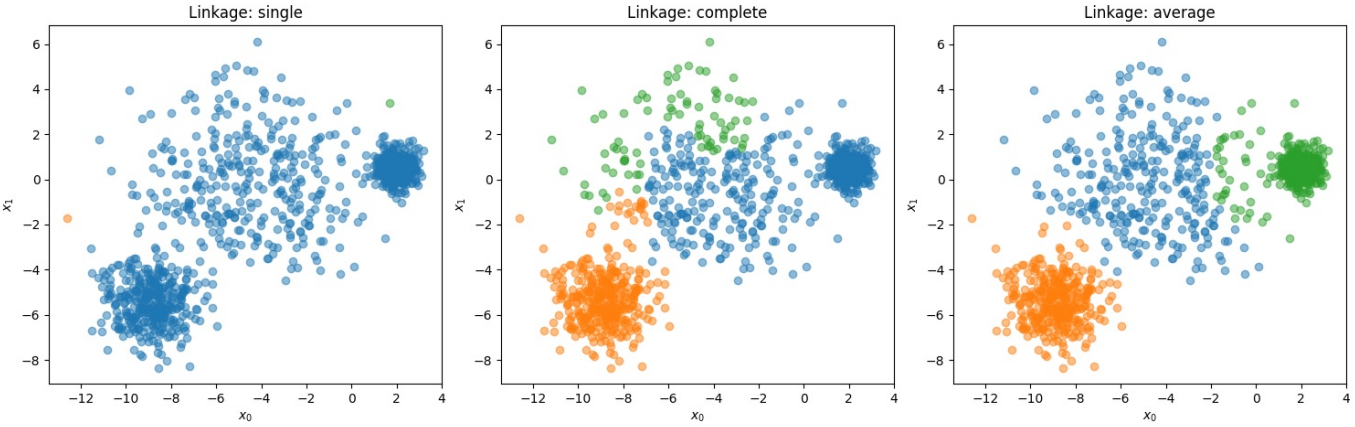
In [3]:
```
## YOUR CODE GOES HERE
plotter(x, labels)
```



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.

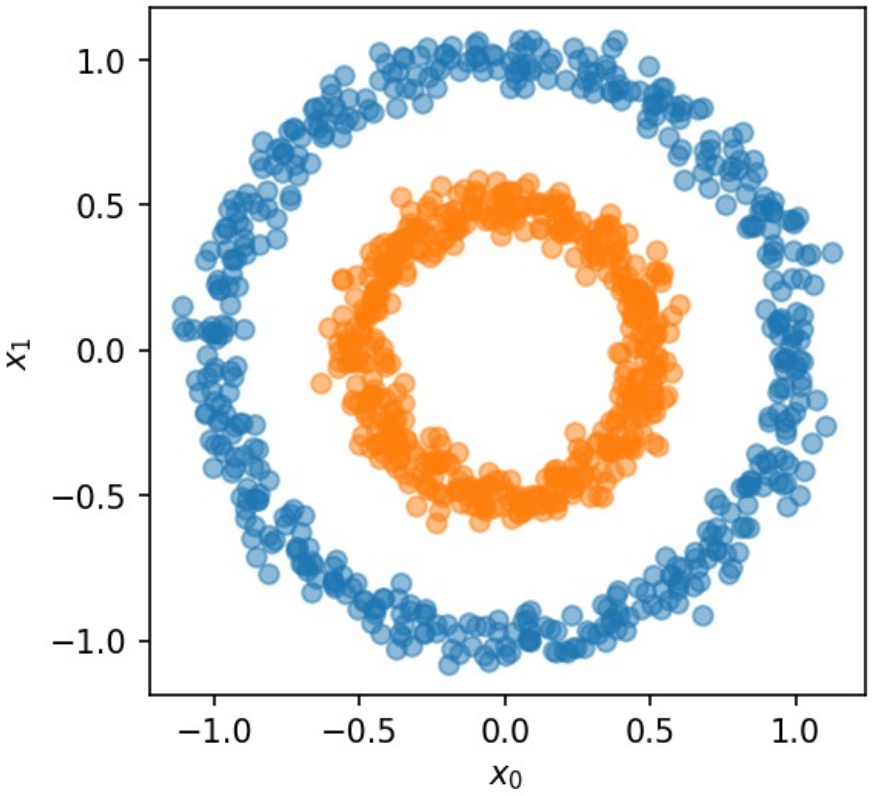
In [ ]:
```
## YOUR CODE GOES HERE
linkage_methods = ['single', 'complete', 'average']

fig, ax = plt.subplots(1, 3, figsize=(15, 5))
for i, linkage in enumerate(linkage_methods):
    model = AgglomerativeClustering(n_clusters=3, linkage=linkage)
    predicted_labels = model.fit_predict(x)

    plotter(x, predicted_labels, ax[i], title=f"Linkage: {linkage}")

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



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

In [6]:
```
## DO NOT MODIFY
x, labels = make_circles(1000, factor = 0.5, noise = 0.05, random_state = 0)
```

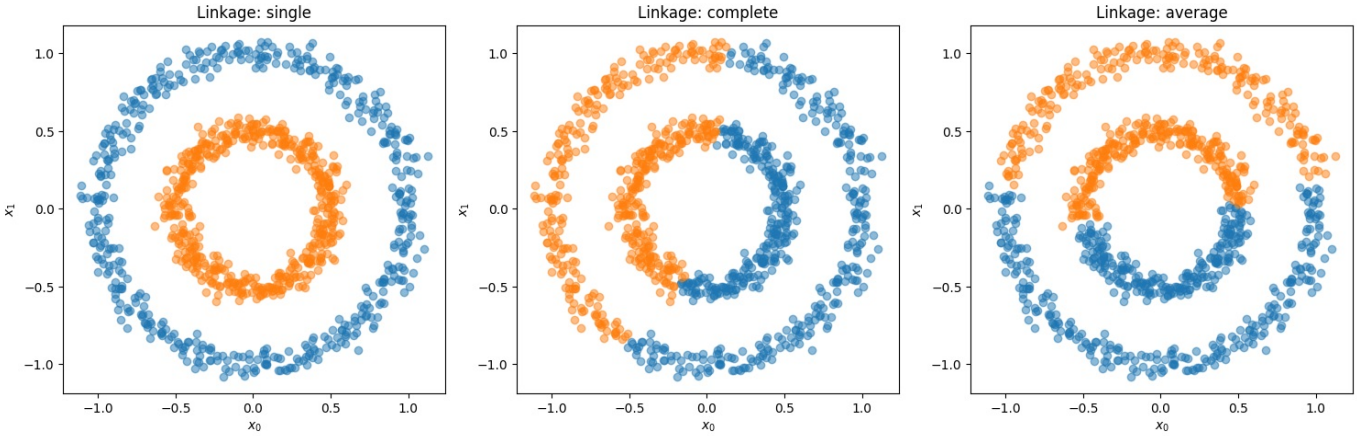
In [7]:
```
## YOUR CODE GOES HERE
plotter(x, labels)
```



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 [9]:
```
## YOUR CODE GOES HERE
fig, ax = plt.subplots(1, 3, figsize=(15, 5))
for i, linkage in enumerate(linkage_methods):
    model = AgglomerativeClustering(n_clusters=2, linkage=linkage)
    predicted_labels = model.fit_predict(x)
```

```
    plotter(x, predicted_labels, ax[i], title=f"Linkage: {linkage}")

plt.tight_layout()
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?

The blob dataset favors compact and spherical cluster criteria like complete or average linkage. The concentric circle dataset, with its non-convex structure, challenges these linkage methods, highlighting their preference for compactness over the global shape of clusters. The performance of linkage criteria depends on how well they align with the dataset's inherent structure. Non-convex datasets often require specialized clustering methods for better results.

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.

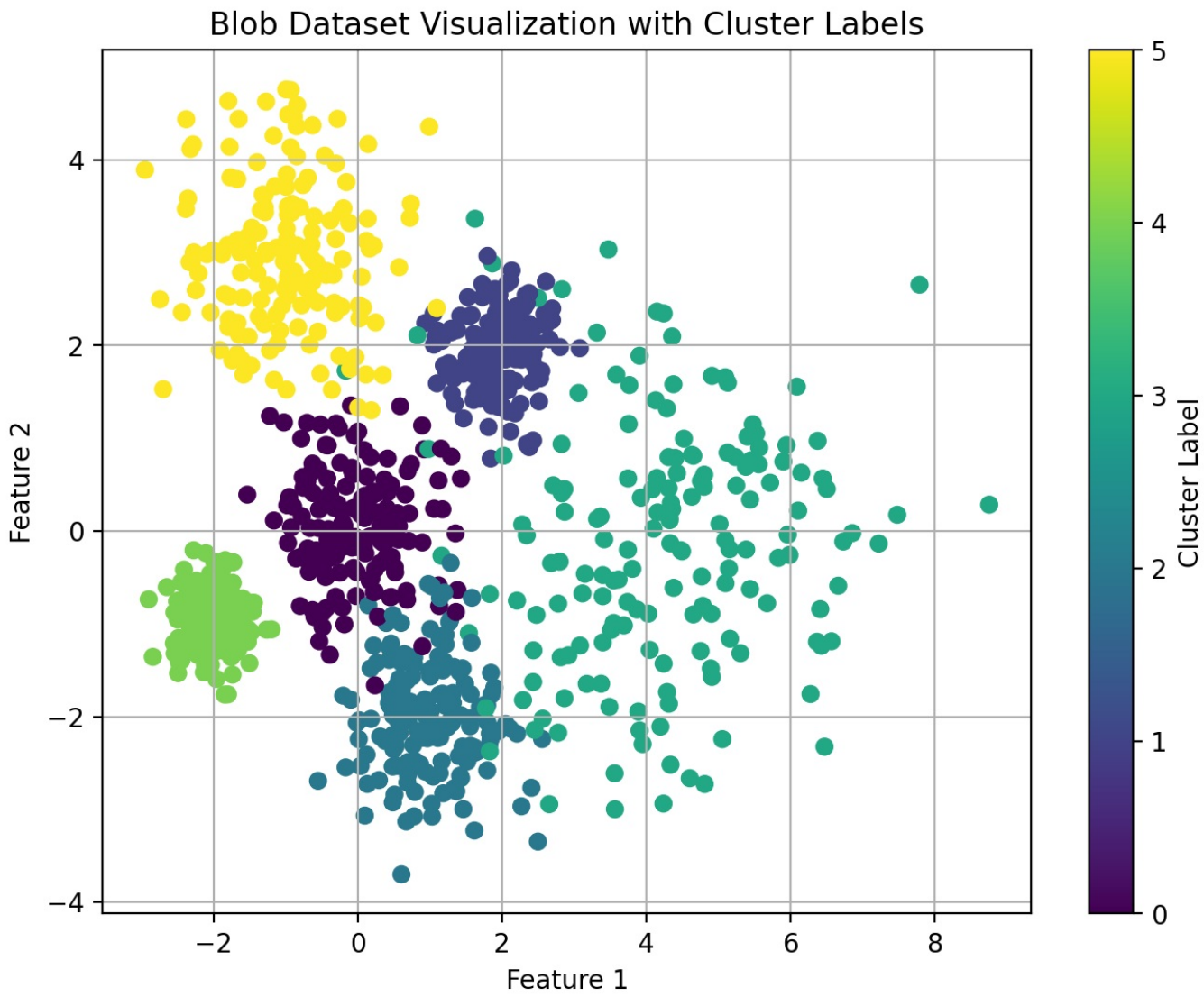
In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
plt.rcParams['figure.dpi'] = 200

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` .

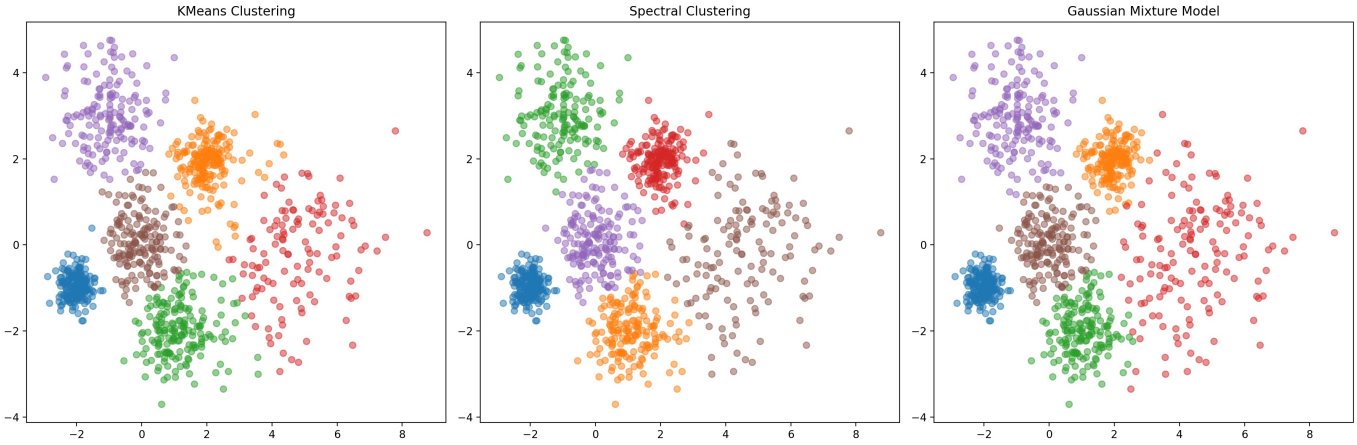
In [2]:
```
## DO NOT MODIFY
x, y = make_blobs(n_samples = 1000, n_features = 2, centers = [[0,0],[2,2],[1,-2],[4,0],[-2,-1],[-1,3]], cluste
```

In [5]:
```
## YOUR CODE GOES HERE
plt.figure(figsize=(8, 6))
plt.scatter(x[:, 0], x[:, 1], c=y)
plt.title('Blob Dataset Visualization with Cluster Labels')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.colorbar(label='Cluster Label')
plt.grid(True)
plt.show()
```



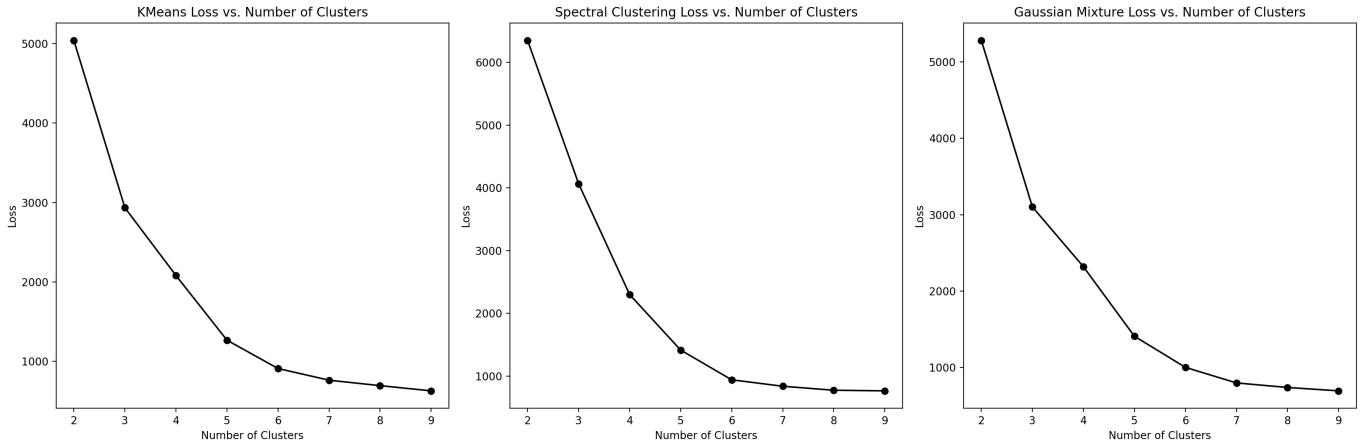
Use the `sklearn` KMeans, Spectral Clustering, and Gaussian Mixture Model functions to cluster the "blob" data with 6 clusters, and modify the parameters until you get satisfactory results. Plot the results of your three models side-by-side using `plt.subplots` and the provided `plot_pred(x, labels,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

In [7]:
```
## YOUR CODE GOES HERE
kmeans = KMeans(n_clusters=6, n_init=10, random_state=0)
kmeans_labels = kmeans.fit_predict(x)

spectral = SpectralClustering(n_clusters=6, affinity='nearest_neighbors', random_state=0)
spectral_labels = spectral.fit_predict(x)

gmm = GaussianMixture(n_components=6, random_state=0)
gmm_labels = gmm.fit_predict(x)

fig, ax = plt.subplots(1, 3, figsize=(18, 6))
plot_pred(x, kmeans_labels, ax[0], 'KMeans Clustering')
plot_pred(x, spectral_labels, ax[1], 'Spectral Clustering')
plot_pred(x, gmm_labels, ax[2], 'Gaussian Mixture Model')
plt.tight_layout()
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_loss(x, labels, ax, title)` function.

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

kmeans_losses = []
spectral_losses = []
gmm_losses = []


for n in n_clust:
    kmeans = KMeans(n_clusters=n, n_init=10, random_state=0)
    kmeans_labels = kmeans.fit_predict(x)
    kmeans_loss = compute_loss(x, kmeans_labels)
    kmeans_losses.append(kmeans_loss)

    spectral = SpectralClustering(n_clusters=n, affinity='nearest_neighbors', random_state=0)
    spectral_labels = spectral.fit_predict(x)
    spectral_loss = compute_loss(x, spectral_labels)
    spectral_losses.append(spectral_loss)

    gmm = GaussianMixture(n_components=n, random_state=0)
    gmm_labels = gmm.fit_predict(x)
    gmm_loss = compute_loss(x, gmm_labels)
    gmm_losses.append(gmm_loss)

fig, ax = plt.subplots(1, 3, figsize=(18, 6))

plot_loss(kmeans_losses, ax=ax[0], title='KMeans Loss vs. Number of Clusters')
plot_loss(spectral_losses, ax=ax[1], title='Spectral Clustering Loss vs. Number of Clusters')
plot_loss(gmm_losses, ax=ax[2], title='Gaussian Mixture Loss vs. Number of Clusters')

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

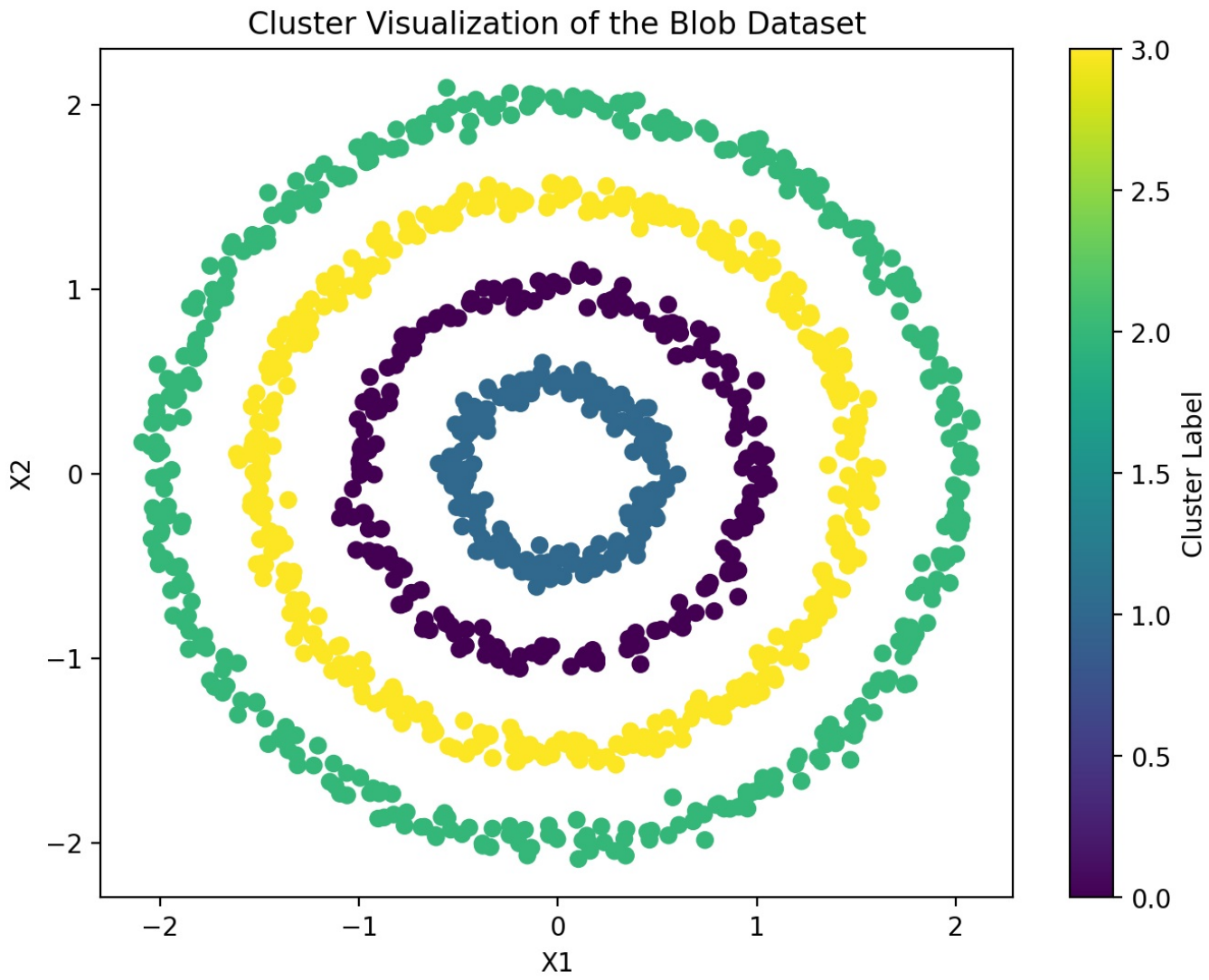


Concentric circles dataset

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`.

In [10]:
```
## 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])
```

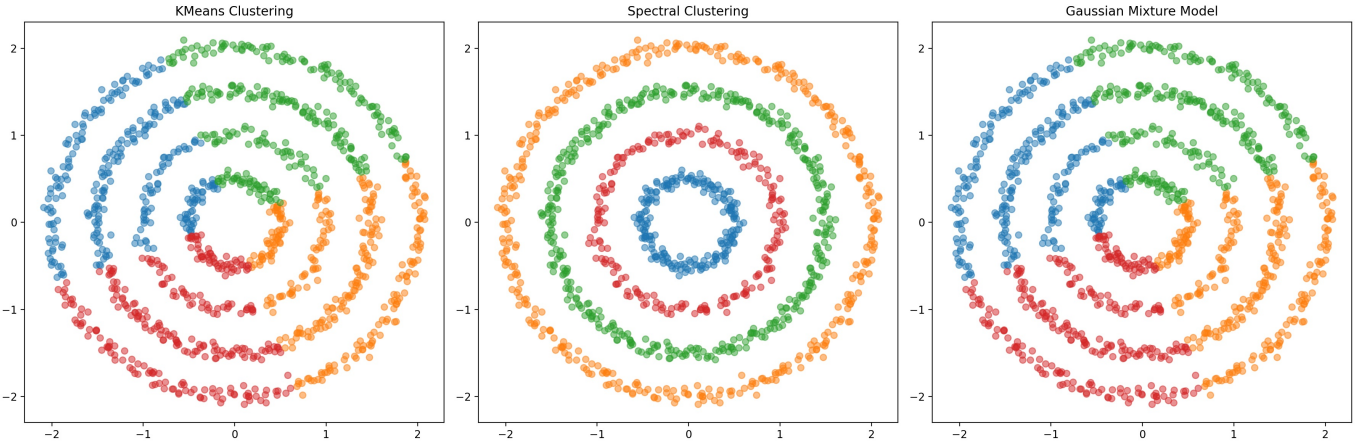
In [12]:
```
## YOUR CODE GOES HERE
plt.figure(figsize=(8, 6))
scatter = plt.scatter(x[:, 0], x[:, 1], c=y)
plt.title('Cluster Visualization of the Blob Dataset')
plt.xlabel('X1')
plt.ylabel('X2')
plt.colorbar(scatter, label='Cluster Label')
plt.show()
```



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 [13]:
```
## YOUR CODE GOES HERE
fig, axes = plt.subplots(1, 3, figsize=(18, 6))

kmeans = KMeans(n_clusters=4, random_state=0)
kmeans_labels = kmeans.fit_predict(x)
plot_pred(x, kmeans_labels, axes[0], "KMeans Clustering")

spectral = SpectralClustering(n_clusters=4, affinity='nearest_neighbors', random_state=0)
spectral_labels = spectral.fit_predict(x)
plot_pred(x, spectral_labels, axes[1], "Spectral Clustering")

gmm = GaussianMixture(n_components=4, random_state=0)
gmm_labels = gmm.fit_predict(x)
plot_pred(x, gmm_labels, axes[2], "Gaussian Mixture Model")

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

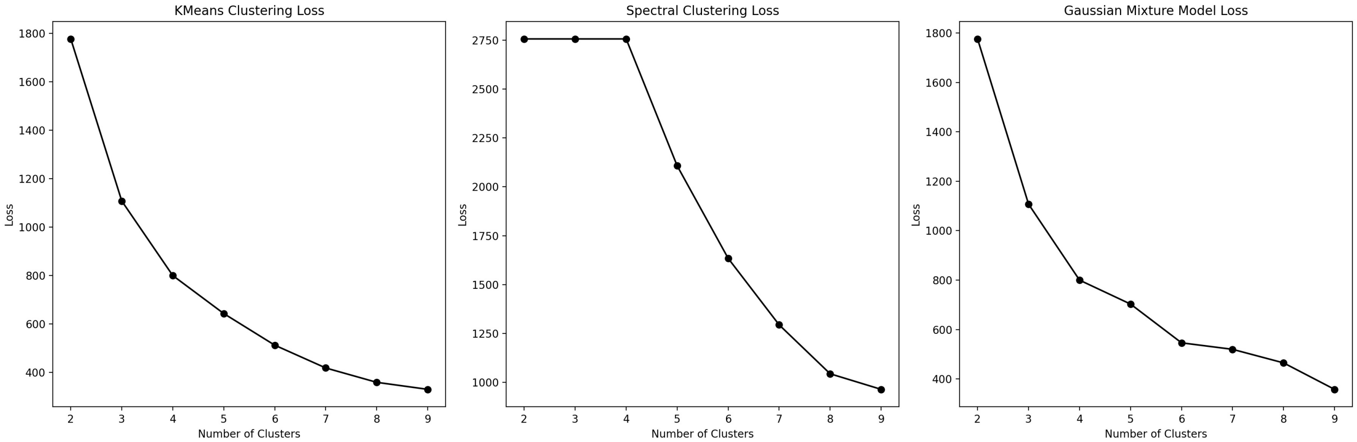
```
c:\Users\barat\AppData\Local\Programs\Python\Python312\Lib\site-packages\sklearn\manifold\_spectral_embedding.py
:329: UserWarning: Graph is not fully connected, spectral embedding may not work as expected.
  warnings.warn(
```



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.

In [14]:
```
## YOUR CODE GOES HERE
fig, axes = plt.subplots(1, 3, figsize=(18, 6))

kmeans_losses = []
spectral_losses = []
gmm_losses = []

for n_clust in range(2, 10):
    kmeans = KMeans(n_clusters=n_clust, random_state=0)
    kmeans_labels = kmeans.fit_predict(x)
    kmeans_loss = compute_loss(x, kmeans_labels)
    kmeans_losses.append(kmeans_loss)

    spectral = SpectralClustering(n_clusters=n_clust, affinity='nearest_neighbors', random_state=0)
    spectral_labels = spectral.fit_predict(x)
    spectral_loss = compute_loss(x, spectral_labels)
    spectral_losses.append(spectral_loss)

    gmm = GaussianMixture(n_components=n_clust, random_state=0)
    gmm_labels = gmm.fit_predict(x)
    gmm_loss = compute_loss(x, gmm_labels)
    gmm_losses.append(gmm_loss)

plot_loss(kmeans_losses, axes[0], "KMeans Clustering Loss")
plot_loss(spectral_losses, axes[1], "Spectral Clustering Loss")
plot_loss(gmm_losses, axes[2], "Gaussian Mixture Model Loss")

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

```
c:\Users\barat\AppData\Local\Programs\Python\Python312\Lib\site-packages\sklearn\manifold\_spectral_embedding.py
:329: UserWarning: Graph is not fully connected, spectral embedding may not work as expected.
  warnings.warn(
c:\Users\barat\AppData\Local\Programs\Python\Python312\Lib\site-packages\sklearn\manifold\_spectral_embedding.py
:329: UserWarning: Graph is not fully connected, spectral embedding may not work as expected.
  warnings.warn(
c:\Users\barat\AppData\Local\Programs\Python\Python312\Lib\site-packages\sklearn\manifold\_spectral_embedding.py
:329: UserWarning: Graph is not fully connected, spectral embedding may not work as expected.
  warnings.warn(
c:\Users\barat\AppData\Local\Programs\Python\Python312\Lib\site-packages\sklearn\manifold\_spectral_embedding.py
:329: UserWarning: Graph is not fully connected, spectral embedding may not work as expected.
  warnings.warn(
c:\Users\barat\AppData\Local\Programs\Python\Python312\Lib\site-packages\sklearn\manifold\_spectral_embedding.py
:329: UserWarning: Graph is not fully connected, spectral embedding may not work as expected.
  warnings.warn(
c:\Users\barat\AppData\Local\Programs\Python\Python312\Lib\site-packages\sklearn\manifold\_spectral_embedding.py
:329: UserWarning: Graph is not fully connected, spectral embedding may not work as expected.
  warnings.warn(
c:\Users\barat\AppData\Local\Programs\Python\Python312\Lib\site-packages\sklearn\manifold\_spectral_embedding.py
:329: UserWarning: Graph is not fully connected, spectral embedding may not work as expected.
  warnings.warn(
c:\Users\barat\AppData\Local\Programs\Python\Python312\Lib\site-packages\sklearn\manifold\_spectral_embedding.py
:329: 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?

KMeans works best for identifying the natural number of clusters with the elbow method on the "blob" dataset. The loss sharply drops and then levels off when the correct number of clusters is reached. The elbow method is less effective for Spectral Clustering and Gaussian Mixture Models in this case, as both algorithms may show less significant elbows due to the nature of their models. However, both still work well in clustering the data. The elbow method is highly effective for KMeans when clusters are spherical and well-separated, but for more complex datasets with irregular or non-convex clusters, the elbow method might be more ambiguous.

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?

In conclusion, Spectral Clustering is the most effective algorithm for clustering the concentric circles dataset, and the elbow method works better for it than for KMeans or GMM. For non-convex datasets like concentric circles, algorithms that do not rely on convexity assumptions (like Spectral Clustering) tend to perform better.

Since KMeans is not suited to handle non-convex shapes, the loss will decrease slowly without a sharp drop that would indicate the optimal number of clusters. The elbow is not clearly identifiable, and the method does not provide reliable information for determining the natural number of clusters in concentric circles.

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?

SSD works well for spherical, compact, and convex clusters, where the clusters are well-separated, and centroids can meaningfully represent the "center" of each cluster. For example, SSD works well for KMeans clustering on well-separated blob-like datasets. SSD fails on non-convex, irregular, or overlapping clusters, where the shape of the clusters deviates from a spherical or convex form.

KMeans: SSD works well for well-separated, convex, and spherical clusters, but it does not perform well on datasets with non-convex or complex shapes. The loss function does not account for non-linear relationships between points, making it unsuitable for datasets with

more intricate structures.

Spectral Clustering: SSD is not the appropriate measure for assessing Spectral Clustering performance. Since Spectral Clustering operates based on graph-based similarities and does not assume spherical clusters, SSD fails to measure the true quality of the clustering when applied to non-convex data.

GMM: Although GMM does not directly rely on SSD, using SSD as an evaluation metric for GMM is still problematic for non-ellipsoidal clusters. GMM works better for capturing Gaussian-like clusters, and while SSD might somewhat capture cluster fit in such cases, it does not provide a robust measure for irregularly shaped clusters.

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:

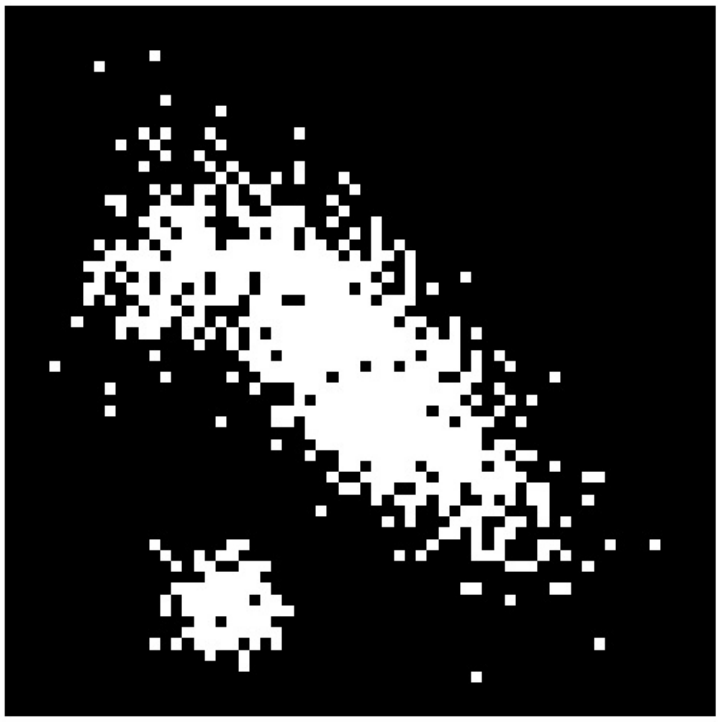
In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn.cluster import DBSCAN

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.

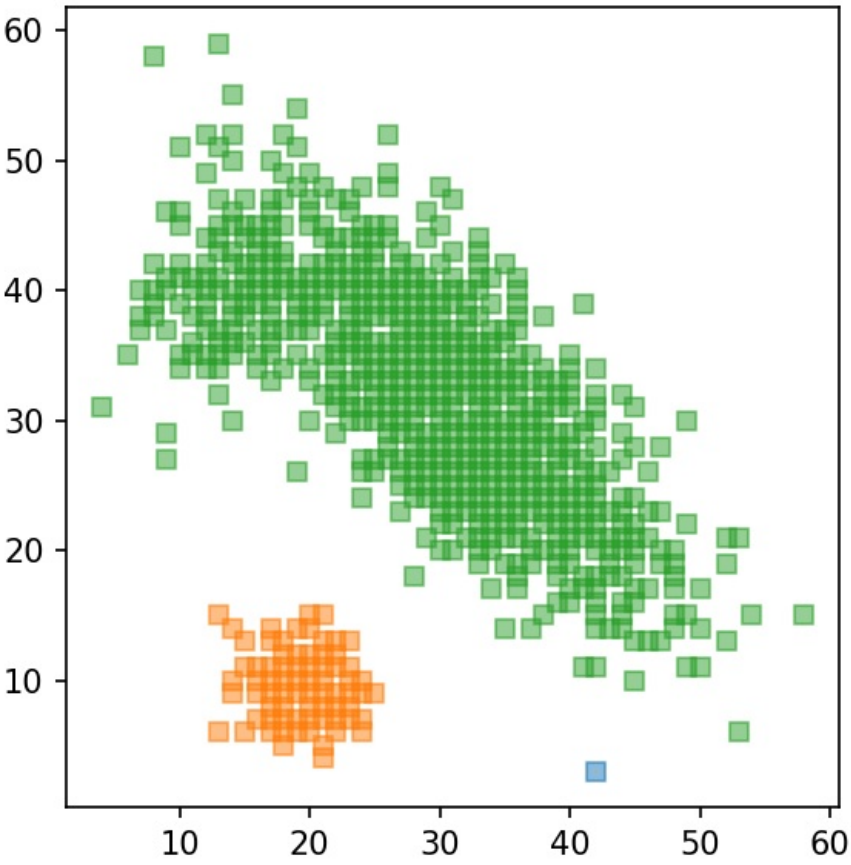
Note: you will use the integer coordinates for clustering, the bitmap is just for visualizing the data.

In [3]:
```
## YOUR CODE GOES HERE
data = np.loadtxt('data/m11-hw1-data1.txt', dtype=int)
bitmap = points_to_bitmap(data)

plot_bitmap(bitmap)
plt.show()
```



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 [46]:
```
## YOUR CODE GOES HERE
db = DBSCAN(eps=7, min_samples=10)
labels = db.fit_predict(data)

plot_points(data, labels)
plt.show()
```



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

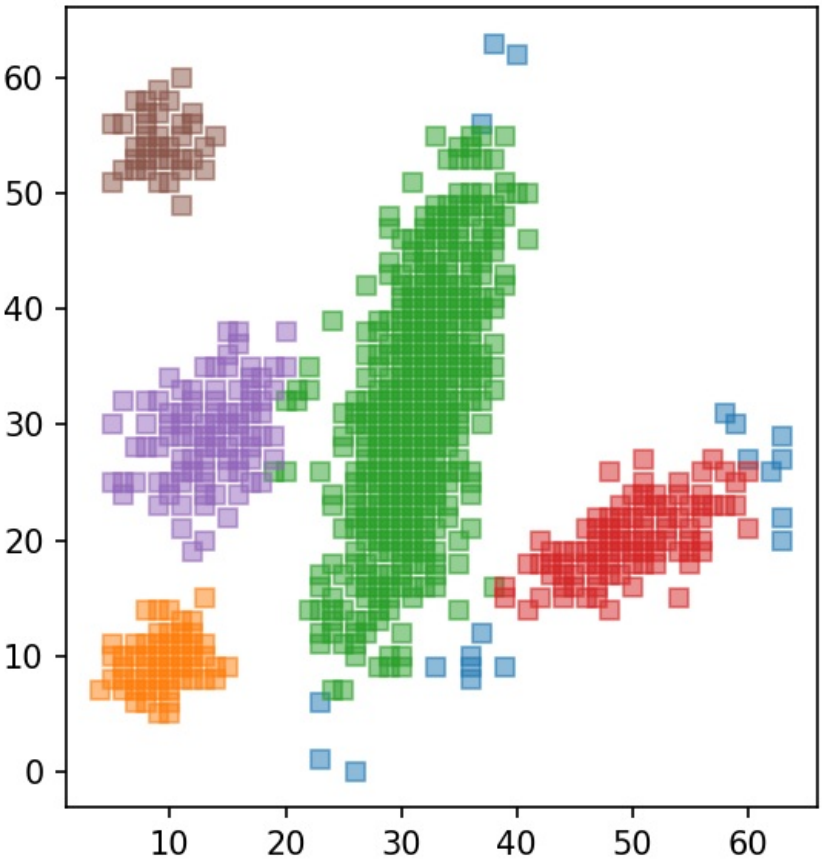
to ensure all values are loaded with their integer coordinates. Again, convert the points to a binary bitmap, and visualize the bitmap using the provided functions.

In [48]:
```
## YOUR CODE GOES HERE
data = np.loadtxt('data/m11-hw1-data2.txt', dtype=int)
bitmap = points_to_bitmap(data)

plot_bitmap(bitmap)
plt.show()
```

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 [80]:
```
## YOUR CODE GOES HERE
db = DBSCAN(eps=5.5, min_samples=35)
labels = db.fit_predict(data)

plot_points(data, labels)
plt.show()
```



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?

First Image (eps=5.5, min_samples=35)

The first image shows well-separated, distinct clusters that required more precise parameter tuning:

1. The smaller eps value of 5.5 helps maintain clear boundaries between the clusters
2. The higher min_samples value of 35 ensures that only dense regions form clusters
3. This configuration successfully identifies 6 distinct clusters with different densities and shapes

Second Image (eps=7, min_samples=10)

The second image presents a more challenging clustering scenario:

1. A larger eps value of 7 was needed to capture the broader connectivity in the elongated cluster
2. The lower min_samples threshold of 10 accommodates the varying density within the clusters
3. This configuration identifies 2 main clusters: one large elongated cluster and one smaller dense cluster

Clustering Effectiveness

The first image allows for more flexible parameter tuning because:

```
The clusters have clear spatial separation between them

Each cluster has relatively uniform density

The clusters are compact and roughly circular in shape
```

The second image is more sensitive to parameter selection because:

```
It contains an elongated, non-spherical cluster with varying density

The density gradient makes it harder to find parameters that work across the entire structure

There's less distinct separation between potential sub-clusters
```