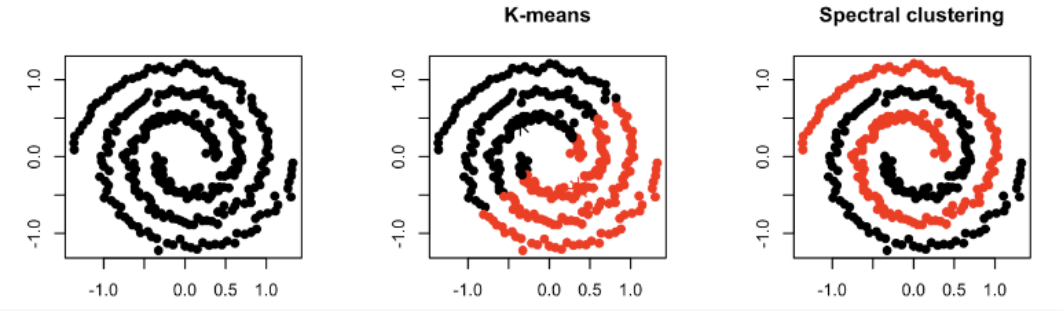
Machine Learning II
Workshop II
Hector Mauricio Rendón López

1. Spectral Clustering

Among the clustering algorithms are K means clustering, Fuzzy analysis clustering, Mean shift, DBSCAN and Spectral are clustering algorithms. The K-means algorithm generally assumes that clusters are spherical or round, that is, within a radius k from the centroid of the cluster. Furthermore, it requires many iterations to determine the centroid of the group. On the spectrum, groups do not follow a fixed shape or pattern. Points that are far away but connected belong to the same group and points that are less distant from each other could belong to different groups if they are not connected. This implies that the algorithm could be effective for data of different shapes and sizes.

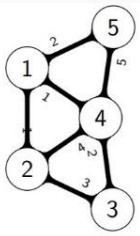


Spectral clustering is a technique originating from graph theory, the approach is used to identify communities of nodes in a graph based on the edges that connect them. The method is flexible and also allows data to be grouped without graphs. This technique is based on the eigenvalues of the similarity matrix of the data to perform a dimensionality reduction before grouping them into fewer dimensions. It consists of constructing a similarity graph, projecting data into a lower-dimensional space, and then clustering the data.

Steps:

1. Form a distance matrix
2. Transform the distance matrix into an affinity matrix A (adjacency matrix)
3. Calculate the degree matrix D and the Laplacian matrix L = D – A.
4. Find the eigenvalues and eigenvectors of L.
5. Form a matrix with the eigenvectors of k largest eigenvalues calculated in the previous step.
6. Normalize the vectors.
7. Group data points in k-dimensional space

$$L = \begin{pmatrix} 4 & -1 & 0 & -1 & -2 \\ -1 & 8 & -3 & -4 & 0 \\ 0 & -3 & 5 & -2 & 0 \\ -1 & -4 & -2 & 12 & -5 \\ -2 & 0 & 0 & -5 & 7 \end{pmatrix}$$



a. Application

- To identify patterns and similarities that across different observations.
- Image segmentation
- Educational data mining
- Entity resolution
- Speech separation
- Spectral clustering of protein sequences
- Text image segmentation.

b. Mathematical fundamentals

Given an enumerated set of data points, the similarity matrix may be defined as a symmetric matrix A, where $A_{ij} >= 0$ represents a measure of the similarity between data points with indices *i* and *j.*

The general approach to spectral clustering is to use a standard clustering method (k-means) on relevant eigenvectors of a Laplacian matrix of A. The eigenvectors that are relevant are the ones that correspond to smallest several eigenvalues

of the Laplacian except for the smallest eigenvalue which will have a value of 0. For computational efficiency, these eigenvectors are often computed as the eigenvectors corresponding to the largest several eigenvalues of a function of the Laplacian. The eigenvalue problem describing transversal vibration modes of a mass-spring system is exactly the same as the eigenvalue problem for the graph Laplacian matrix defined as

$$L: D - A$$

Where D is the diagonal matrix,

$$D_{ii} = \sum_j A_{ij},$$

And A is the adjacency matrix.

So that the components of the eigenvectors corresponding to the smallest eigenvalues of the graph Laplacian.

c. Algorithm to compute it

1. Calculate the Laplacian ***L*** (or the normalized Laplacian)
2. Calculate the first ***k*** eigenvectors (the eigenvectors corresponding to the ***k*** smallest eigenvalues of ***L***)
3. Consider the matrix formed by the first ***k*** eigenvectors; the ***l-th*** row defines the features of graph node ***l***
4. Cluster the graph nodes based on these features (e.g., using k-means clustering).

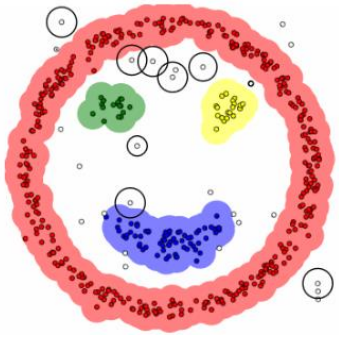
```
import numpy as np
from sklearn.cluster import SpectralClustering
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt

# Generate some sample data
n_samples = 300
n_clusters = 3
X, y = make_blobs(n_samples=n_samples, centers=n_clusters, random_state=42)
# Create a similarity matrix (e.g., using radial basis function kernel)
def rbf_kernel(X, gamma=1.0):
    n = X.shape[0]
    similarity_matrix = np.zeros((n, n))
    for i in range(n):
        for j in range(i, n):
            similarity = np.exp(-gamma * np.linalg.norm(X[i] - X[j])**2)
            similarity_matrix[i, j] = similarity
            similarity_matrix[j, i] = similarity
    return similarity_matrix
gamma = 1.0
similarity_matrix = rbf_kernel(X, gamma)
# Perform spectral clustering
n_clusters = 3  # Number of clusters
sc = SpectralClustering(n_clusters=n_clusters, affinity='precomputed')
labels = sc.fit_predict(similarity_matrix)
# Plot the results
plt.scatter(X[:, 0], X[:, 1], c=labels, cmap=plt.cm.get_cmap("viridis", n_clusters))
plt.title("Spectral Clustering")
plt.show()
```

d. Relation to some of the concepts viwes

Clustering is one of the initial steps done in exploratory data analysis to visualize the similarity and to identify the pattern lying hidden in data points. The motive of clustering is to find the similarity within a cluster and the difference between two clusters. K mean clustering, Fuzzy analysis clustering, Mean shift, DBSCAN and Spectral are clustering algorithms.

2. DBSCAN (Density-Based Clustering Algorithms)

Density-Based Clustering refers to unsupervised learning methods that identify distinctive groups/clusters in the data, based on the idea that a cluster in data space is a contiguous region of high point density, separated from other such clusters by contiguous regions of low point density. Density-Based Spatial Clustering of Applications with Noise (DBSCAN) is a base algorithm for density-based clustering. It can discover clusters of different shapes and sizes from a large amount of data, which is containing noise and outliers.



a. Application

- To detect clusters that are oddly or irregularly shaped, such as clusters that are ring-shaped.
- It is able to detect outliers and exclude them from the clusters entirely. That means that DBSCAN is very robust to outliers and great for datasets with multiple outliers.
- It can automatically detect the number of clusters that exist in the data. This is great for cases where you do not have much intuition on how many clusters there should be.
- Is less sensitive to initialization conditions like the order of the observations in the dataset and the seed that is used.
- There are multiple implementations of DBSCAN that aim to optimize the time complexity of the algorithm. DBSCAN is generally slower than k-means clustering but faster than hierarchical clustering and spectral clustering.

b. Mathematical fundamentals

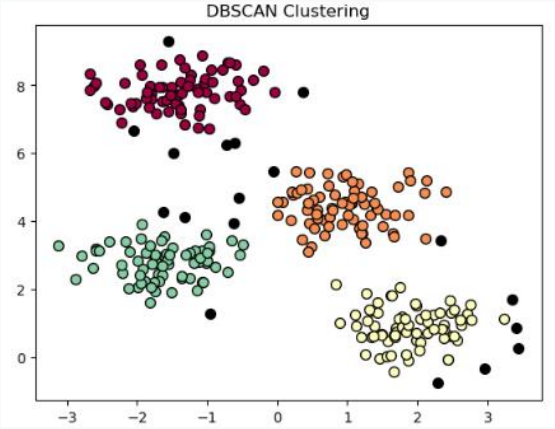
Consider a set of points in some space to be clustered. Let ε be a parameter specifying the radius of a neighborhood with respect to some point. For the purpose of DBSCAN clustering, the points are classified as core points, (directly-) reachable points and outliers, as follows:

- A point p is a core point if at least minPts points are within distance ε of it (including p).
- A point q is directly reachable from p if point q is within distance ε from core point p. Points are only said to be directly reachable from core points.
- A point q is reachable from p if there is a path p1, ..., pn with p1 = p and pn = q, where each pi+1 is directly reachable from pi. Note that this implies that the initial point and all points on the path must be core points, with the possible exception of q.
- All points not reachable from any other point are outliers or noise points.

Now if p is a core point, then it forms a cluster together with all points (core or non-core) that are reachable from it. Each cluster contains at least one core point; non-core points can be part of a cluster, but they form its "edge", since they cannot be used to reach more points.

c. Algorithm to compute it - Using Scikit-learn

```
from sklearn.cluster import DBSCAN
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt

# Generate synthetic data
X, _ = make_blobs(n_samples=300, centers=4, cluster_std=0.60, random_state=0)

# Create DBSCAN instance
dbscan = DBSCAN(eps=0.5, min_samples=5)

# Fit the DBSCAN model to the data
dbscan.fit(X)

# Get cluster labels (-1 represents noise points)
labels = dbscan.labels_
# Create a scatter plot of the data points with color-coded clusters
unique_labels = set(labels)
colors = [plt.cm.Spectral(each) for each in np.linspace(0, 1, len(unique_labels))]

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

    class_member_mask = (labels == k)

    xy = X[class_member_mask]
    plt.scatter(xy[:, 0], xy[:, 1], c=[col], marker='o', s=50, edgecolor='k')

plt.title('DBSCAN Clustering')
plt.show()
```



e. Relation to some of the concepts viwes

Like other clustering techniques, it allows exploratory data analysis to visualize similarity and identify patterns. in this case DBSCAN is a technique for data set with irregularly shaped clusters, data with outliers and anomaly detection.

3. The elbow method in clustering

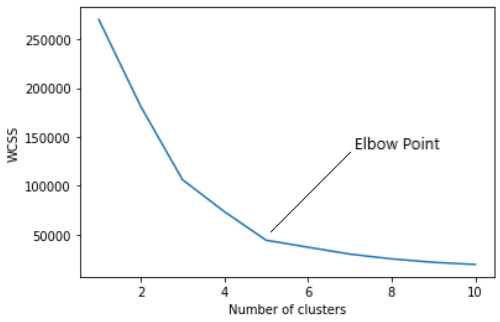
The elbow method is a popular unsupervised learning algorithm used in K-Means clustering to determine the optimal number of clusters for a given dataset. In the Elbow method, we are actually varying the number of clusters (K) from 1 – 10. For each value of K, we are calculating WCSS (Within-Cluster Sum of Square). WCSS is the sum of the squared distance between each point and the centroid in a cluster. When we plot the WCSS with the K value, the plot looks like an Elbow. As the number of clusters increases, the WCSS value will start to decrease. WCSS value is largest when K = 1. When we analyze the graph, we can see that the graph will rapidly change at a point and thus creating an elbow shape. From this

point, the graph moves almost parallel to the X-axis. The K value corresponding to this point is the optimal value of K or an optimal number of clusters.

The intuition behind the elbow method is that you want to find a balance between having too few clusters (underfitting) and having too many clusters (overfitting). The "elbow point" represents a good compromise where adding more clusters does not significantly reduce the WCSS, suggesting that it's a suitable number of clusters for your data.

Flaws does it pose to assess quality:

- Can be subjective determining the exact elbow point, involves visual judgment which can be imprecise.
- Different initializations of cluster center can lead to different results, impacting the shape elbow.
- Assumes that clusters are globular and equally sized.
- Is primarily designed for K-means clustering, It may not be applicable or suitable for other clustering algorithms with different cost functions or cluster structures.
- It's a heuristic approach and may not always lead to the most meaningful or interpretable clustering results.

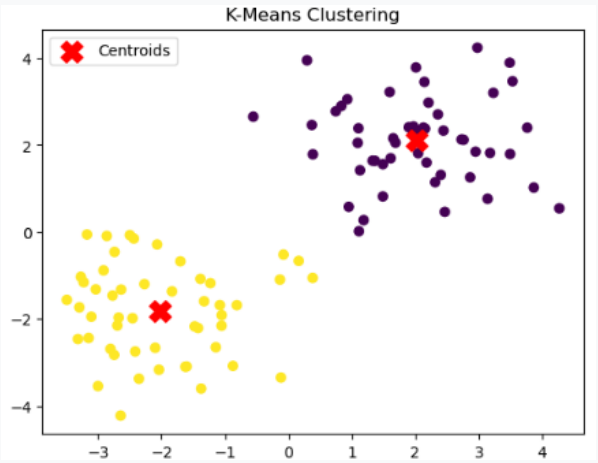


4. Python package

a. k-means module using Python and Numpy

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

class KMeans:
    def __init__(self, n_clusters=2, max_iters=500, random_state=None):
        self.n_clusters = n_clusters
        self.max_iters = max_iters

    def fit(self, X):
        # Initialize cluster centroids randomly
        self.centroids = X[np.random.choice(X.shape[0], self.n_clusters, replace=False)]

        for _ in range(self.max_iters):
            # Assign each point to the nearest cluster
            labels = self._assign_clusters(X)

            # Update cluster centroids
            new_centroids = self._update_centroids(X, labels)

            # Check for convergence
            if np.all(self.centroids == new_centroids):
                break

            self.centroids = new_centroids

        self.labels = self._assign_clusters(X)

    def _assign_clusters(self, X):
        distances = np.linalg.norm(X[:, np.newaxis] - self.centroids, axis=2)
        return np.argmin(distances, axis=1)

    def _update_centroids(self, X, labels):
        new_centroids = np.array([X[labels == i].mean(axis=0) for i in range(self.n_clusters)])
        return new_centroids
```

```python
def main():
    # Generate some random data points for demonstration
    np.random.seed(0)
    data = np.concatenate([np.random.randn(50, 2) + [2, 2], np.random.randn(50, 2) + [-2, -2]])

    # Initialize and fit the K-means model
    kmeans = KMeans(n_clusters=2)
    kmeans.fit(data)

    # Visualize the data points and cluster centroids
    plt.scatter(data[:, 0], data[:, 1], c=kmeans.labels, cmap='viridis')
    plt.scatter(kmeans.centroids[:, 0], kmeans.centroids[:, 1], marker='X', s=200, c='red', label='Centroids')
    plt.legend()
    plt.title('K-Means Clustering')
    plt.show()

if __name__ == "__main__":
    main()
```

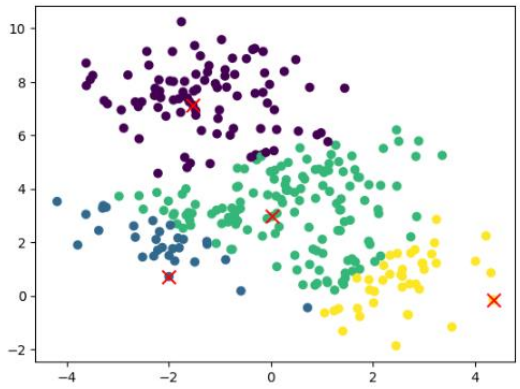


b. k-mediods module using Python and Numpy

```python
import numpy as np
from sklearn.base import BaseEstimator, ClusterMixin
from sklearn.metrics import pairwise_distances_argmin_min


class KMedoids(BaseEstimator, ClusterMixin):
    def __init__(self, n_clusters=8, max_iter=300, random_state=None):
        self.n_clusters = n_clusters
        self.max_iter = max_iter
        self.random_state = random_state

    def fit(self, X):
        np.random.seed(self.random_state)

        # Initialize medoids randomly
        n_samples, n_features = X.shape
        medoids_idx = np.random.choice(n_samples, size=self.n_clusters, replace=False)
        medoids = X[medoids_idx]

        for _ in range(self.max_iter):
            # Assign each point to the nearest medoid
            labels, distances = pairwise_distances_argmin_min(X, medoids)
```

```
            # Compute the new medoids as the point with the minimum total distance
            new_medoids = np.empty_like(medoids)
            for i in range(self.n_clusters):
                cluster_points = X[labels == i]
                total_distances = np.sum(pairwise_distances_argmin_min(cluster_points, cluster_points)[1])
                best_medoid_idx = np.argmin(total_distances)
                new_medoids[i] = cluster_points[best_medoid_idx]

            # Check for convergence
            if np.all(new_medoids == medoids):
                break

            medoids = new_medoids

        self.medoids_ = medoids
        self.labels_ = labels

    def fit_predict(self, X):
        self.fit(X)
        return self.labels_
```

```
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt
# Create synthetic data
X, _ = make_blobs(n_samples=300, centers=4, random_state=0, cluster_std=1.0)
# Fit the KMedoids model
kmedoids = KMedoids(n_clusters=4, random_state=0)
labels = kmedoids.fit_predict(X)
# Plot the results
plt.scatter(X[:, 0], X[:, 1], c=labels)
plt.scatter(kmedoids.medoids_[:, 0], kmedoids.medoids_[:, 1], c='red', marker='x', s=100)
plt.show()
```



5. Modules in unsupervised to cluster some toy data.

a. Code snippet to create scattered data X

```
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans

# Create scattered data
X, y = make_blobs(
    n_samples=500,
    n_features=2,
    centers=4,
    cluster_std=1,
    center_box=(-10.0, 10.0),
    shuffle=True,
    random_state=1,
)
```

b. Plot the resulting data set.

```
# Use K-means clustering to identify clusters
kmeans = KMeans(n_clusters=4, random_state=1)
kmeans.fit(X)

# Get cluster centers and labels
cluster_centers = kmeans.cluster_centers_
labels = kmeans.labels_

# Plot the scattered data with cluster centers
plt.scatter(X[:, 0], X[:, 1], c=labels, marker='o', s=25, cmap='viridis')
plt.scatter(cluster_centers[:, 0], cluster_centers[:, 1], c='red', marker='x', s=100, label='Cluster Centers')
plt.title("Clustered Data")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.legend()
plt.show()

# Number of clusters
num_clusters = len(cluster_centers)
print(f"Number of clusters: {num_clusters}")
```

There are 4 conglomerates
Distance from each other:

```
from scipy.spatial.distance import euclidean

# Calculate pairwise distances within each cluster
cluster_distances = []
for cluster_id in range(4):
    cluster_points = X[y == cluster_id]
    pairwise_distances = []
    for i in range(len(cluster_points)):
        for j in range(i + 1, len(cluster_points)):
            distance = euclidean(cluster_points[i], cluster_points[j])
            pairwise_distances.append(distance)
    cluster_distances.append(pairwise_distances)

# Calculate the average distance within each cluster
average_distances = [np.mean(distances) for distances in cluster_distances]

print("Average distance within each cluster:")
for i, avg_distance in enumerate(average_distances):
    print(f"Cluster {i}: {avg_distance}")
```

```
Average distance within each cluster:
Cluster 0: 1.7106234583137094
Cluster 1: 1.7886004620399283
Cluster 2: 1.7972021752346738
Cluster 3: 1.6412785317170073
```

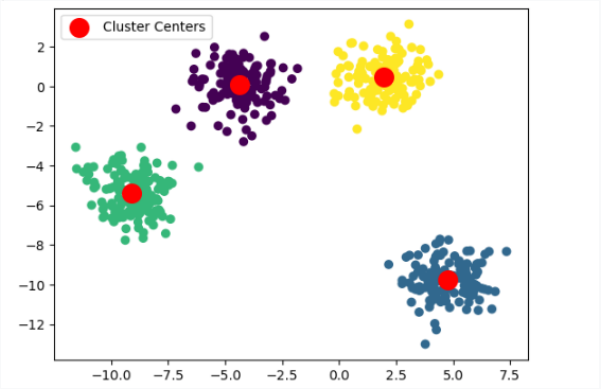
The best number of clusters (K) is 3 with a silhouette score of 0.65

6. Snippet to create different types of scattered data

```
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt
from sklearn import cluster, datasets, mixture

n_samples = 500
noisy_circles = datasets.make_circles(n_samples=n_samples, factor=0.5, noise=0.05)
noisy_moons = datasets.make_moons(n_samples=n_samples, noise=0.05)
blobs = datasets.make_blobs(n_samples=n_samples, random_state=8)
no_structure = np.random.rand(n_samples, 2), None
random_state=170
transformation = [[0.6, -0.6], [-0.4, 0.8]]
X_aniso = np.dot(X, transformation)
aniso = (X_aniso, y)

# Generate  data
X,y = make_blobs(n_samples=n_samples, centers=4, n_features=2,random_state=random_state)

varied = datasets.make_blobs(
n_samples=n_samples, cluster_std=[1.0, 2.5, 0.5], random_state=random_state
)
```

```
 # Plot the data points and cluster centers
 plt.scatter(X[:, 0], X[:, 1], c=labels)
 plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], s=200, c='red', label='Cluster Centers')
 plt.legend()
 plt.show()
```

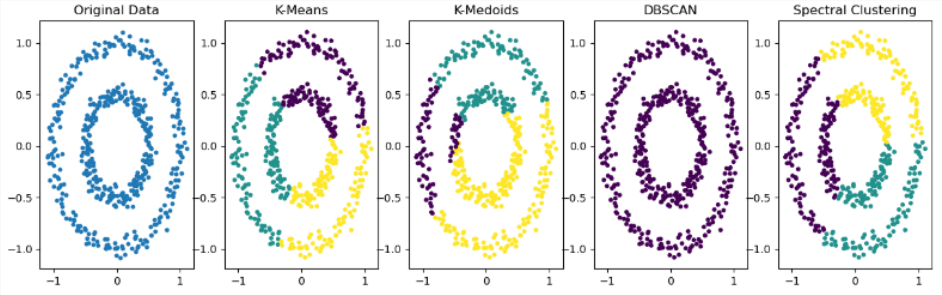
}

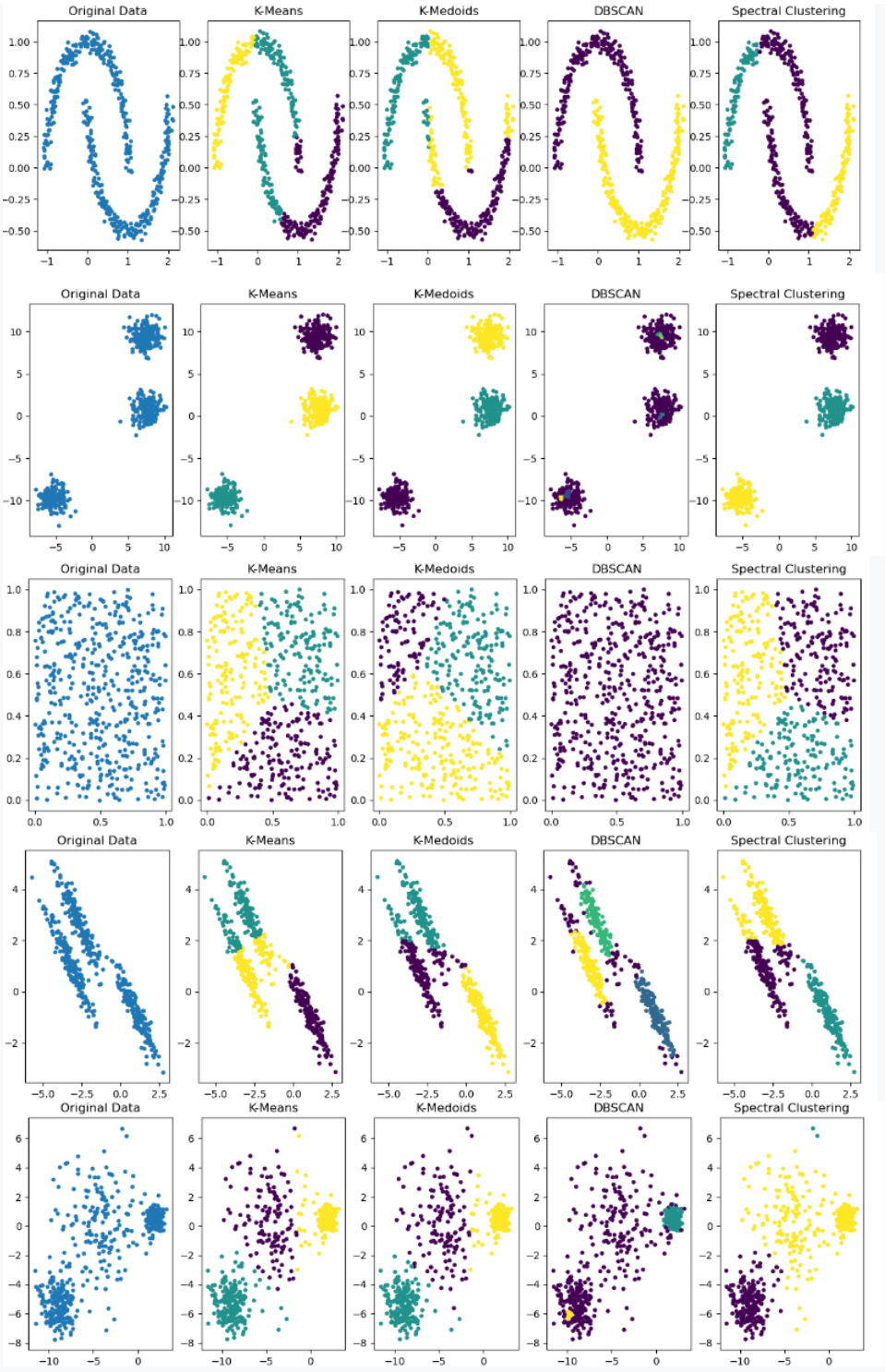


Apply k-means, k-medoids, DBSCAN and Spectral Clustering from Scikit-Learn over each dataset and compare the results of each algorithm with respect to each dataset:

```
from sklearn.cluster import KMeans, DBSCAN, SpectralClustering
import numpy as np
from sklearn.base import BaseEstimator, ClusterMixin
from sklearn.metrics import pairwise_distances_argmin_min

class KMedoids(BaseEstimator, ClusterMixin):
    def __init__(self, n_clusters=8, max_iter=300, random_state=None):
        self.n_clusters = n_clusters
        self.max_iter = max_iter
        self.random_state = random_state

    def fit(self, X):
        np.random.seed(self.random_state)

        # Initialize medoids randomly
        n_samples, n_features = X.shape
        medoids_idx = np.random.choice(n_samples, size=self.n_clusters, replace=False)
        medoids = X[medoids_idx]
```

```
        for _ in range(self.max_iter):
            # Assign each point to the nearest medoid
            labels, distances = pairwise_distances_argmin_min(X, medoids)

            # Compute the new medoids as the point with the minimum total distance
            new_medoids = np.empty_like(medoids)
            for i in range(self.n_clusters):
                cluster_points = X[labels == i]
                total_distances = np.sum(pairwise_distances_argmin_min(cluster_points, cluster_points)[1])
                best_medoid_idx = np.argmin(total_distances)
                new_medoids[i] = cluster_points[best_medoid_idx]

            # Check for convergence
            if np.all(new_medoids == medoids):
                break

            medoids = new_medoids

        self.medoids_ = medoids
        self.labels_ = labels

    def fit_predict(self, X):
        self.fit(X)
        return self.labels_

# Create a dictionary to store the clustering results for each dataset and algorithm
results = {}

# Define clustering algorithms
algorithms = {
    "K-Means": KMeans(n_clusters=3, random_state=0),
    "K-Medoids": KMedoids(n_clusters=3, random_state=0),
    "DBSCAN": DBSCAN(eps=0.3, min_samples=10),
    "Spectral Clustering": SpectralClustering(n_clusters=3, random_state=0),
}

# Loop through the datasets and apply clustering algorithms
for dataset_name, (X, _) in datasets_list:
    plt.figure(figsize=(12, 4))
    plt.subplots_adjust(left=0.02, right=0.98, bottom=0.1, top=0.9)

    # Plot the original data
    plt.subplot(1, 5, 1)
    plt.scatter(X[:, 0], X[:, 1], s=10)
    plt.title("Original Data")

    results[dataset_name] = {}

    for algorithm_name, algorithm in algorithms.items():
        algorithm.fit(X)
        labels = algorithm.labels_

        # Plot the clustering results
        plt.subplot(1, 5, len(results[dataset_name]) + 2)
        plt.scatter(X[:, 0], X[:, 1], c=labels, s=10)
        plt.title(algorithm_name)
        results[dataset_name][algorithm_name] = labels

plt.show()
```





This code will generate separate figures for each dataset and plot the original data as well as the results of the four clustering algorithms (K-Means, K-Medoids, DBSCAN, and Spectral Clustering) for each dataset. It can visually compare the performance of each algorithm on different types of scattered data.

- Noisy Circles: This dataset consists of two intertwined circles with some noise. It's a challenging dataset for clustering algorithms because the clusters are not linearly separable.
- Noisy Moons: Similar to the noisy circles dataset, this one contains two moon-shaped clusters with noise. It's also a non-linearly separable dataset.

- Blobs: This dataset contains three well-defined clusters of data points. It's a relatively straightforward dataset for clustering algorithms as the clusters are spherical and well-separated.
- No Structure: This dataset appears as random scattered points with no discernible structure or clusters. It's essentially noise with no underlying patterns.
- Anisotropic Data: The data in this dataset is distributed in an anisotropic manner, meaning the clusters are stretched along specific directions. It's another challenging dataset for clustering algorithms due to its elongated cluster shapes.
- Varied Variances: This dataset contains three clusters, each with a different standard deviation, making it challenging for clustering algorithms to identify the clusters with different variances.