DBSCAN in Unsupervised Learning

Purpose

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is an unsupervised machine learning algorithm used for clustering tasks. Its purpose is to find clusters in data of arbitrary shape and to identify noise (outliers).

When to Use DBSCAN

DBSCAN is particularly useful when:

1. **Data has clusters of varying shapes**: Unlike K-means, which assumes spherical clusters, DBSCAN can find clusters of any shape.
2. **Handling noise**: DBSCAN can identify and ignore noise points, making it robust to outliers.
3. **Number of clusters is unknown**: DBSCAN does not require specifying the number of clusters in advance.

Suitable Data Types

DBSCAN is suitable for:

1. **Geospatial data**: Finding regions of high density, such as hotspots in geographic data.
2. **Image analysis**: Identifying clusters of pixels with similar intensities.
3. **Anomaly detection**: Identifying outliers in datasets, such as fraudulent transactions in financial data.

Example: Implementing DBSCAN on a Real-Life Dataset

Let's implement DBSCAN using a real-life dataset, the `Mall_Customers` dataset, which contains information about customers of a mall including their annual income and spending score.

Steps

1. **Load the dataset**.
2. **Preprocess the data**: Select relevant features and standardize them.
3. **Apply DBSCAN**: Use the DBSCAN algorithm to cluster the data.
4. **Visualize the results**.

Implementation

```
import pandas as pd

import numpy as np
```

```
import matplotlib.pyplot as plt

from sklearn.preprocessing import StandardScaler

from sklearn.cluster import DBSCAN


# Step 1: Load dataset

url = 'https://raw.githubusercontent.com/Machine-Learning-Tokyo/MLT-exploration/master/Clustering/Mall_Customers.csv'

df = pd.read_csv(url)


# Step 2: Preprocess the data

X = df[['Annual Income (k$)', 'Spending Score (1-100)']].values

scaler = StandardScaler()

X_scaled = scaler.fit_transform(X)


# Step 3: Apply DBSCAN

dbscan = DBSCAN(eps=0.5, min_samples=5)

clusters = dbscan.fit_predict(X_scaled)


# Adding cluster labels to the original dataframe

df['Cluster'] = clusters


# Step 4: Visualize the results

plt.figure(figsize=(10, 6))

plt.scatter(df['Annual Income (k$)'], df['Spending Score (1-100)'], c=df['Cluster'], cmap='viridis', marker='o')

plt.title('DBSCAN Clustering of Mall Customers')

plt.xlabel('Annual Income (k$)')

plt.ylabel('Spending Score (1-100)')

plt.colorbar(label='Cluster')
```

```
plt.show()
```

Explanation

- **Loading the Dataset**: The dataset is loaded from a CSV file into a pandas DataFrame.
- **Preprocessing**: The features `Annual Income (k$)` and `Spending Score (1-100)` are selected for clustering. These features are then standardized using `StandardScaler` to ensure they have a mean of 0 and a standard deviation of 1.
- **DBSCAN Application**: The DBSCAN algorithm is applied to the standardized features. The `eps` parameter is set to 0.5, which is the maximum distance between two samples for them to be considered as in the same neighborhood. The `min_samples` parameter is set to 5, which is the minimum number of samples in a neighborhood for a point to be considered a core point.
- **Visualization**: The resulting clusters are visualized using a scatter plot. Each point is colored according to its cluster label.

This example demonstrates how DBSCAN can be used to cluster real-life customer data based on their spending patterns and income, providing insights into customer segmentation.

Example2

1. Load the Dataset

First, we will load the dataset.

```
import pandas as pd


# Load dataset
url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/00292/Wholesale%20customers%20data.csv'
df = pd.read_csv(url)


# Display first few rows of the dataset
df.head()
```

2. Visualize the Data Before Clustering

We'll visualize the data to understand its structure before applying clustering.

```
import matplotlib.pyplot as plt
import seaborn as sns

# Visualize the data
sns.pairplot(df, diag_kind='kde')
plt.suptitle('Pairplot of Wholesale Customers Data', y=1.02)
plt.show()
```

3. Preprocess the Data

We'll select relevant features for clustering and standardize them.

```
from sklearn.preprocessing import StandardScaler

# Selecting relevant features for clustering
X = df[['Grocery', 'Milk']].values  # Choosing only two features for easy visualization

# Standardizing the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

4. Apply DBSCAN

We'll apply the DBSCAN algorithm with appropriate parameters.

```
from sklearn.cluster import DBSCAN

# Applying DBSCAN
```

```
dbscan = DBSCAN(eps=0.5, min_samples=5)
clusters = dbscan.fit_predict(X_scaled)

# Adding cluster labels to the original dataframe
df['Cluster'] = clusters
```

5. Visualize the Data After Clustering

We'll visualize the data after clustering, highlighting the clusters and labeling the results.

```
# Plotting the clusters
plt.figure(figsize=(10, 6))
plt.scatter(df['Grocery'], df['Milk'], c=df['Cluster'], cmap='viridis', marker='o', edgecolor='k')
plt.title('DBSCAN Clustering of Wholesale Customers')
plt.xlabel('Annual Spending on Grocery')
plt.ylabel('Annual Spending on Milk')
plt.colorbar(label='Cluster')
plt.show()

# Display the dataframe with cluster labels
df.head()
```

Alo Show Legend for Better Understanding

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import DBSCAN
import numpy as np

# Step 1: Load dataset
url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/00292/Wholesale%20customers%20data.csv'
df = pd.read_csv(url)

# Step 2: Visualize the data before clustering
sns.pairplot(df, diag_kind='kde')
plt.suptitle('Pairplot of Wholesale Customers Data', y=1.02)
plt.show()

# Step 3: Preprocess the data
X = df[['Grocery', 'Milk']].values  # Choosing only two features for easy visualization
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

```
# Step 4: Apply DBSCAN
dbscan = DBSCAN(eps=0.5, min_samples=5)
clusters = dbscan.fit_predict(X_scaled)

# Adding cluster labels to the original dataframe
df['Cluster'] = clusters

# Step 5: Visualize the data after clustering with legend
plt.figure(figsize=(10, 6))
unique_labels = set(clusters)
colors = [plt.cm.viridis(each) for each in np.linspace(0, 1,
len(unique_labels))]
cluster_names = {label: f'Cluster {label}' if label != -1 else 'Noise' for label in
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 = (clusters == k)
    xy = X[class_member_mask]
    plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col),
markeredgecolor='k', markersize=6, label=cluster_names[k])
```

```
plt.title('DBSCAN Clustering of Wholesale Customers')

plt.xlabel('Annual Spending on Grocery')

plt.ylabel('Annual Spending on Milk')

plt.legend(title='Clusters')

plt.show()

# Display the dataframe with cluster labels

df.head()
```

Explain Code

- **pandas as pd**: Imports the pandas library and assigns it the alias `pd`. Pandas is used for data manipulation and analysis.
- **matplotlib.pyplot as plt**: Imports the `pyplot` module from the matplotlib library, which provides a MATLAB-like interface for plotting.
- **seaborn as sns**: Imports the seaborn library, which is based on matplotlib and provides a high-level interface for drawing attractive statistical graphics.
- **from sklearn.preprocessing import StandardScaler**: Imports the `StandardScaler` class from `sklearn.preprocessing`, which is used to standardize features by removing the mean and scaling to unit variance.
- **from sklearn.cluster import DBSCAN**: Imports the `DBSCAN` class from `sklearn.cluster`, which implements the DBSCAN clustering algorithm.
- **import numpy as np**: Imports the numpy library and assigns it the alias `np`. Numpy is used for numerical operations.

- **url**: Defines the URL of the dataset to be loaded.
- **df = pd.read_csv(url)**: Reads the dataset from the specified URL into a pandas DataFrame `df`.

- **sns.pairplot(df, diag_kind='kde')**: Creates a pair plot for the DataFrame `df`. The `diag_kind='kde'` option specifies that Kernel Density Estimation plots should be used on the diagonal.
- **plt.suptitle('Pairplot of Wholesale Customers Data', y=1.02)**: Adds a title above the pair plot. The `y=1.02` argument adjusts the position of the title.
- **plt.show()**: Displays the plot.

- **X = df[['Grocery', 'Milk']].values**: Selects the 'Grocery' and 'Milk' columns from the DataFrame and converts them to a numpy array `X`. These columns are chosen for easy visualization in a 2D plot.
- **scaler = StandardScaler()**: Creates an instance of `StandardScaler`.

- **X_scaled = scaler.fit_transform(X)**: Standardizes the features by removing the mean and scaling to unit variance. The result is stored in `X_scaled`.

- **dbscan = DBSCAN(eps=0.5, min_samples=5)**: Creates an instance of the DBSCAN algorithm with `eps=0.5` (maximum distance between two samples for them to be considered as in the same neighborhood) and `min_samples=5` (minimum number of samples in a neighborhood for a point to be considered a core point).
- **clusters = dbscan.fit_predict(X_scaled)**: Fits the DBSCAN model to the standardized data `X_scaled` and returns the cluster labels for each point. These labels are stored in `clusters`.

df['Cluster'] = clusters: Adds a new column 'Cluster' to the DataFrame `df` containing the cluster labels assigned by DBSCAN.

- **plt.figure(figsize=(10, 6))**: Creates a new figure with the specified size (10 inches by 6 inches).
- **unique_labels = set(clusters)**: Gets the unique cluster labels from the `clusters` array.
- **colors = [plt.cm.viridis(each) for each in np.linspace(0, 1, len(unique_labels))]**: Generates a list of colors using the `viridis` colormap. `np.linspace(0, 1, len(unique_labels))` generates evenly spaced values between 0 and 1 for the number of unique labels.
- **cluster_names = {label: f'Cluster {label}' if label != -1 else 'Noise' for label in unique_labels}**: Creates a dictionary `cluster_names` that maps each cluster label to a string name, using 'Noise' for the label `-1`.

- **for k, col in zip(unique_labels, colors)**: Iterates over the unique cluster labels and their corresponding colors.
- **if k == -1**: Checks if the cluster label is `-1`, which indicates noise.
- **col = [0, 0, 0, 1]**: Sets the color to black for noise points.
- **class_member_mask = (clusters == k)**: Creates a boolean mask `class_member_mask` for points belonging to the current cluster `k`.
- **xy = X[class_member_mask]**: Selects the points in `X` that belong to the current cluster `k`.
- **plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col), markeredgecolor='k', markersize=6, label=cluster_names[k])**: Plots the points in `xy` using the specified color and labels them according to `cluster_names`.

- **plt.title('DBSCAN Clustering of Wholesale Customers')**: Sets the title of the plot.
- **plt.xlabel('Annual Spending on Grocery')**: Labels the x-axis.
- **plt.ylabel('Annual Spending on Milk')**: Labels the y-axis.
- **plt.legend(title='Clusters')**: Adds a legend to the plot with the title 'Clusters'.
- **plt.show()**: Displays the plot.

df.head(): Displays the first few rows of the DataFrame `df`, including the new 'Cluster' column with the cluster labels assigned by DBSCAN.

Key Differences of DBSCAN

1. **Cluster Shape**:
   - **DBSCAN**: Can find arbitrarily shaped clusters. It defines clusters based on the density of data points, which allows it to identify complex, non-linear shapes.
   - **K-means**: Assumes clusters are spherical and equally sized. It tends to form clusters that are roughly circular or spherical in shape.
   - **Hierarchical Clustering**: Can produce clusters of various shapes depending on the linkage criteria (single, complete, average, etc.), but it still tends to struggle with arbitrarily shaped clusters.
2. **Handling Noise**:
   - **DBSCAN**: Explicitly identifies noise points (outliers). Points that do not belong to any cluster are labeled as noise.
   - **K-means**: Does not have a built-in mechanism for handling noise. Every point is assigned to a cluster.
   - **Hierarchical Clustering**: Like K-means, it assigns every point to a cluster without explicitly identifying noise.
3. **Number of Clusters**:
   - **DBSCAN**: Does not require the number of clusters to be specified a priori. It determines the number of clusters based on the data density.
   - **K-means**: Requires the number of clusters to be specified before running the algorithm.
   - **Hierarchical Clustering**: Does not require the number of clusters to be specified a priori. The number of clusters can be chosen after the dendrogram is created.
4. **Parameter Sensitivity**:
   - **DBSCAN**: Requires two parameters: `eps` (maximum distance between two samples for them to be considered as in the same neighborhood) and `min_samples` (minimum number of points to form a dense region). The choice of these parameters can significantly affect the results.
   - **K-means**: Sensitive to the initial placement of centroids. Multiple runs with different initializations can lead to different results.
   - **Hierarchical Clustering**: The results depend on the linkage criteria used (single, complete, average, etc.) and the distance metric.
5. **Scalability**:
   - **DBSCAN**: Can be computationally intensive with a time complexity of O(n log n) for low-dimensional data. It can handle large datasets but can be slow for very large datasets or high-dimensional data.
   - **K-means**: Generally faster with a time complexity of O(n), where n is the number of data points. Suitable for large datasets.
   - **Hierarchical Clustering**: Typically slower with a time complexity of O(n^3), making it less suitable for large datasets.
6. **Cluster Initialization**:
   - **DBSCAN**: No need for initialization of cluster centers or initial conditions.
   - **K-means**: Requires initialization of cluster centers, which can affect the outcome.
   - **Hierarchical Clustering**: Does not require initialization but needs a method to measure distance between clusters.