Name:	Saurabhsing Dipaksing Pardeshi
Roll No:	35
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 Decision Tree classifier.

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.

Output:

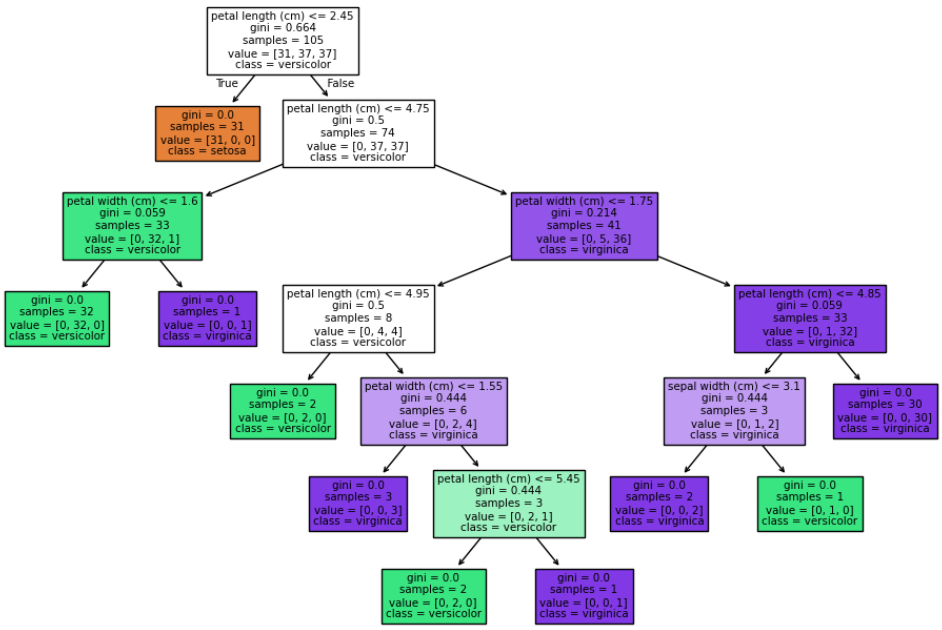
Decision Tree using languages like python.

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris
from sklearn.metrics import classification_report, accuracy_score
from sklearn import tree
import matplotlib.pyplot as plt

data = load_iris()
X = data.data
```

```
y = data.target

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

clf = DecisionTreeClassifier(criterion='gini', random_state=42)
clf.fit(X_train, y_train)

y_pred = clf.predict(X_test)

print("Accuracy:", accuracy_score(y_test, y_pred))
print("Classification Report:\n", classification_report(y_test, y_pred))

plt.figure(figsize=(12,8))
tree.plot_tree(clf, filled=True, feature_names=data.feature_names,
class_names=data.target_names)
plt.show()
```

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

```
Accuracy: 1.0
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       1.00      1.00      1.00        13
           2       1.00      1.00      1.00        13

    accuracy                           1.00        45
   macro avg       1.00      1.00      1.00        45
weighted avg       1.00      1.00      1.00        45
```



Conclusion

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

Class imbalance in datasets can cause decision trees to favor the majority class, leading to biased predictions. To address this, several techniques can be applied:

- Class Weighting: Decision trees can assign higher misclassification penalties to the minority class using class_weight='balanced'. This encourages the model to focus more on correctly classifying minority class examples.
- Resampling: Oversampling methods like SMOTE generate synthetic data for the minority class, while undersampling reduces the majority class size. Both help balance the dataset but come with risks—oversampling may cause overfitting, and undersampling can lead to information loss.
- Modified Splitting Criteria: Adjust the split criteria (like Gini or entropy) to consider class distribution, creating cost-sensitive trees. This encourages the model to create splits that better handle the minority class.
- Ensemble Methods: Random Forests and boosting algorithms like AdaBoost can be adapted by balancing class weights in their bootstrap samples or by focusing on misclassified minority instances. These ensemble techniques increase model robustness while addressing imbalance.
- Hybrid Approaches: Combining resampling with weighted decision trees or ensemble methods enhances performance by balancing the data and training process.
- Evaluation Metrics: Rather than relying on accuracy, metrics like precision, recall, F1-score, and AUC-ROC should be used to better evaluate performance on imbalanced datasets.