Types of Unsupervised Machine Learning

In the realm of machine learning, Unsupervised Learning stands as a pillar, allowing algorithms to glean patterns and structures from data without explicit labels or guidance. This unit delves into three fundamental techniques within Unsupervised Learning: K-means clustering, Hierarchical Clustering, and Principal Component Analysis (PCA). These methods, each unique in its approach, empower machines to discover inherent structures and relationships within datasets.

Unsupervised learning is a type of machine learning where the algorithm is trained on unlabeled data without explicit guidance. The system tries to learn the patterns and the structure from the data without any predefined labels or categories. The goal is to allow the algorithm to explore the data and find hidden patterns or relationships.

Here are some common types of unsupervised learning:

- K-means: Partitioning Data into Clusters
- Hierarchical Clustering: **Agglomerative** and **Divisive** Techniques
- Principal Component Analysis (PCA)

Let's understand all of them one by one.

- **K-means: Partitioning Data into Clusters:** K-means clustering is a partitioning method that categorizes data points into K clusters based on their similarities. The algorithm iteratively assigns data points to clusters and updates the cluster centroids until convergence. The objective is to minimize the intra-cluster variance.
  a. **Intuition:**
     Imagine you have a bag of marbles of different colors, and your task is to group them based on color similarity. K-means is akin to sorting these marbles into K baskets, where each basket represents a cluster of marbles with similar colors.

b. Equations:

The algorithm's steps can be outlined as follows:

1. **Initialization:** Randomly select K data points as initial cluster centroids.
2. **Assignment:** Assign each data point to the nearest centroid, forming K clusters.
3. **Update:** Recalculate the centroids as the mean of the data points in each cluster.
4. **Repeat:** Repeat steps 2 and 3 until convergence (minimal change in centroids).

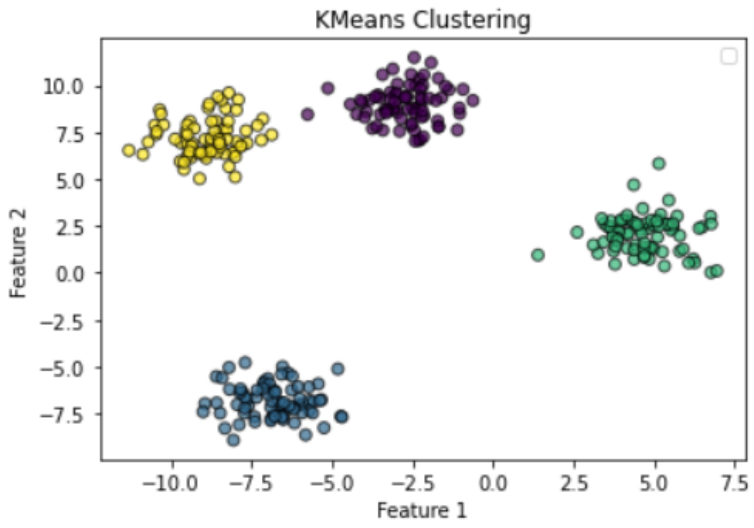
   The algorithm converges when the centroids stabilize, indicating that the assignment of data points to clusters has reached an optimal state.

c. Examples Code:

Consider a 2D dataset with points (x, y):

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

# Generate random data for clustering
np.random.seed(42)
X, _ = make_blobs(n_samples=300, centers=4,
random_state=42)

# Apply KMeans clustering
kmeans = KMeans(n_clusters=4)
kmeans.fit(X)

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

# Plot the original data points and cluster centers
plt.scatter(X[:, 0], X[:, 1], c=labels,
cmap='viridis', alpha=0.7, edgecolors='k')
plt.title('KMeans Clustering')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend()
plt.show()
```

Output:



Explanation of the Code:

In this example, the KMeans algorithm is used to cluster the 2D data into three clusters. **labels** contain the cluster assignments, and **centroids** store the final cluster centers.

Importing Libraries:

- numpy is imported as np for numerical operations.
- matplotlib.pyplot is imported as plt for plotting.
- KMeans is imported from sklearn.cluster to perform KMeans clustering.
- make_blobs is imported from sklearn.datasets to generate synthetic data with blobs.

Generating Random Data:

```
np.random.seed(42)
X, _ = make_blobs(n_samples=300, centers=4,
random_state=42)
```

- Sets a random seed for reproducibility.
- Generates synthetic data using make_blobs with 300 samples, forming 4 centers.

Applying KMeans Clustering:

```
kmeans = KMeans(n_clusters=4)
kmeans.fit(X)
```

- Creates a KMeans clustering model with 4 clusters.
- Fits the model to the generated data **X**.

Getting Cluster Centers and Labels:

```
centers = kmeans.cluster_centers_
labels = kmeans.labels_
```

Retrieves the cluster centers and labels assigned to each data point after fitting the KMeans model.

Plotting the Results:

```
plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis',
alpha=0.7, edgecolors='k')
plt.title('KMeans Clustering')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend()
plt.show()
```

- lots the original data points with colors assigned based on their cluster labels.
- **c=labels** determines the color of each point based on its assigned cluster.
- **cmap='viridis'** sets the color map.
- **alpha=0.7** sets the transparency of the points.
- **edgecolors='k'** adds black edges to the points for better visibility.
- Title, xlabel, ylabel are set for the plot.
- **plt.show()** displays the plot.

- **Hierarchical Clustering: Agglomerative and Divisive Techniques:**
  Hierarchical clustering builds a hierarchy of clusters, either by merging (agglomerative) or splitting (divisive) data points based on their similarities.

  a. **Intuition:** Imagine a family tree where individuals are grouped based on their genetic similarities. Agglomerative clustering starts with individuals and merges them into families, while divisive clustering starts with a large family and divides it into subfamilies.

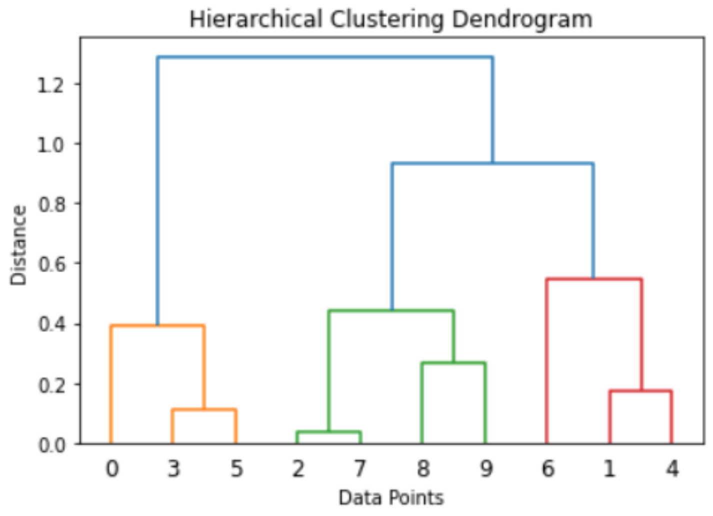
  b. **Equations:** The two primary hierarchical clustering techniques are:
     1. Agglomerative Clustering:
        i. Start with each point as a singleton cluster.
        ii. Iteratively merge the closest clusters until only one cluster remains.
     2. Divisive Clustering:
        i. Start with all points in one cluster.
        ii. Iteratively split the least **cohesive** cluster until each point is a singleton cluster.

     Agglomerative clustering is computationally more efficient, while divisive clustering can provide insight into the hierarchical structure from a top-down perspective.

  c. **Example Code:** Consider a 2D dataset and perform hierarchical clustering:

```
from scipy.cluster.hierarchy import dendrogram, linkage
import matplotlib.pyplot as plt

# Generating random data
np.random.seed(42)
X = np.random.rand(10, 2)

# Creating a linkage matrix using Ward's method
linkage_matrix = linkage(X, method='ward')

# Plotting the dendrogram
dendrogram(linkage_matrix)
plt.title('Hierarchical Clustering Dendrogram')
plt.xlabel('Data Points')
plt.ylabel('Distance')
plt.show()
```

Output:



Explanation of the Code:

This code snippet uses scipy to create a dendrogram, visualizing the hierarchy of clusters. The linkage matrix is computed using Ward's method, and the dendrogram shows the step-by-step merging of clusters.

Importing Libraries:

```
from scipy.cluster.hierarchy import dendrogram, linkage
import matplotlib.pyplot as plt
```

This imports the necessary functions from **scipy.cluster.hierarchy** for hierarchical clustering and the **matplotlib.pyplot** library for plotting.

Generating Random Data:

```
np.random.seed(42)
X = np.random.rand(10, 2)
```

It sets a random seed for reproducibility and generates a 10x2 array of random data points. In this case, each row represents a data point with two features.

Creating Linkage Matrix using Ward's Method:

```
linkage_matrix = linkage(X, method='ward')
```

The **linkage** function is used to calculate the linkage matrix based on the input data **X** and the specified method, which is 'ward' in this case. Ward's method is a hierarchical clustering method that minimizes the variance within each cluster.

Plotting the Dendrogram:

```
dendrogram(linkage_matrix)
plt.title('Hierarchical Clustering Dendrogram')
plt.xlabel('Data Points')
plt.ylabel('Distance')
plt.show()
```

The **dendrogram** function is used to visualize the hierarchical clustering. The linkage matrix is passed to this function, and then the plot is displayed with **plt.show()**. The title and axis labels are added for better interpretation.

- **Principal Component Analysis (PCA):** Principal Component Analysis (PCA) is a **dimensionality reduction** technique that transforms high-dimensional data into a lower-dimensional representation while retaining the most significant variance.

  a. Intuition: Imagine looking at a 3D object from different perspectives. PCA identifies the directions (principal components) where the data varies the most.

  b. Equations:

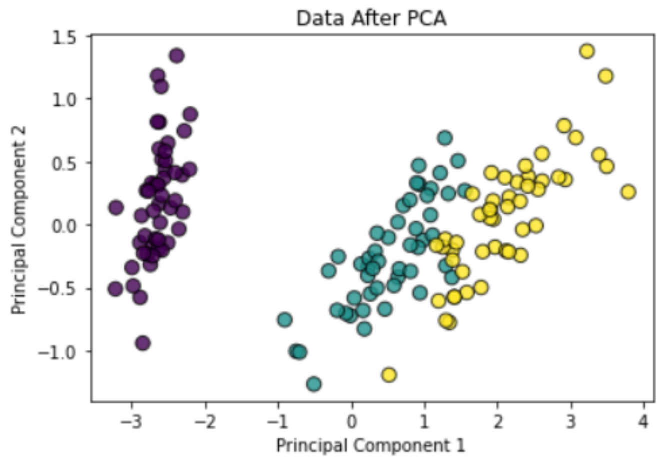
  1. **Covariance Matrix:** Compute the covariance matrix of the data.
  2. **Eigenvalue Decomposition:** Find the eigenvalues and eigenvectors of the covariance matrix.
  3. **Select Principal Components:** Order eigenvectors by eigenvalues and choose the top k (number of desired dimensions).
  4. **Projection:** Project the data onto the selected principal components.

The first principal component captures the most variance, followed by the second, and so on. PCA is widely used for visualization, **noise reduction**, and **feature extraction**.

c. **Example Code:** Consider a dataset with two features and perform PCA:

```
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.datasets import load_iris

# Load iris dataset
iris = load_iris()
X = iris.data
y = iris.target

# Apply PCA
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)

# Plot the data after PCA
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=y,
cmap='viridis', edgecolor='k', s=70, alpha=0.8)
plt.title('Data After PCA')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.show()

# Access explained variance
explained_variance_ratio = pca.explained_variance_ratio_
print(f"Explained Variance Ratio:
{explained_variance_ratio}")
```

Output:



```
Explained Variance Ratio: [0.92461872 0.05306648]
```

Explanation of the Code:

This code snippet demonstrates the use of Principal Component Analysis (PCA) from the scikit-learn library to reduce the dimensionality of the Iris dataset and visualize it in a 2D space.

Importing necessary libraries:

```
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.datasets import load_iris
```

This imports the required libraries, including Matplotlib for plotting, PCA for dimensionality reduction, and the Iris dataset.

Loading the Iris dataset:

```
iris = load_iris()
X = iris.data
y = iris.target
```

It loads the Iris dataset, where **X** contains the feature data, and **y** contains the corresponding labels.

Applying PCA:

```
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)
```

It creates a PCA object with the desired number of components (2 in this case) and applies PCA transformation to the original data.

Plotting the data after PCA:

```
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=y, cmap='viridis',
edgecolor='k', s=70, alpha=0.8)
plt.title('Data After PCA')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.show()
```

It creates a scatter plot of the data in the reduced 2D space, where each point is colored based on its class (target value). This visualization helps to understand the distribution of the data after PCA.

Accessing explained variance:

```
explained_variance_ratio = pca.explained_variance_ratio_
print(f"Explained Variance Ratio:
{explained_variance_ratio}")
```

It prints the explained variance ratio for each principal component. The explained variance ratio indicates the proportion of the dataset's variance that is captured by each principal component.