

Experiment no:06

Aim:Experiment to implement any one of the classification algorithms(Decision tree/Naive Bayes) /Technique using python.

Theory:

Classification is a supervised learning technique in machine learning used to categorize data points into predefined classes or labels. The goal of classification is to learn a mapping function from input features to output labels based on training data. It is commonly used in applications such as spam detection, sentiment analysis, medical diagnosis, and credit scoring.

Key Concepts in Classification:

1. **Features:** These are the input variables or attributes used to predict the class label of data points. Features can be categorical or numerical.
2. **Classes:** These are the predefined categories or labels that the classifier aims to predict for each data point.
3. **Training Data:** This is a labeled dataset used to train the classification model. It consists of input feature vectors along with their corresponding class labels.
4. **Classification Model:** This is the learned mapping function that predicts the class label of unseen data points based on their input features.
5. **Evaluation Metrics:** These are measures used to evaluate the performance of a classification model, such as accuracy, precision, recall, F1-score, and area under the ROC curve (AUC-ROC).

Naive Bayes:

Naive Bayes is a probabilistic classification algorithm based on Bayes' theorem with an assumption of independence between features. Despite its simplicity, it often performs well in practice and is widely used for text classification tasks such as spam detection and sentiment analysis.

Working of Naive Bayes:

1. **Training:** Naive Bayes learns the probability of each class given the input features from the training data. It calculates the likelihood of each feature occurring in each class and the prior probability of each class.
2. **Prediction:** To predict the class label of a new data point, Naive Bayes applies Bayes' theorem to calculate the conditional probability of each class given the input features. The class with the highest conditional probability is chosen as the predicted class.
3. **Assumption of Independence:** Naive Bayes assumes that all features are conditionally independent given the class label. This simplifies the calculation of probabilities and makes the algorithm computationally efficient.

Decision Tree:

Decision tree is a non-parametric supervised learning algorithm used for classification and regression tasks. It builds a tree-like model of decisions based on input features, where each internal node represents a decision based on a feature, each branch represents an outcome of the decision, and each leaf node represents the class label or regression value.

Working of Decision Tree:

1. Feature Selection: Decision tree selects the best feature to split the data at each node based on criteria such as Gini impurity, entropy, or information gain. It chooses the feature that maximizes the purity of the resulting subsets.
2. Splitting: The selected feature is used to split the dataset into two or more subsets based on its possible values. This process is repeated recursively for each subset until a stopping criterion is met, such as reaching a maximum tree depth or minimum number of samples at a node.
3. Prediction: To predict the class label of a new data point, Decision tree traverses the tree from the root node to a leaf node based on the values of its input features. The class label associated with the leaf node reached by the data point is assigned as the predicted class.

Use of Naive Bayes and Decision Tree:

- Naive Bayes is commonly used in text classification tasks, such as spam detection and sentiment analysis, due to its simplicity and efficiency.
- Decision trees are widely used for both classification and regression tasks, as they are easy to interpret and visualize. They are also used in ensemble methods such as Random Forest and Gradient Boosting for improved performance.

Implementation:

1. Preprocess data. Split data into train and test set

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
import matplotlib.pyplot as plt
import pydotplus
from IPython.display import Image
```

```
[ ] # Load data
    data = pd.read_csv('/content/Credit Score Classification Dataset.csv')

[ ] # Handling missing values (if any)
    data.fillna(method='ffill', inplace=True) # Forward fill missing values
```

```
[ ] # Encode categorical variables
label_encoder = LabelEncoder()
data['Gender'] = label_encoder.fit_transform(data['Gender'])
data['Marital Status'] = label_encoder.fit_transform(data['Marital Status'])
data['Home Ownership'] = label_encoder.fit_transform(data['Home Ownership'])
data['Education'] = label_encoder.fit_transform(data['Education'])
```

```
▶ # Split data into features and target variable
X = data.drop(columns=['Credit Score']) # Features
y = data['Credit Score'] # Target variable
```

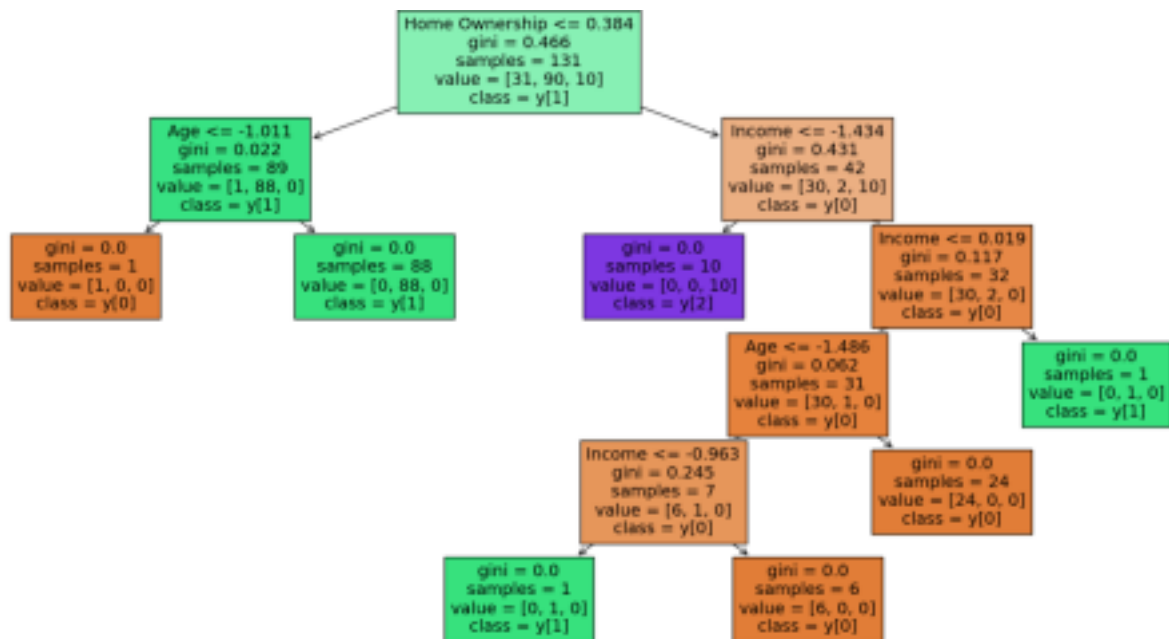
```
[ ] # Split data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

2. Build a Classification model using the inbuilt library function on training data

```
▶ # Scale numerical features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

```
[ ] # Decision Tree using inbuilt function
dt_model_inbuilt = DecisionTreeClassifier()
dt_model_inbuilt.fit(X_train_scaled, y_train)
dt_y_pred_inbuilt = dt_model_inbuilt.predict(X_test_scaled)
```

```
[ ] # Decision Tree using inbuilt function with visualization
plt.figure(figsize=(20, 10))
plot_tree(dt_model_inbuilt, feature_names=X.columns, class_names=True, filled=True)
plt.show()
```



3. Calculate metrics based on test data using an inbuilt function

```
[ ] # Calculate evaluation metrics for Decision Tree (inbuilt)
dt_accuracy_inbuilt = accuracy_score(y_test, dt_y_pred_inbuilt)
dt_precision_inbuilt = precision_score(y_test, dt_y_pred_inbuilt, average='macro')
dt_recall_inbuilt = recall_score(y_test, dt_y_pred_inbuilt, average='macro')
dt_f1_inbuilt = f1_score(y_test, dt_y_pred_inbuilt, average='macro')
```

```
[ ] print("Accuracy:", dt_accuracy_inbuilt)
print("Precision:", dt_precision_inbuilt)
print("Recall:", dt_recall_inbuilt)
print("F1-score:", dt_f1_inbuilt)
```

```
Accuracy: 1.0
Precision: 1.0
Recall: 1.0
F1-score: 1.0
```

Naive Bayes using InBuilt Function:

```
[ ] # Naive Bayes using inbuilt function
nb_model_inbuilt = GaussianNB()
nb_model_inbuilt.fit(X_train_scaled, y_train)
nb_y_pred_inbuilt = nb_model_inbuilt.predict(X_test_scaled)

# Calculate evaluation metrics for Naive Bayes (inbuilt)
nb_accuracy_inbuilt = accuracy_score(y_test, nb_y_pred_inbuilt)
nb_precision_inbuilt = precision_score(y_test, nb_y_pred_inbuilt, average='macro')
nb_recall_inbuilt = recall_score(y_test, nb_y_pred_inbuilt, average='macro')
nb_f1_inbuilt = f1_score(y_test, nb_y_pred_inbuilt, average='macro')

[ ] print("Accuracy:", nb_accuracy_inbuilt)
print("Precision:", nb_precision_inbuilt)
print("Recall:", nb_recall_inbuilt)
print("F1-score:", nb_f1_inbuilt)

Accuracy: 0.8787878787878788
Precision: 0.7791580400276054
Recall: 0.7855072463768117
F1-score: 0.7632850241545893
```

4. Build a Classification model using a UserDefined function

Decision Tree using user-defined function

class Node:

```
    def __init__(self, feature_index=None, threshold=None, left=None,
right=None, value=None):

        self.feature_index = feature_index
        self.threshold = threshold
        self.left = left
        self.right = right
        self.value = value # Value if the node is a leaf
```

class DecisionTree:

```
    def __init__(self, max_depth=None):
        self.max_depth = max_depth

    def _entropy(self, y):
        _, counts = np.unique(y, return_counts=True)
        probabilities = counts / len(y)
        entropy = -np.sum(probabilities * np.log2(probabilities))
```

```

        return entropy

def _information_gain(self, X, y, feature_index, threshold):

    left_indices = X[:, feature_index] < threshold
    y_left, y_right = y[left_indices], y[~left_indices]
    if len(y_left) == 0 or len(y_right) == 0:
        return 0
    p_left = len(y_left) / len(y)
    p_right = len(y_right) / len(y)
    gain = self._entropy(y) - (p_left * self._entropy(y_left) +
p_right * self._entropy(y_right))
    return gain

def _find_best_split(self, X, y):
    best_gain = 0
    best_feature_index = None
    best_threshold = None
    for feature_index in range(X.shape[1]):
        thresholds = np.unique(X[:, feature_index])
        for threshold in thresholds:
            gain = self._information_gain(X, y, feature_index,
threshold)

            if gain > best_gain:
                best_gain = gain
                best_feature_index = feature_index
                best_threshold = threshold
    return best_feature_index, best_threshold

def _build_tree(self, X, y, depth):
    if len(np.unique(y)) == 1 or depth == self.max_depth:
        return Node(value=np.argmax(np.bincount(y)))
    feature_index, threshold = self._find_best_split(X, y)
    left_indices = X[:, feature_index] < threshold
    X_left, X_right = X[left_indices], X[~left_indices]
    y_left, y_right = y[left_indices], y[~left_indices]
    left = self._build_tree(X_left, y_left, depth + 1)
    right = self._build_tree(X_right, y_right, depth + 1)
    return Node(feature_index, threshold, left, right)

```

```

def fit(self, X, y):
    self.tree = self._build_tree(X, y, 0)

def _predict_instance(self, x, node):

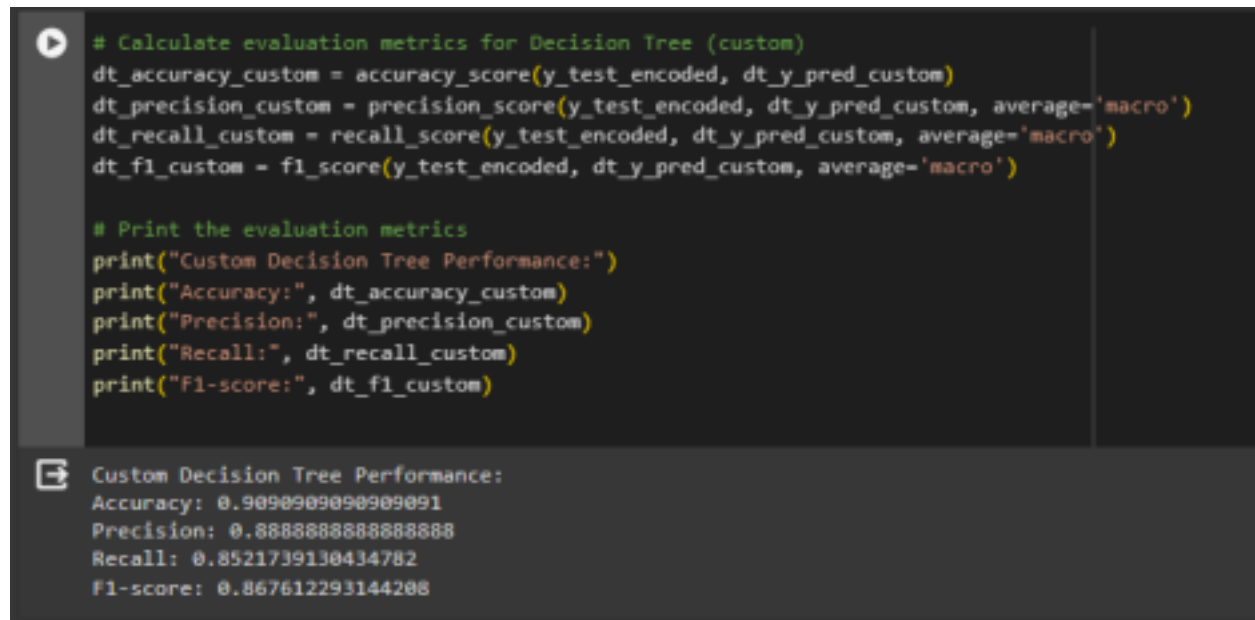
    if node.value is not None:
        return node.value
    if x[node.feature_index] < node.threshold:
        return self._predict_instance(x, node.left)
    else:
        return self._predict_instance(x, node.right)

def predict(self, X):
    return np.array([self._predict_instance(x, self.tree) for x in X])

# Initialize and train the Decision Tree model using self-defined function
dt_model_custom = DecisionTree()
dt_model_custom.fit(X_train_scaled, y_train_encoded)
dt_y_pred_custom = dt_model_custom.predict(X_test_scaled)

```

5. Calculate metrics based on test data using an inbuilt function



```

# Calculate evaluation metrics for Decision Tree (custom)
dt_accuracy_custom = accuracy_score(y_test_encoded, dt_y_pred_custom)
dt_precision_custom = precision_score(y_test_encoded, dt_y_pred_custom, average='macro')
dt_recall_custom = recall_score(y_test_encoded, dt_y_pred_custom, average='macro')
dt_f1_custom = f1_score(y_test_encoded, dt_y_pred_custom, average='macro')

# Print the evaluation metrics
print("Custom Decision Tree Performance:")
print("Accuracy:", dt_accuracy_custom)
print("Precision:", dt_precision_custom)
print("Recall:", dt_recall_custom)
print("F1-score:", dt_f1_custom)

```

Custom Decision Tree Performance:
Accuracy: 0.9090909090909091
Precision: 0.8888888888888888
Recall: 0.8521739130434782
F1-score: 0.867612293144208

Naive Bayes using UserDefined Function:

```
[ ] class NaiveBayes:
    def __init__(self):
        self.class_probabilities = None
        self.feature_probabilities = None

    def fit(self, X, y):
        self.classes = np.unique(y)
        self.class_probabilities = {c: np.mean(y == c) for c in self.classes}
        self.feature_probabilities = {c: {} for c in self.classes}

        for c in self.classes:
            X_c = X[y == c]
            for feature in range(X.shape[1]):
                mean = np.mean(X_c[:, feature])
                std = np.std(X_c[:, feature])
                self.feature_probabilities[c][feature] = (mean, std)

    def _calculate_probability(self, x, mean, std):
        exponent = np.exp(-((x - mean) ** 2) / (2 * std ** 2))
        return (1 / (np.sqrt(2 * np.pi) * std)) * exponent

    def predict(self, X):
        predictions = []
        for x in X:
            probabilities = {c: self.class_probabilities[c] for c in self.classes}
            for c in self.classes:
                for feature, (mean, std) in self.feature_probabilities[c].items():
                    probabilities[c] *= self._calculate_probability(x[feature], mean, std)
            predictions.append(max(probabilities, key=probabilities.get))
        return predictions
```

Performance Evaluation:


```
[ ] # Initialize and train the Naive Bayes model using the user-defined function
nb_model_custom = NaiveBayes()
nb_model_custom.fit(X_train_scaled, y_train)

# Make predictions on the test set
nb_y_pred_custom = nb_model_custom.predict(X_test_scaled)

# Calculate evaluation metrics for Naive Bayes (custom)
nb_accuracy_custom = accuracy_score(y_test, nb_y_pred_custom)
nb_precision_custom = precision_score(y_test, nb_y_pred_custom, average='macro')
nb_recall_custom = recall_score(y_test, nb_y_pred_custom, average='macro')
nb_f1_custom = f1_score(y_test, nb_y_pred_custom, average='macro')

# Print the evaluation metrics
print("Custom Naive Bayes Performance:")
print("Accuracy:", nb_accuracy_custom)
print("Precision:", nb_precision_custom)
print("Recall:", nb_recall_custom)
print("F1-score:", nb_f1_custom)
```

```
Custom Naive Bayes Performance:
Accuracy: 0.7878787878787878
Precision: 0.4521739130434783
Recall: 0.5855072463768116
F1-score: 0.49661835748792277
```

Comparison of Decision Tree/ Naive Bayes using Inbuilt function and userDefined Fucntion:

	Decision Tree(In Built Fn) Decision Tree (User Defined)
Accuracy	1.0 0.9090909090909091
Precision	1.0 0.8888888888888888
Recall	1.0 0.8521739130434782
F1-Score	1.0 0.867612293144208

	Naive Bayes(In Built Fn) Naive Bayes (User Defined)
Accuracy	0.8787878787878788 0.7878787878787878
Precision	0.7791580400276054 0.4521739130434783
Recall	0.7855072463768117 0.5855072463768116
F1-Score	0.7632850241545893 0.49661835748792277

Conclusion:We have Successfully Understood and implemented Decision Tree

and Naive Bayes Algorithm using InBuilt Function as well as UserDefined Function.