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.

Code:-

```
import pandas as pd

from sklearn.model_selection import train_test_split

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import classification_report, confusion_matrix, accuracy_score


# Load the dataset

data = pd.read_csv('Employee_performance.csv')


# Map categorical values to numerical

data['Project_Deadline'] = data['Project_Deadline'].map({'Met': 1, 'Missed': 0})

data['Performance'] = data['Performance'].map({'High': 2, 'Medium': 1, 'Low': 0})


# Features and target variable

X = data[['Work_Hours', 'Tasks_Completed', 'Teamwork_Score', 'Project_Deadline']]

y = data['Performance']


# Split the dataset

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)


# Train the Decision Tree Classifier

clf = DecisionTreeClassifier()

clf.fit(X_train, y_train)


# Predictions

y_pred = clf.predict(X_test)
```

```
# Evaluation metrics
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
print("\nClassification Report:\n", classification_report(y_test, y_pred))
accuracy = accuracy_score(y_test, y_pred) * 100
print(f"Accuracy: {accuracy:.2f}%")

# New data for prediction
new_data = pd.DataFrame({
    'Work_Hours': [38],
    'Tasks_Completed': [16],
    'Teamwork_Score': [7],
    'Project_Deadline': [0]
})

# Predicting performance for new data
predicted_performance = clf.predict(new_data)
performance_mapping = {2: 'High', 1: 'Medium', 0: 'Low'}
predicted_performance_label = performance_mapping[predicted_performance[0]]

print(f"The predicted performance for the new data is: {predicted_performance_label}")
```

Dataset: Employee_performance.csv

Work_Hours	Tasks_Completed	Teamwork_Score	Project_Deadline	Performance
45	20	8	Met	High
38	16	7	Missed	Medium
50	24	9	Met	High
30	12	6	Missed	Low
40	18	7	Met	Medium
42	20	8	Met	High
35	14	6	Missed	Low
48	22	9	Met	High
32	10	5	Missed	Low
43	19	8	Met	High

Output:

```
PS C:\Users\Lenovo\Desktop\DWM> python -u "c:\Users\Lenovo\Desktop\DWM\decision-tree.py"
Confusion Matrix:
 [[1 0]
 [0 1]]

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         1
           1       1.00      1.00      1.00         1

    accuracy                           1.00         2
   macro avg       1.00      1.00      1.00         2
weighted avg       1.00      1.00      1.00         2

Accuracy: 100.00%
The predicted performance for the new data is: Medium
PS C:\Users\Lenovo\Desktop\DWM>
```

Conclusion:

In this practical, we implemented a decision tree classifier to evaluate employee performance based on various metrics such as work hours, tasks completed, teamwork score, and project deadlines. We preprocessed the data and split it into training and testing sets to ensure robust evaluation. The model demonstrated a notable accuracy, indicating its effectiveness in predicting performance levels. Additionally, we visualized the decision tree to gain insights into the decision-making process.

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

Ans. Class imbalance can significantly impact the performance of decision tree algorithms, leading to biased predictions favoring the majority class. Here are some techniques and modifications that can help address issues caused by class imbalance:

1. **Resampling Techniques**:
   - **Oversampling**: Increase the number of instances in the minority class by duplicating existing samples or generating synthetic data (e.g., using SMOTE—Synthetic Minority Over-sampling Technique).
   - **Undersampling**: Reduce the number of instances in the majority class to balance the dataset. Care must be taken to avoid losing valuable information.
2. **Cost-sensitive Learning**:
   - Modify the decision tree algorithm to incorporate misclassification costs. By assigning a higher cost to misclassifying minority class instances, the model is encouraged to focus more on correctly classifying them.
3. **Class Weighting**:
   - Many decision tree implementations allow for class weights to be specified. By assigning higher weights to the minority class, the algorithm will prioritize minimizing errors on those instances.
4. **Ensemble Methods**:
   - **Bagging and Boosting**: Techniques like Random Forests or AdaBoost can be used, where multiple decision trees are trained on different samples. They can also use class weights or focus more on misclassified instances in subsequent iterations.
   - **Balanced Random Forest**: A variant of Random Forest that incorporates both oversampling of the minority class and downsampling of the majority class in each bootstrapped sample.
5. **Anomaly Detection Approaches**:
   - Treating the minority class as anomalies and applying anomaly detection algorithms can help identify instances that might be underrepresented.
6. **Evaluation Metrics**:
   - Instead of accuracy, use metrics that provide better insights into model performance on imbalanced datasets, such as precision, recall, F1-score, or area under the ROC curve (AUC-ROC).
7. **Tree Pruning**:

- Pruning can be adjusted to prevent overfitting to the majority class by focusing on generalizing to the minority class during the tree construction phase.

8. **Hybrid Approaches**:
   - Combine different strategies, such as using cost-sensitive learning alongside ensemble methods or resampling techniques, to create a more robust model.