Name:	Aaryan Chandrakant Gole
Roll No:	10
Class/Sem:	TE/V
Experiment No.:	7
Title:	Implementation of Decision Tree using languages like JAVA/ Python.
Date of Performance:	
Date of Submission:	
Marks:	
Sign of Faculty:	

Aim: To implement Naïve Bayesian classification

Objective

Develop a program to implement a Decision Tree classifier.

Theory

Decision Tree is a popular supervised learning algorithm used for both classification and regression tasks. It operates by recursively partitioning the data into subsets based on the most significant attribute, creating a tree structure where leaf nodes represent the class labels.

Steps in Decision Tree Classification:

1. **Tree Construction**: The algorithm selects the best attribute of the dataset at each node as the root of the tree. Instances are then split into subsets based on the attribute values.
2. **Attribute Selection**: Common metrics include Information Gain, Gini Index, or Gain Ratio, which measure the effectiveness of an attribute in classifying the data.
3. **Stopping Criteria**: The tree-building process stops when one of the stopping criteria is met, such as all instances in a node belonging to the same class, or when further splitting does not add significant value.
4. **Classification Decision**: New instances are classified by traversing the tree from the root to a leaf node, where the majority class determines the prediction.

Example

Given a dataset with attributes and corresponding class labels:

- Construct a decision tree by recursively selecting the best attributes for splitting.
- Use the tree to classify new instances by traversing from the root to the appropriate leaf node.

Code:

```
#Decision Tree
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix

#Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

#Initialize DecisionTreeClassifier
clf = DecisionTreeClassifier(random_state=42)

#Train the classifier
```

```
clf.fit(X_train, y_train)

#Make predictions
y_pred = clf.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
roc_auc = roc_auc_score(y_test, knn_model.predict_proba(X_test)[:, 1])
classification_rep = classification_report(y_test, y_pred)

print(f'Accuracy: {accuracy}')
print(f'Precision: {precision}')
print(f'Recall: {recall}')
print(f'ROC AUC Score: {roc_auc}')
print(f'Classification Report:\n{classification_rep}')
```

Output:

- Predict the class label for new instances based on the constructed decision tree.

```
Accuracy: 0.9406392694063926
Precision: 0.21052631578947367
Recall: 0.26666666666666666
ROC AUC Score: 0.6367218282111899
Classification Report:
              precision    recall  f1-score   support

           0       0.97      0.96      0.97       846
           1       0.21      0.27      0.24        30

    accuracy                           0.94       876
   macro avg       0.59      0.62      0.60       876
weighted avg       0.95      0.94      0.94       876
```

Conclusion

Describe techniques or modifications to decision tree algorithms that can address issues caused by class imbalance in datasets.

To handle class imbalance in decision trees, you can:
1. Class Weight Adjustment: Assign higher weights to minority classes using the `class_weight='balanced'` parameter.
2. Resampling: Use oversampling (e.g., SMOTE) or undersampling to balance the dataset.
3. Ensemble Methods: Implement techniques like Balanced Random Forest or EasyEnsemble for better handling of imbalanced data.
4. Cost-sensitive Learning: Apply higher costs to misclassifying minority classes.
5. Pruning: Limit tree depth and size to prevent overfitting to the majority class.
6. Optimize Metrics: Focus on precision, recall, and F1-score rather than overall accuracy.