LAB MANUAL [DENSITY-BASED CLUSTERING]

DBSCAN [Density-Based Spatial Clustering of Applications with Noise]

DBSCAN, which stands for Density-Based Spatial Clustering of Applications with Noise, is a powerful clustering algorithm that groups points that are closely packed together in data space. Unlike some other clustering algorithms, DBSCAN doesn't require you to specify the number of clusters beforehand, making it particularly useful for exploratory data analysis.

The algorithm works by defining clusters as dense regions separated by regions of lower density. This approach allows DBSCAN to discover clusters of arbitrary shape and identify outliers as noise.

DBSCAN revolves around three key concepts:

1. Core Points: These are points that have at least a minimum number of other points (MinPts) within a specified distance (ε or epsilon).
2. Border Points: These are points that are within the ε distance of a core point but don't have MinPts neighbors themselves.
3. Noise Points: These are points that are neither core points nor border points. They're not close enough to any cluster to be included.

DBSCAN uses two main parameters:

ε (epsilon): The maximum distance between two points for them to be considered as neighbors.
MinPts: The minimum number of points required to form a dense region.

Implementing DBSCAN in Python

In this section, we'll look at the implementation of DBSCAN using Python and the scikit-learn library.

Setting up the environment

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_moons
from sklearn.cluster import DBSCAN
from sklearn.neighbors import NearestNeighbors
```

These imports provide the necessary tools for data manipulation, visualization, dataset creation, and implementing the DBSCAN algorithm.

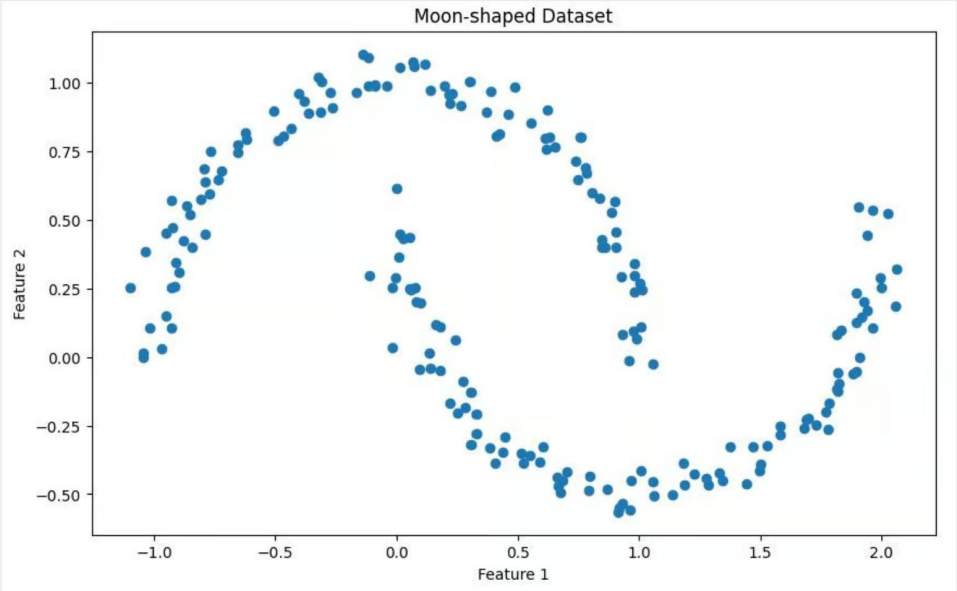
Generating sample data

```
X, _ = make_moons(n_samples=200, noise=0.05, random_state=42)
```

Let's visualize this dataset to better understand its structure:

```
# Visualize the dataset
plt.figure(figsize=(10, 6))
plt.scatter(X[:, 0], X[:, 1])
plt.title('Moon-shaped Dataset')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.show()
```

This will show you the two interleaving half-moon shapes in our dataset as shown below

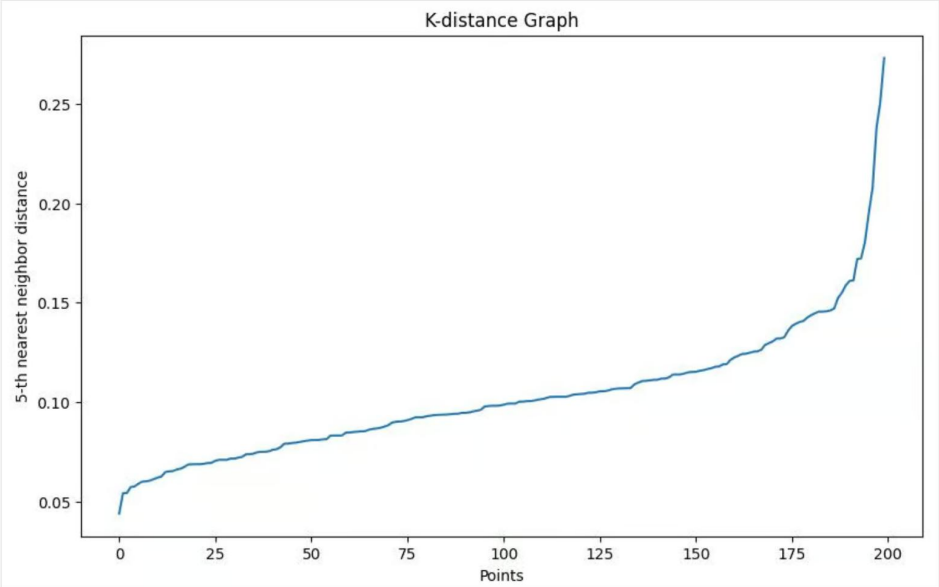


Determining the epsilon parameter
We use the k-distance graph method to help choose an appropriate epsilon value:

We define a function plot_k_distance_graph that calculates the distance to the k-th nearest neighbor for each point.
The distances are sorted and plotted.
We look for an "elbow" in the resulting graph to choose epsilon.

```
# Function to plot k-distance graph
def plot_k_distance_graph(X, k):
    neigh = NearestNeighbors(n_neighbors=k)
    neigh.fit(X)
    distances, _ = neigh.kneighbors(X)
    distances = np.sort(distances[:, k-1])
    plt.figure(figsize=(10, 6))
    plt.plot(distances)
    plt.xlabel('Points')
    plt.ylabel(f'{k}-th nearest neighbor distance')
    plt.title('K-distance Graph')
    plt.show()
# Plot k-distance graph
plot_k_distance_graph(X, k=5)
```

Output



In our example, based on the k-distance graph, we choose an epsilon of 0.15.

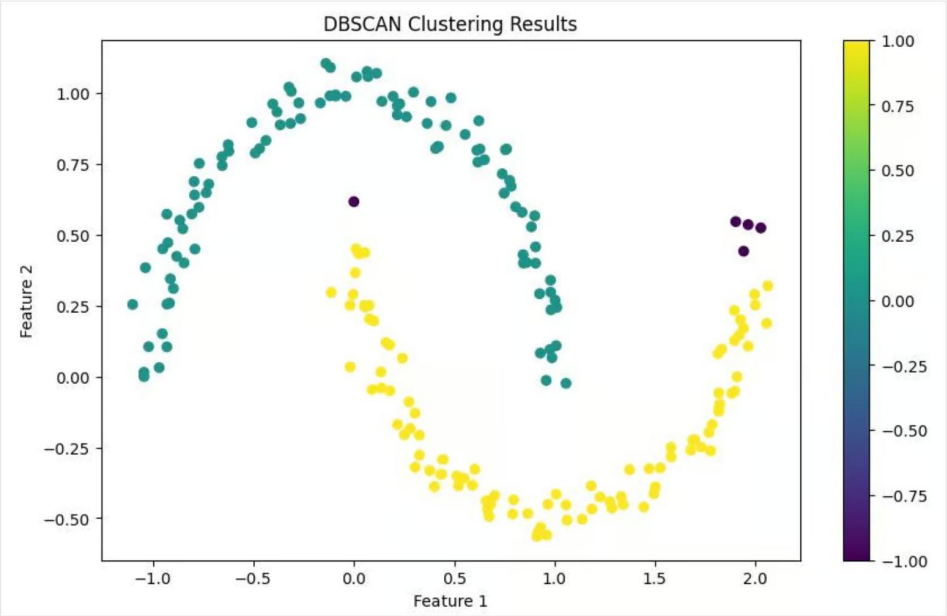
Performing DBSCAN clustering
We use scikit-learn's DBSCAN implementation:

We set epsilon=0.15 based on our k-distance graph.
We set min_samples=5 (2 * num_features, as our data is 2D).
We fit the model to our data and predict the clusters.

```
# Perform DBSCAN clustering
epsilon = 0.15  # Chosen based on k-distance graph
min_samples = 5  # 2 * num_features (2D data)
dbscan = DBSCAN(eps=epsilon, min_samples=min_samples)
clusters = dbscan.fit_predict(X)
```

Visualizing the results
We create a scatter plot of our data points, coloring them according to their assigned clusters. Points classified as noise are typically colored differently (often black).

Output



Interpreting the results

Finally, we print out the number of clusters found and the number of points classified as noise. This gives us a quick summary of the clustering results.

```
# Print number of clusters and noise points
n_clusters = len(set(clusters)) - (1 if -1 in clusters else 0)
n_noise = list(clusters).count(-1)
print(f'Number of clusters: {n_clusters}')
print(f'Number of noise points: {n_noise}')
```

Output:

Number of clusters: 2
Number of noise points: 5

Visual comparison

To illustrate these differences, let's apply both algorithms to our moon-shaped dataset

```
from sklearn.cluster import KMeans

# DBSCAN clustering
dbscan = DBSCAN(eps=0.15, min_samples=5)
dbscan_labels = dbscan.fit_predict(X)

# K-Means clustering
kmeans = KMeans(n_clusters=2, random_state=42)
kmeans_labels = kmeans.fit_predict(X)

# Visualize the results
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 6))

ax1.scatter(X[:, 0], X[:, 1], c=dbscan_labels, cmap='viridis')
ax1.set_title('DBSCAN Clustering')

ax2.scatter(X[:, 0], X[:, 1], c=kmeans_labels, cmap='viridis')
ax2.set_title('K-Means Clustering')

plt.show()
```

Output