

Unit 3

Introduction to Decision Trees

A decision tree is a popular machine learning algorithm used for both classification and regression tasks. It works by recursively splitting the dataset into subsets based on the feature that provides the highest information gain, eventually forming a tree-like structure where each internal node represents a decision based on a feature, and each leaf node represents a class label (in classification) or a value (in regression).

Example of Classification Decision Tree

Let's consider a simple example where we use a decision tree to classify whether a person should play tennis based on weather conditions. The dataset includes features like outlook (sunny, overcast, rainy), temperature (hot, mild, cool), humidity (high, normal), and wind (weak, strong).

Dataset:

Outlook	Temperature	Humidity	Wind	PlayTennis
Sunny	Hot	High	Weak	No
Sunny	Hot	High	Strong	No
Overcast	Hot	High	Weak	Yes
Rainy	Mild	High	Weak	Yes
Rainy	Cool	Normal	Weak	Yes
Rainy	Cool	Normal	Strong	No
Overcast	Cool	Normal	Strong	Yes
Sunny	Mild	High	Weak	No
Sunny	Cool	Normal	Weak	Yes
Rainy	Mild	Normal	Weak	Yes
Sunny	Mild	Normal	Strong	Yes
Overcast	Mild	High	Strong	Yes
Overcast	Hot	Normal	Weak	Yes
Rainy	Mild	High	Strong	No

Measures of Impurity for Evaluating Splits in Decision Trees

1. Gini Impurity:

- Gini impurity measures the probability of a randomly chosen element being incorrectly classified.
- Formula: $Gini(D) = 1 - \sum_{i=1}^n p_i^2$ where (p_i) is the probability of an element being classified to class (i) .

2. Entropy (Information Gain):

- Entropy measures the amount of uncertainty in the dataset.
- Formula: $Entropy(D) = - \sum_{i=1}^n p_i \log_2(p_i)$ where (p_i) is the proportion of class (i) in the dataset.
- Information Gain: $IG(T, x) = Entropy(T) - \sum_{v \in Values(x)} \frac{|T_v|}{|T|} Entropy(T_v)$ where (T) is the total set, (x) is the attribute, and (T_v) is the subset for each value (v) .

3. Chi-square:

- Measures the statistical significance of the splits.

4. Reduction in Variance:

- Used for regression trees to minimize the variance within subsets.

ID3 Decision Trees

The ID3 algorithm (Iterative Dichotomiser 3) uses entropy and information gain to construct a decision tree.

Steps:

- Calculate the entropy of the entire dataset.
- For each feature, calculate the information gain by splitting the dataset based on the feature's values.
- Choose the feature with the highest information gain.
- Repeat the process recursively for each subset until a stopping criterion is met (e.g., all instances have the same label or no features are left).

Pruning the Tree

Pruning is used to reduce the complexity of the decision tree, avoid overfitting, and improve generalization.

1. Pre-pruning (Early Stopping):

- Stop the tree growth early based on a criterion like maximum depth, minimum samples per leaf, or minimum information gain.

2. Post-pruning (Prune After Training):

- Grow the tree fully and then remove nodes that do not provide additional power by using techniques like reduced error pruning or cost-complexity pruning.

Metrics for Assessing Classification Accuracy

1. Accuracy:

- Ratio of correctly predicted instances to the total instances.
- Formula: $(\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN})$ where TP = True Positives, TN = True Negatives, FP = False Positives, FN = False Negatives.

2. Precision:

- Ratio of correctly predicted positive observations to the total predicted positives.
- Formula: $(\text{Precision} = \frac{TP}{TP + FP})$

3. Recall (Sensitivity or True Positive Rate):

- Ratio of correctly predicted positive observations to all observations in the actual class.
- Formula: $(\text{Recall} = \frac{TP}{TP + FN})$

4. F1 Score:

- Harmonic mean of precision and recall.
- Formula: $(\text{F1 Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}})$

5. Confusion Matrix:

- A matrix that shows the count of true positive, true negative, false positive, and false negative predictions.

6. ROC-AUC:

- Receiver Operating Characteristic - Area Under Curve. Measures the trade-off between true positive rate and false positive rate.

Exikit-learn` library. The dataset is small and simple for illustration purposes. In practice, you would work with larger and more complex datasets.

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

# Example dataset
data = [
    ['Sunny', 'Hot', 'High', 'Weak', 'No'],
    ['Sunny', 'Hot', 'High', 'Strong', 'No'],
    ['Overcast', 'Hot', 'High', 'Weak', 'Yes'],
    ['Rainy', 'Mild', 'High', 'Weak', 'Yes'],
    ['Rainy', 'Cool', 'Normal', 'Weak', 'Yes'],
    ['Rainy', 'Cool', 'Normal', 'Strong', 'No'],
    ['Overcast', 'Cool', 'Normal', 'Strong', 'Yes'],
    ['Sunny', 'Mild', 'High', 'Weak', 'No'],
    ['Sunny', 'Cool', 'Normal', 'Weak', 'Yes'],
    ['Rainy', 'Mild', 'Normal', 'Weak', 'Yes'],
    ['Sunny', 'Mild', 'Normal', 'Strong', 'Yes'],
    ['Overcast', 'Mild', 'High', 'Strong', 'Yes'],
    ['Overcast', 'Hot', 'Normal', 'Weak', 'Yes'],
    ['Rainy', 'Mild', 'High', 'Strong', 'No']
]

# Convert the data to a DataFrame
import pandas as pd
df = pd.DataFrame(data, columns=['Outlook', 'Temperature', 'Humidity', 'Wind', 'PlayTennis'])

# Encoding categorical features
df = pd.get_dummies(df, columns=['Outlook', 'Temperature', 'Humidity', 'Wind'])

# Features and target variable
X = df.drop('PlayTennis', axis=1)
y = df['PlayTennis']

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Decision tree classifier
clf = DecisionTreeClassifier(criterion='entropy')
clf.fit(X_train, y_train)

# Predictions
y_pred = clf.predict(X_test)

# Evaluation
print("Accuracy:", accuracy_score(y_test, y_pred))
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
print("Classification Report:\n", classification_report(y_test, y_pred))
```



```
Accuracy: 1.0
Confusion Matrix:
[[1 0]
 [0 2]]
```

Classification Report:					
	precision	recall	f1-score	support	
No	1.00	1.00	1.00	1	
Yes	1.00	1.00	1.00	2	
accuracy			1.00	3	
macro avg			1.00	3	
weighted avg			1.00	3	

▾
 Unit 4

▾
 Clustering Overview

Clustering is a type of unsupervised learning used to group similar data points into clusters. Unlike supervised learning, clustering does not rely on labeled data. Instead, it finds patterns and structures within the data based on similarity or distance measures.

Unsupervised Learning

Unsupervised learning involves training a model on data without labeled responses. The goal is to infer the natural structure present within a set of data points. Clustering is one of the primary methods of unsupervised learning.

Types of Clustering

1. Centroid-based Clustering:

- Clusters are represented by central points, called centroids.
- Each data point is assigned to the nearest centroid.
- Examples: K-means, K-medoids.

2. Density-based Clustering:

- Clusters are formed based on the density of data points in the region.
- Can find arbitrarily shaped clusters.
- Examples: DBSCAN, OPTICS.

3. Hierarchical Clustering:

- Clusters are formed by creating a hierarchy of data points.
- Can be agglomerative (bottom-up) or divisive (top-down).
- Examples: Agglomerative Clustering, Divisive Clustering.

4. Distribution-based Clustering:

- Assumes data points are generated from a mixture of probability distributions.
- Examples: Gaussian Mixture Models (GMM).

5. Grid-based Clustering:

- Divides the data space into a grid structure and forms clusters based on the density of data points in the grid cells.
- Examples: STING, CLIQUE.

Centroid-based Clustering

In centroid-based clustering, clusters are defined by central points, and each data point belongs to the cluster with the nearest centroid. The most common algorithm in this category is K-means.

K-means Clustering

K-means is a popular and simple clustering algorithm.

Steps:

1. Choose the number of clusters (K).
2. Initialize (K) centroids randomly.
3. Assign each data point to the nearest centroid.
4. Recalculate the centroids as the mean of all points assigned to each centroid.
5. Repeat steps 3 and 4 until convergence (no changes in assignments or centroids).

Advantages:

- Simple and easy to implement.
- Scales well with large datasets.

Disadvantages:

- Requires the number of clusters (K) to be specified.
- Sensitive to initial placement of ce) plt.ylabel('Principal Component 2') plt.show()

PCA helps in reducing the dimensionality of the data, making it easier to visualize and more efficient to cluster.

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

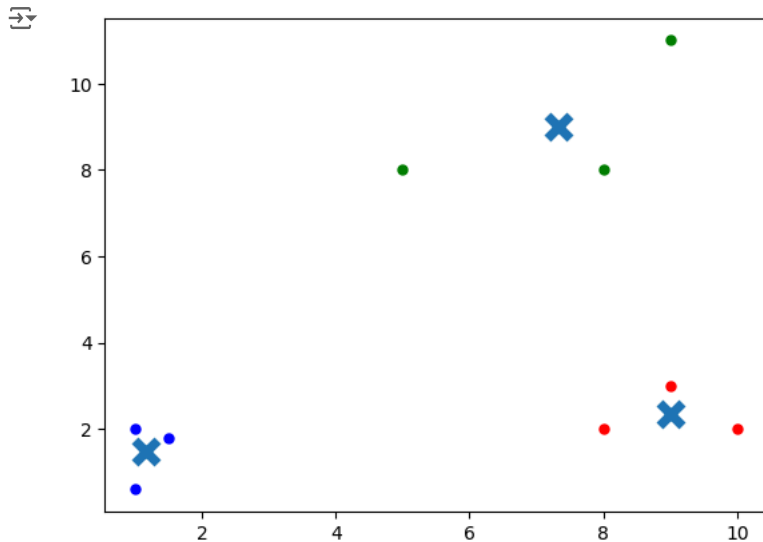
# Example data
data = np.array([
    [1, 2], [1.5, 1.8], [5, 8], [8, 8], [1, 0.6], [9, 11],
    [8, 2], [10, 2], [9, 3]
])

# K-means clustering with explicit n_init parameter
kmeans = KMeans(n_clusters=3, n_init=10)
kmeans.fit(data)

# Cluster centers and labels
centroids = kmeans.cluster_centers_
labels = kmeans.labels_

# Plotting
colors = ["g.", "r.", "b."]

for i in range(len(data)):
    plt.plot(data[i][0], data[i][1], colors[labels[i]], markersize=10)
plt.scatter(centroids[:, 0], centroids[:, 1], marker='x', s=150, linewidths=5, zorder=10)
plt.show()
```



✓ Hierarchical Clustering

Hierarchical clustering builds a tree-like structure (dendrogram) to represent the nested groupings of data points.

Types:

1. Agglomerative (Bottom-up):

- Starts with each data point as a separate cluster.
- Iteratively merges the closest pairs of clusters until all points are in a single cluster.

2. Divisive (Top-down):

- Starts with all data points in a single cluster.
- Iteratively splits clusters until each point is in its own cluster.

Linkage Criteria:

- **Single Linkage:** Minimum distance between points in different clusters.
- **Complete Linkage:** Maximum distance between points in different clusters.
- **Average Linkage:** Average distance between points in different clusters.
- **Ward's Method:** Minimizes the variance within clusters.

```

from sklearn.cluster import AgglomerativeClustering
import scipy.cluster.hierarchy as shc
import matplotlib.pyplot as plt
import numpy as np

```

```

# Example data

```

```

data = np.array([
    [1, 2], [1.5, 1.8], [5, 8], [8, 8], [1, 0.6], [9, 11],
    [8, 2], [10, 2], [9, 3]
])

```

```

# Dendrogram

```

```

plt.figure(figsize=(10, 7))
plt.title("Dendrogram")
dend = shc.dendrogram(shc.linkage(data, method='ward'))
plt.show()

```

```

# Hierarchical clustering

```

```

cluster = AgglomerativeClustering(n_clusters=3, affinity='euclidean', linkage='ward')
labels = cluster.fit_predict(data)

```

```

# Plotting

```

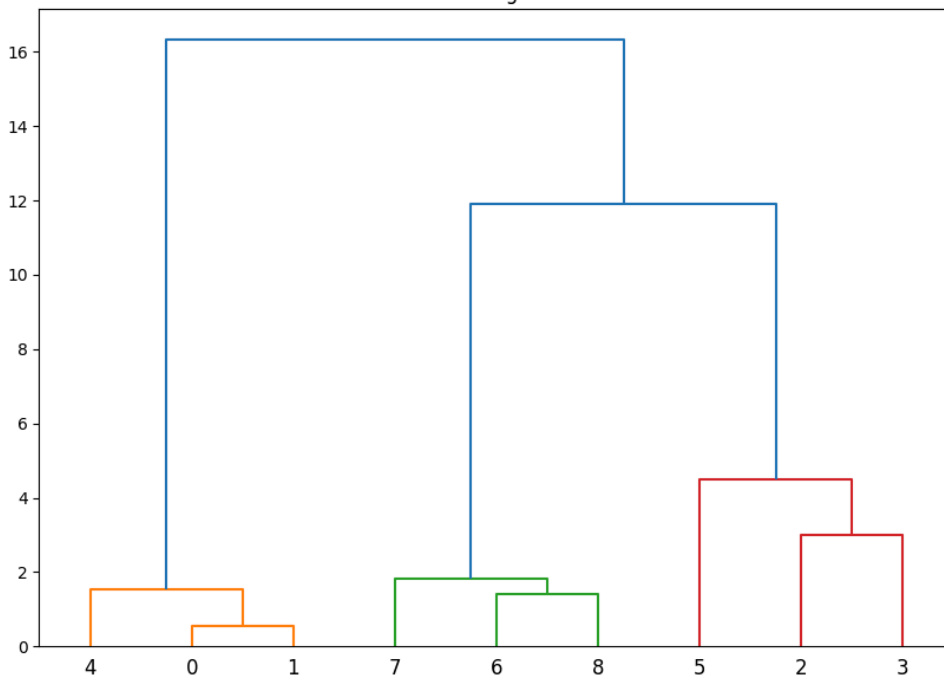
```

plt.scatter(data[:, 0], data[:, 1], c=labels, cmap='rainbow')
plt.show()

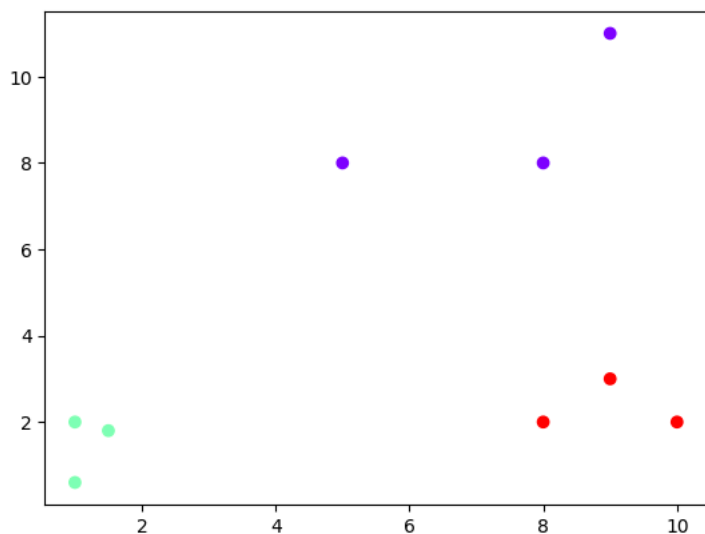
```



Dendrogram



/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_agglomerative.py:983: FutureWarning: A
warnings.warn(



✓ Principal Component Analysis (PCA)

PCA is a dimensionality reduction technique often used before clustering to reduce the number of features while preserving the variance in the data.

Steps:

1. Standardize the data.
2. Compute the covariance matrix.
3. Compute the eigenvectors and eigenvalues of the covariance matrix.
4. Sort eigenvalues and select the top (k) eigenvectors.
5. Transform the data to the new (k)-dimensional space.

Advantages:

- Reduces the complexity of the data.
- Can reveal the internal structure of the data.

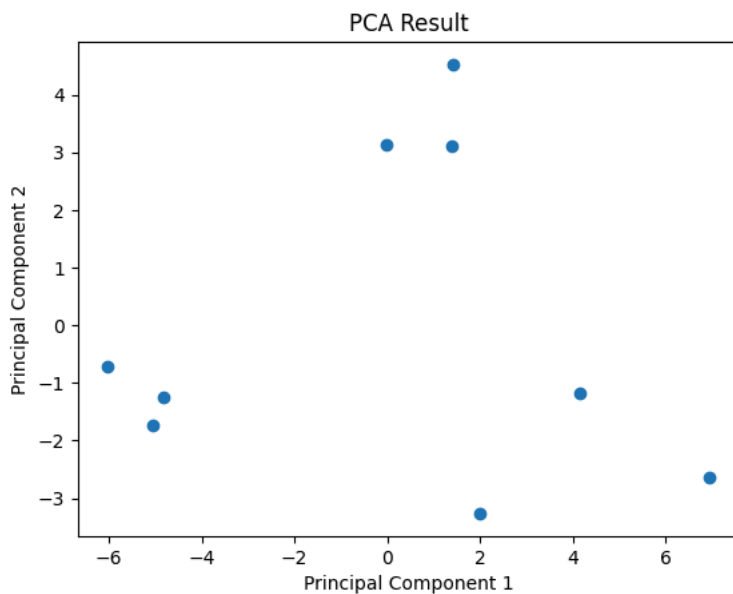
Example in Python:

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

# Example data
data = np.array([
    [1, 2], [1.5, 1.8], [5, 8], [8, 8], [1, 0.6], [9, 11],
    [8, 2], [10, 2], [9, 3]
])

# PCA transformation
pca = PCA(n_components=2)
transformed_data = pca.fit_transform(data)

# Plotting
plt.scatter(transformed_data[:, 0], transformed_data[:, 1])
plt.title('PCA Result')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.show()
```



✓ Unit 5

✓ 1. Neuron Models

Neuron models are mathematical representations of biological neurons, used in artificial neural networks. The most basic model is the perceptron.

Perceptron

- **Inputs:** ($\mathbf{x} = [x_1, x_2, \dots, x_n]$)
- **Weights:** ($\mathbf{w} = [w_1, w_2, \dots, w_n]$)
- **Bias:** (b)
- **Activation function:** (f)
- **Output:** ($y = f(\mathbf{w} \cdot \mathbf{x} + b)$)

2. Network Architectures

Network architectures refer to the structure of the neural network, which includes the arrangement of neurons and layers.

Types of Architectures

- **Single-layer Perceptron:** Consists of a single layer of output nodes connected directly to input nodes.
- **Multi-layer Perceptron (MLP):** Consists of multiple layers (input layer, hidden layers, and output layer).

3. Perceptrons

A perceptron is the simplest type of artificial neural network and is used for binary classifications.

4. The Error-Correction Delta Rule

The delta rule is used to update the weights of the perceptron to minimize the error.

$$\Delta w_i = \eta (d - y) x_i$$

Where:

- Δw_i is the change in weight.
- η is the learning rate.
- d is the desired output.
- y is the actual output.
- x_i is the input.

5. Multi-Layer Perceptron (MLP) Networks

MLPs are a class of feedforward artificial neural network (ANN). They consist of at least three layers of nodes: an input layer, one or more hidden layers, and an output layer.

Characteristics

- **Non-linearity:** Each neuron uses a non-linear activation function.
- **Layers:** Can have multiple layers to learn complex patterns.

6. Error-Backpropagation Algorithm

Backpropagation is used to train MLPs by updating the weights to minimize the error.

Steps:

1. **Forward Pass:** Calculate the output by passing inputs through the network.
2. **Compute Error:** Compute the difference between the predicted output and the actual output.
3. **Backward Pass:** Calculate the gradient of the loss function with respect to each weight using the chain rule.
4. **Update Weights:** Adjust the weights using the gradients to minimize the error.

Mathematical Formulation

For each weight (w) in the network:

$$w_{\text{new}} = w_{\text{old}} - \eta \frac{\partial E}{\partial w}$$

Where:

- E is the error.
- η is the learning rate.
- $\frac{\partial E}{\partial w}$ is the gradient of the error with respect to the weight.

Example Code: Training a Simple MLP Using Backpropagation

Here's a simple implementation. The activation function used is the sigmoid function.

Feel free to ask if you need more details or further assistance!

```
import numpy as np

# Activation function (Sigmoid)
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Derivative of sigmoid
def sigmoid_derivative(x):
    return x * (1 - x)

# Input dataset
inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])

# Output dataset
outputs = np.array([[0], [1], [1], [0]])

# Initialize weights and biases
input_layer_neurons = inputs.shape[1]
hidden_layer_neurons = 2
output_neurons = 1

# Hidden to Output weights
hidden_output_weights = np.random.randn(hidden_layer_neurons, output_neurons)
# Input to Hidden weights
input_hidden_weights = np.random.randn(input_layer_neurons, hidden_layer_neurons)
# Biases
hidden_bias = np.random.randn(hidden_layer_neurons)
output_bias = np.random.randn(output_neurons)
```

```

hidden_weights = np.random.uniform(size=(input_layer_neurons, hidden_layer_neurons))
hidden_bias = np.random.uniform(size=(1, hidden_layer_neurons))
output_weights = np.random.uniform(size=(hidden_layer_neurons, output_neurons))
output_bias = np.random.uniform(size=(1, output_neurons))

# Learning rate
lr = 0.1

# Training the MLP
for epoch in range(10000):
    # Forward pass
    hidden_layer_activation = np.dot(inputs, hidden_weights)
    hidden_layer_activation += hidden_bias
    hidden_layer_output = sigmoid(hidden_layer_activation)

    output_layer_activation = np.dot(hidden_layer_output, output_weights)
    output_layer_activation += output_bias
    predicted_output = sigmoid(output_layer_activation)

    # Calculate error
    error = outputs - predicted_output






    # Backward pass
    d_predicted_output = error * sigmoid_derivative(predicted_output)

    error_hidden_layer = d_predicted_output.dot(output_weights.T)
    d_hidden_layer = error_hidden_layer * sigmoid_derivative(hidden_layer_output)

    # Update weights and biases
    output_weights += hidden_layer_output.T.dot(d_predicted_output) * lr
    output_bias += np.sum(d_predicted_output, axis=0, keepdims=True) * lr
    hidden_weights += inputs.T.dot(d_hidden_layer) * lr
    hidden_bias += np.sum(d_hidden_layer, axis=0, keepdims=True) * lr

# Output the final predicted outputs
print(predicted_output)

```

→     

```

[[0.05607618]
 [0.94892898]
 [0.9486645 ]
 [0.05511413]]

```