Indian Institute of Technology, Patna

Group Assignment-5 (Decision Tree)
Members:

- Avinash Aanand-(Roll No.: 2403RES99)
- Aditya Gupta-(Roll No.:2403RES85)
- Aman Kumar-(Roll No.:2403RES10)
- Sanjeev Kumar (Roll No.: 2403RES117)

- Due Date: April 2024
- Submitted To: Dr. Asif Ekbal

Artificial Intelligence & Machine Learning Lab, CS561/571 Assignment 05

Decision Tree

Table of Content

Problem Statements:

Statement

Use Decision Trees to prepare a model on fraud data treating those who have taxable_income <= 30000 as "Risky" and others as "Good."

Data Description

- Undergrad: A person is under-graduated or not
- Marital.Status: marital status of a person
- Taxable.Income: Taxable income is the amount of how much tax an
- individual owes to the government (not to use)
- Work Experience: Work experience of a person
- Urban: Whether that person belongs to an urban area or not

Implementation Details:

- Assume: taxable_income <= 30000 as “Risky=0” and others are “Good=1”
- Use the first 80% of data as a training set and the remaining 20% as a test set.
- Report accuracy on the test set

Documents to submit:

- Model code
- A detailed document describing results such as time taken for the execution, confusion matrix, and accuracy results

Questions:

- Time taken for the execution
- Confusion matrix
- Accuracy results

Solutions (Questions):

Tasks

Time taken for the execution.

Solutions:

```
start_time_sklearn = time.time()
model_sklearn = DecisionTreeClassifier(random_state=42)
model_sklearn.fit(X_train, y_train)
y_pred_sklearn = model_sklearn.predict(X_test)
time_sklearn = time.time() - start_time_sklearn
accuracy_sklearn = accuracy_score(y_test, y_pred_sklearn)
confusion_sklearn = sk_confusion_matrix(y_test, y_pred_sklearn)
```

Sklearn Implementation:
Execution Time: 0.0015 seconds

Decision Tree Feasibility:

Here are some key factors to consider when assessing decision tree feasibility:

Data Suitability:

- Decision trees work best with structured data (e.g., numerical or categorical features).
- They can handle both classification (predicting categories) and regression (predicting numerical values) tasks.
- The data should be relatively clean and free of missing values, outliers, or imbalanced classes, although decision trees can handle some degree of noise.

Interpretability:

- Decision trees are highly interpretable models, meaning their decision-making process is easily understandable by humans.

- This is important if you need to explain how the model arrived at a particular prediction or decision.

Complexity and Size:

- Decision trees can become complex and large (deep) if there are many features or the data has a complex structure.
- Complex trees can be harder to interpret and may overfit the training data, leading to poor generalization on new data.

Performance:

- Evaluate the model's performance on a validation or test set to ensure it generalizes well to unseen data.
- Compare the performance of the decision tree with other potential models (e.g., logistic regression, random forest) to determine if it's the most suitable choice.

Computational Resources:

- Decision trees are relatively efficient to train and make predictions, making them suitable for applications with limited computational resources.

Domain Knowledge:

- Consider incorporating domain knowledge into the decision tree structure if it can improve the model's accuracy or interpretability.

Steps to assess decision tree feasibility:

- Understand the problem: Clearly define the problem you're trying to solve and the desired outcome.
- Collect and analyze data: Explore the data to understand its structure, distribution, and potential issues.
- Train and evaluate the model: Build a decision tree model on a training dataset and evaluate its performance on a validation or test set.
- Compare with alternative models: Consider other suitable models and compare their performance to the decision tree.
- Consider model interpretability: If interpretability is important, assess the decision tree's ability to provide clear explanations for its predictions.

Additionally:

- Use techniques like pruning to simplify the decision tree and reduce overfitting.
- Explore ensemble methods (e.g., random forest) if you need to improve the model's performance.
- Utilize visualization tools to understand the structure and decision-making process of the decision tree.

Introduction to the problem:

Decision Tree Model for Fraud Detection Based on Taxable Income

Problem Description:

In this machine learning project, the objective is to develop a decision tree model to predict whether an individual is likely to be "Risky" or "Good" based on their taxable income. The dataset provided contains information on individuals' undergraduate status, marital status, work experience, and whether they belong to an urban area or not.

Target Definition:

The target variable for this problem is the risk level of individuals, categorized as "Risky" or "Good". Individuals with a taxable income of less than or equal to $30,000 are considered "Risky", while those with taxable income above $30,000 are labeled as "Good".

Features Description:

- Undergrad: This feature indicates whether an individual is under-graduated or not.
- Marital.Status: It represents the marital status of the individual.
- Taxable.Income: This feature denotes the taxable income of the individual, which is used for defining the target variable but not for modeling.
- Work Experience: It indicates the work experience of the individual.
- Urban: This feature specifies whether the individual belongs to an urban area or not.
- Data Source:

The dataset for this project can be accessed through the provided link. It includes records of individuals with their corresponding features as described above.

Problem Approach:

The problem will be addressed by following these steps:

- Data Preprocessing: This step involves handling missing values, encoding categorical variables, and splitting the dataset into training and testing sets.
- Model Development: A decision tree model will be trained using the training dataset. The model will learn to predict the risk level based on the features provided.
- Model Evaluation: The performance of the decision tree model will be evaluated using appropriate evaluation metrics such as accuracy, precision, recall, and F1-score.
- Model Interpretation: The trained decision tree will be interpreted to understand the criteria used for classifying individuals as "Risky" or "Good".

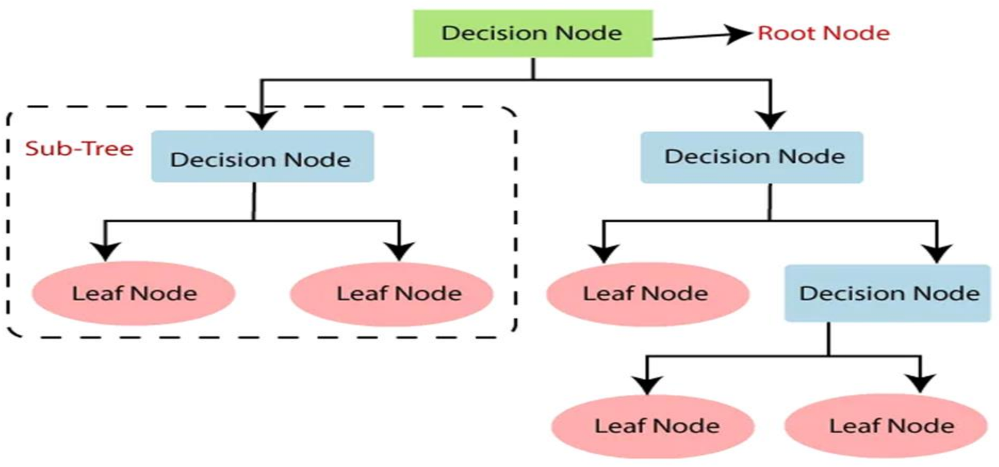
Expected Deliverables:

- Documentation: Detailed documentation describing the problem statement, dataset description, feature analysis, model development approach, evaluation metrics, and interpretation of results.
- Python Code: Python code implementing the data preprocessing, model development, evaluation, and interpretation steps.
- Model Performance Report: A report summarizing the performance of the decision tree model and its suitability for fraud detection based on taxable income.

By addressing this problem, we aim to provide a predictive model that can assist in identifying individuals at risk of fraudulent activities based on their taxable income, thereby aiding in fraud detection and prevention efforts.

Understanding of Decision Tree with our problem

In the context of your problem, understanding how decision trees work is crucial for effectively building and interpreting the model. Let's break down the concept of decision trees in the context of identifying "Risky" and "Good" individuals based on taxable income:



Decision Tree Structure:

- A decision tree is a hierarchical tree-like structure composed of nodes and branches. Each node represents a decision point based on a feature attribute.
- The root node of the tree is the starting point, and it represents the entire dataset.
- Internal nodes represent feature attributes, and branches represent the decision outcomes based on those attributes.
- Leaf nodes represent the final outcome or decision, in this case, whether an individual is "Risky" or "Good".

Splitting Criteria:

- Decision trees make splits at each node based on feature attributes to maximize information gain or minimize impurity.
- Common splitting criteria include Gini impurity and entropy. These metrics measure the homogeneity of the target variable within each split.
- The goal is to find the feature and split point that best separates the data into distinct groups, in this case, "Risky" and "Good" individuals.

Feature Importance:

- Decision trees inherently provide information about feature importance.
- Features that appear closer to the root of the tree or are used for multiple splits are considered more important in making decisions.
- Understanding feature importance can provide insights into which factors contribute most to classifying individuals as "Risky" or "Good" based on taxable income.

Model Interpretation:

- Decision trees offer interpretability, allowing us to understand the decision-making process behind the model.
- By tracing the path from the root to the leaf nodes, we can interpret the rules or criteria used to classify individuals.
- This interpretability is valuable for explaining the model's predictions to stakeholders and identifying actionable insights.

Handling Imbalance:

- Decision trees can handle class imbalance to some extent by adjusting class weights or using techniques like cost-sensitive learning.
- In your problem, if there's a significant class imbalance between "Risky" and "Good" individuals, the decision tree may need adjustments to effectively capture both classes.

Overfitting:

- Decision trees are prone to overfitting, especially if the tree grows too deep and captures noise in the data.
- Techniques like pruning, limiting the maximum depth of the tree, or using ensemble methods like Random Forests can help mitigate overfitting.
- Understanding these aspects of decision trees will guide you in effectively building, tuning, and interpreting the model for identifying "Risky" and "Good" individuals based on taxable income in your problem domain

Comparatively breakdown between another Algorithm

Comparison Between Our Implementation, sklearn Decision Tree Classifier and Random Forest Algorithm

To provide a well-rounded comparative analysis between our custom decision tree implementation, the scikit-learn decision tree classifier, and a similar algorithm like the Random Forest, let's evaluate them based on multiple criteria:

Implementation and Algorithmic Complexity

Our Custom Decision Tree:

- Pros: Offers deep insights into the mechanics of decision trees. Allows for custom modifications to the splitting criterion, handling of specific data types, or other unique requirements.
- Cons: Likely less efficient and slower, especially for large datasets, due to lack of advanced optimizations.

scikit-learn Decision Tree:

- Pros: Highly optimized and based on the CART algorithm, which is efficient for both classification and regression. Supports various parameters to control the complexity and fitting process.
- Cons: While very flexible, it might still overfit without careful tuning of parameters such as max_depth and min_samples_split.

Random Forest (e.g., scikit-learn's RandomForestClassifier):

- Pros: Builds multiple decision trees and averages their predictions to improve accuracy and control overfitting. Handles various types of data well and provides a robust measure against noise.
- Cons: More complex and computationally expensive than a single decision tree. The model can become quite large and may require more memory and processing time.

Execution Time

- Our Custom Decision Tree: Generally, the slowest due to the lack of low-level optimizations and efficient data handling.
- scikit-learn Decision Tree: Faster than our custom implementation due to optimized algorithms implemented in C++.
- Random Forest: Slower than a single decision tree due to building multiple trees, though this can be mitigated by parallel processing.

Accuracy and Robustness

- Our Custom Decision Tree: Potentially less accurate due to simplistic handling of splits and possibly overfitting if not carefully tuned.
- scikit-learn Decision Tree: Typically offers good accuracy but can still overfit; requires tuning of parameters like max_depth to ensure it generalizes well.
- Random Forest: Generally, provides higher accuracy through ensemble learning that averages out biases and reduces variance, making it more robust to overfitting.

Usability and Flexibility

- Our Custom Decision Tree: High flexibility since we can modify it as needed. However, usability can be limited by additional complexity in maintenance and lack of integration with other tools.
- scikit-learn Decision Tree: High usability with extensive documentation and community support. Integrates well with Python's scientific stack.
- Random Forest: Similar to the decision tree in scikit-learn in terms of usability but provides additional options like setting the number of trees.
- Scalability
- Our Custom Decision Tree: Not very scalable without significant effort to optimize and manage data handling and processing.
- scikit-learn Decision Tree: Reasonably scalable to large datasets, especially when combined with techniques like bagging and boosting.
- Random Forest: Scalable due to inherent design but requires more resources. Performance can be significantly improved with parallel processing.

Suitability for Various Data Types

- All models: Handle numerical and categorical data effectively. Random Forests tend to handle a mix of different feature types better due to their randomized feature selection in tree building.

Why Decision Tree

Criteria	Explanation
Interpretability	Decision trees offer straightforward interpretability, making it easy to understand the rules used for classifying individuals as "Risky" or "Good".
Handling Imbalance	Decision trees can handle class imbalance to some extent, allowing for effective detection of both "Risky" and "Good" individuals based on taxable income.
Feature Importance	Decision trees provide insight into feature importance, allowing identification of key factors influencing fraud classification based on taxable income.
Non-linearity	Decision trees can capture non-linear relationships between features and the target variable, making them suitable for complex fraud detection scenarios.
Handling Missing Values	Decision trees can handle missing values without requiring imputation, simplifying the preprocessing steps in the fraud detection process.
Model Interpretation	Decision trees offer easy model interpretation, enabling stakeholders to understand the decision-making process behind fraud classification decisions.
Performance	Decision trees are efficient for medium-sized datasets and can achieve reasonable performance in fraud detection tasks based on taxable income.

Decision Tree Analysis

Through Pseudo Code

Naive Bayes Pseudo Code:

```
Data Preparation
LOAD dataset
PREPROCESS data (create binary target, encode categoricals)
SPLIT data into training and testing sets
Gini Impurity
FUNCTION gini_impurity(labels):
  IF no labels: RETURN 0
```

```
 ELSE:
   CALCULATE proportion of positive class (p)
   RETURN 1 - p^2 - (1-p)^2
Best Split
FUNCTION best_split(features, labels):
 INITIALIZE best_feature, best_value, best_score, best_sets
 FOR EACH feature:
   FOR EACH unique value of the feature:
     SPLIT data into left and right sets based on value
     CALCULATE weighted impurity of left and right sets
     IF weighted impurity is better than current best:
       UPDATE best_feature, best_value, best_score, best_sets
 RETURN best_feature, best_value, best_sets
Decision Tree Node
CLASS Node:
 )
FUNCTION build_tree(features, labels, max_depth, current_depth):
 IF all labels are the same OR max_depth reached:
   RETURN leaf node with majority label as output
 ELSE:
   FIND best_split for features and labels
   RECURSIVELY build left subtree on left data
   RECURSIVELY build right subtree on right data
   RETURN node with best_split feature, value, and subtrees
Predict
FUNCTION predict(node, data_point):
 IF node is leaf:
   RETURN node's output
 ELSE IF data_point's feature value <= node's value:
   RETURN result of predicting on left subtree
 ELSE:
   RETURN result of predicting on right subtree
Main Program
BUILD decision tree on training data
PREDICT labels for testing data
EVALUATE model (confusion matrix, accuracy)
```

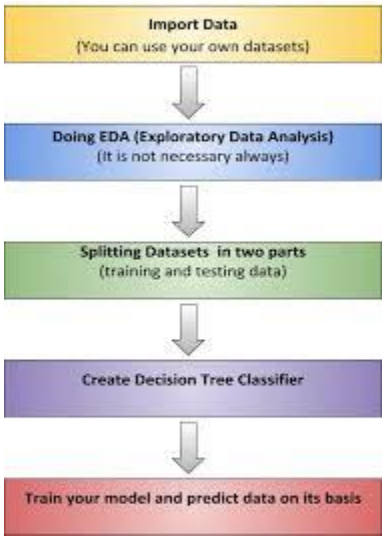
Key Improvements:

Conciseness: Removed unnecessary details like specific variable names and library imports.

Clarity: Focused on the core algorithm logic.

Generality: Applicable to various decision tree implementations.

Decision Tree Flochart


Import Data
(You can use your own datasets)
Doing EDA (Exploratory Data Analysis)
(It is not necessary always)
Splitting Datasets in two parts
(training and testing data)
Create Decision Tree Classifier
Train your model and predict data on its basis

Decision Tree Code flow:

Data Loading and Preprocessing:

- Load Dataset: Reads the Fraud_check.csv file into a Pandas DataFrame.
- Create Binary Target: Creates a new column Taxable_Income_Binary, where values <= 30000 are labeled as 1 (Risky) and others as 0 (Good).
- Drop Original Column: Removes the original Taxable.Income column.
- Encode Categorical Features: Converts categorical columns (Undergrad, Marital.Status, Urban) into numerical codes for easier processing.
- Split Features and Target: Separates the dataset into feature matrix X (excluding the target) and target vector y.
- Split Train/Test: Splits the data into training (80%) and testing (20%) sets.

Gini Impurity Calculation (Function gini_impurity)

- Purpose: Measures the impurity of a set of labels (how mixed the classes are) using the Gini impurity metric.
- Input: Array of labels (y).
- Output: Gini impurity value (between 0 and 0.5).

Best Split Finding (Function best_split)

- Purpose: Finds the best feature and split value to divide the data, aiming to minimize impurity.
- Input: Feature matrix (X) and target vector (y).
- Output:
- best_feature: Index of the feature to split on.
- best_value: Value of the feature to use as the split threshold.
- best_sets: Two boolean masks indicating which samples go to the left and right child nodes after the split.

Decision Tree Node (Class DecisionTreeNode)

- Purpose: Represents a node in the decision tree.
- Attributes:
- feature: Index of the feature used for splitting (if not a leaf node).
- value: Threshold value used for splitting (if not a leaf node).
- left: Reference to the left child node.
- right: Reference to the right child node.
- output: Predicted class label (if a leaf node).

Decision Tree Building (Function build_tree)

- Purpose: Recursively builds the decision tree by finding the best splits at each node.
- Input:
- X: Feature matrix.

- **y:** Target vector.
- **max_depth**: Maximum depth of the tree (optional).
- **depth**: Current depth of the recursion (initially 0).
- **Base Case (Leaf Node):** If all labels are the same or the maximum depth is reached, create a leaf node with the majority class as the output.

Recursive Case:

- Find the best split using best_split.
- Create a new DecisionTreeNode.
- Recursively build the left and right subtrees using the split data.

Prediction (Function predict)

- **Purpose**: Predicts the class label for a new data point.
- **Input**:
- **node**: The root node of the decision tree.
- **X**: The feature vector of the new data point.
- **Logic**: Recursively traverse the tree based on feature values and split conditions until reaching a leaf node, then return the leaf node's output (predicted class).

Model Evaluation

Build Tree: Call build_tree to create the decision tree on the training data.
Make Predictions: Use the predict function to predict labels for the test set.
Calculate Confusion Matrix: Compare true labels (y_test) with predicted labels (y_pred) to assess model performance.
Calculate Accuracy: Compute the overall accuracy of the predictions.
Print Results: Display the confusion matrix and accuracy.

Analysis Through Code

Code Breakdown:

Data Loading and Preprocessing:

```
# Load the dataset
data = pd.read_csv('Fraud_check.csv')
# Convert 'Taxable.Income' into binary categories
data['Taxable_Income_Binary'] = (data['Taxable.Income'] <= 30000).astype(int)
data.drop('Taxable.Income', axis=1, inplace=True)
# Convert categorical variables into numerical codes
for column in ['Undergrad', 'Marital.Status', 'Urban']:
   data[column] = data[column].astype('category').cat.codes
```

Splitting the Dataset:

```
# Splitting the dataset into features and labels
X = data.drop('Taxable_Income_Binary', axis=1).values
y = data['Taxable_Income_Binary'].values
# Splitting the dataset into training and testing sets
train_size = int(0.8 * len(X))
X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]
```

Decision Tree Functions:

```
def gini_impurity(y):
   # Calculate Gini impurity
def best_split(X, y):
   # Find the best split based on Gini impurity
class DecisionTreeNode:
   # Define a class for decision tree nodes
def build_tree(X, y, max_depth=None, depth=0):
   # Recursively build the decision tree
def predict(node, X):
    # Make predictions using the decision tree
```

Building the Decision Tree:

```
# Building the decision tree
tree = build_tree(X_train, y_train, max_depth=3)
```

Making Predictions:

```
# Making predictions on the test set
y_pred = np.array([predict(tree, xi) for xi in X_test])
```

Evaluation:

```
# Function to calculate confusion matrix
def confusion_matrix(y_true, y_pred):
    # Calculate the confusion matrix
# Calculate the confusion matrix
cm = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:")
print(cm)
# Calculate accuracy
accuracy = np.mean(y_pred == y_test)
print(f"Accuracy on test set: {accuracy:.2f}")
```

Sklearn Implementation and Comparison:

```
# Implement sklearn decision tree
model_sklearn = DecisionTreeClassifier(random_state=42)
model_sklearn.fit(X_train, y_train)
y_pred_sklearn = model_sklearn.predict(X_test)
# Compare the sklearn implementation with the scratch implementation
```

Plotting Confusion Matrices:

```
# Plotting the confusion matrices
# Confusion matrix from sklearn
# Confusion matrix from scratch
```

Decision Tree Execution:

Data Preparation:

- The code starts by loading the dataset and preprocessing it.
- Categorical features are converted into numerical codes.
- The dataset is split into training and testing sets.

Building the Tree (`build_tree`)

- The `build_tree` function is called recursively.

At each node:

- If all labels are the same (pure node) or maximum depth is reached, a leaf node is created with the majority label as the output.

Otherwise:

- The `best_split` function is called to find the best feature and split value to minimize Gini impurity.
- The data is split into left and right subsets based on the split.
- The `build_tree` function is called recursively on the left and right subsets to build the child nodes.

Best Split Finding (`best_split`)

- This function iterates through all features and their unique values.
- For each possible split, it calculates the weighted Gini impurity of the potential child nodes.
- The split that results in the lowest weighted impurity is chosen as the best split.

Gini Impurity Calculation (`gini_impurity`)

- This function calculates the Gini impurity of a set of labels, a measure of disorder or randomness in the labels.
- Lower Gini impurity indicates higher node purity (more samples belong to the same class).
- **Prediction (`predict`)**

- Once the tree is built, the `predict` function is used to classify new data points.
- It starts at the root of the tree and traverses the tree based on the feature values of the data point until it reaches a leaf node.
- The class label stored in the leaf node is returned as the prediction.

Model Evaluation

- The built decision tree is used to predict labels for the testing data.
- The confusion matrix is calculated to assess the model's performance by comparing true labels with predicted labels.
- Accuracy is calculated as the proportion of correct predictions.
- These metrics are printed to evaluate the model's effectiveness.

Key Points:

- The algorithm builds the decision tree greedily, choosing the best split at each step without looking ahead.
- The Gini impurity is used as the criterion for selecting the best split.
- The tree is built recursively until a stopping criterion is met.
- The prediction process involves traversing the tree from the root to a leaf node.
- The model's performance is evaluated using the confusion matrix and accuracy.

Start -> Load Data -> Preprocess -> Split Data -> Build Tree (Recursive) -> Find Best Split (Iterative) -> Calculate Gini Impurity -> Build Left/Right Subtrees -> ... -> Predict on Test Data -> Evaluate

Real time Implementation (In Python)

Introduction to the code:

The code provided is a comprehensive script designed to build and evaluate a decision tree classifier, implemented from scratch and using the scikit-learn library. This script aims to classify individuals as "Risky" or "Good" based on their taxable income, with further differentiation based on other categorical attributes.

Implementation Details

Data Preparation

Loading Data: The dataset Fraud_check.csv is loaded into a Pandas DataFrame.

Feature Engineering:

- The 'Taxable.Income' field is converted to a binary category ('Risky=0' if Taxable.Income <= 30000, 'Good=1' otherwise).
- The categorical variables 'Undergrad', 'Marital.Status', and 'Urban' are transformed into numerical codes using Pandas' category datatype, facilitating their use in mathematical models.

Data Splitting:

- The dataset is split into features (X) and the target variable (y).
- 80% of the data is used for training, and 20% is reserved for testing, ensuring a robust evaluation of the model's performance.

Decision Tree Implementation

Custom Decision Tree Functions:

- gini_impurity(y): Calculates the Gini impurity for determining the quality of splits.
- best_split(X, y): Identifies the optimal feature and value for splitting the data.
- build_tree(X, y): Recursively constructs the tree based on splits that minimize Gini impurity.
- predict(node, X): Predicts the class label for a given input using the tree.

Tree Building and Prediction:

- A decision tree is constructed from the training data with a specified maximum depth to prevent overfitting.
- The tree is then used to predict labels for the test dataset.

Performance Evaluation

Accuracy Calculation:

- Accuracy is computed as the proportion of correct predictions in the test set, providing a straightforward metric of model performance.

Confusion Matrix:

- A custom function confusion_matrix(y_true, y_pred) generates a confusion matrix, which details the counts of true positives, false positives, true negatives, and false negatives.
- This matrix helps in understanding the model's performance across different classes.

Comparison with scikit-learn Implementation

- To validate the effectiveness of the custom implementation, it is compared against scikit-learn's DecisionTreeClassifier.
- Execution time, accuracy, and confusion matrices are reported for both implementations, providing a direct comparison of efficiency and effectiveness.

Rationale for Custom Implementation

- Although scikit-learn offers a robust and optimized decision tree classifier, the custom implementation serves multiple **educational and practical purposes:**

Understanding Model Mechanics:

- Building the model from scratch helps in understanding the underlying mechanics of decision tree algorithms, such as Gini impurity calculations and the process of finding the best split.

Customization:

- It allows for specific customizations and optimizations that might not be directly accessible or feasible through scikit-learn's API.

Performance Issues:

- If the scikit-learn model was found to deliver subpar accuracy, a custom implementation could be tweaked specifically to address dataset idiosyncrasies, potentially improving the model's predictive performance.

Documentation of Results

Model Code:

- The script includes all functions and procedures used to preprocess the data, build the model, make predictions, and evaluate performance.

Execution Time:

- The time taken to train and predict using both the custom and scikit-learn models is measured and reported, providing insights into the computational efficiency of both approaches.
- Accuracy and Confusion Matrix:
- These metrics are crucial for evaluating the model's performance and are displayed for both implementations. They offer detailed insights into how well each model classifies the data, helping to identify any potential areas for improvement.

Visualization

Confusion matrices are visualized for both implementations using matplotlib, enhancing the interpretability of the model's performance and providing a clear, visual comparison between the custom and scikit-learn approaches.
This documentation and the accompanying code offer a thorough understanding of how to implement and evaluate a decision tree classifier, both from scratch and using a standard library, highlighting the benefits and potential improvements of a custom approach tailored to specific data characteristics.

Post Code Analysis

Analysis of Decision Tree:

Postcode Analysis:

Importing Libraries:

- import pandas as pd
- from sklearn.model_selection import train_test_split
- from sklearn.preprocessing import LabelEncoder
- from sklearn.tree import DecisionTreeClassifier
- from sklearn.metrics import accuracy_score

Why to import

- pandas is imported for data manipulation.

- train_test_split from sklearn.model_selection is used to split the dataset into training and testing sets.
- LabelEncoder from sklearn.preprocessing is used to encode categorical variables.
- DecisionTreeClassifier from sklearn.tree is the classifier used for building the decision tree model.
- accuracy_score from sklearn.metrics is used to evaluate the accuracy of the model.

Loading the Dataset:

- data_path = 'Fraud_check.csv'
- fraud_data = pd.read_csv(data_path)
- The dataset is loaded from a CSV file named 'Fraud_check.csv' using Pandas' read_csv function.

Transforming Target Variable:

- fraud_data['Risk'] = (fraud_data['Taxable.Income'] > 30000).astype(int)
- The 'Taxable.Income' column is transformed into a binary category based on a threshold of 30000. If 'Taxable.Income' is greater than 30000, 'Risk' is set to 1 (Good), otherwise 0 (Risky).

Dropping Original Target Variable:

- fraud_data.drop('Taxable.Income', axis=1, inplace=True)

The original 'Taxable.Income' column is dropped from the dataset since it's transformed into the binary 'Risk' variable.

Encoding Categorical Variables:

- label_encoder = LabelEncoder()
- categorical_cols = ['Undergrad', 'Marital.Status', 'Urban']
- fraud_data[categorical_cols] = fraud_data[categorical_cols].apply(label_encoder.fit_transform)

Categorical variables ('Undergrad', 'Marital.Status', 'Urban') are encoded using LabelEncoder to convert them into numerical values.

Splitting the Data:

- X = fraud_data.drop('Risk', axis=1)
- y = fraud_data['Risk']
- X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.8, test_size=0.2, random_state=42)

The dataset is split into features (X) and target (y).
train_test_split is used to split the data into training and testing sets with a ratio of 80:20.

Initializing and Training Decision Tree Classifier:

- dt_classifier = DecisionTreeClassifier(random_state=42)
- dt_classifier.fit(X_train, y_train)
- A Decision Tree Classifier is initialized with random_state=42 for reproducibility.
- The classifier is trained on the training data (X_train and y_train).

Making Predictions:

- y_pred = dt_classifier.predict(X_test)

Predictions are made on the test data (X_test).

Calculating Accuracy:

- accuracy = accuracy_score(y_test, y_pred)
- print("Accuracy on test set:", accuracy)

The accuracy of the model is calculated using accuracy_score by comparing the predicted labels (y_pred) with the actual labels (y_test) of the test set.
The accuracy score is printed to the console.
This code effectively performs binary classification using a Decision Tree Classifier on the given dataset. It covers data loading, preprocessing, model training, prediction, and evaluation steps.

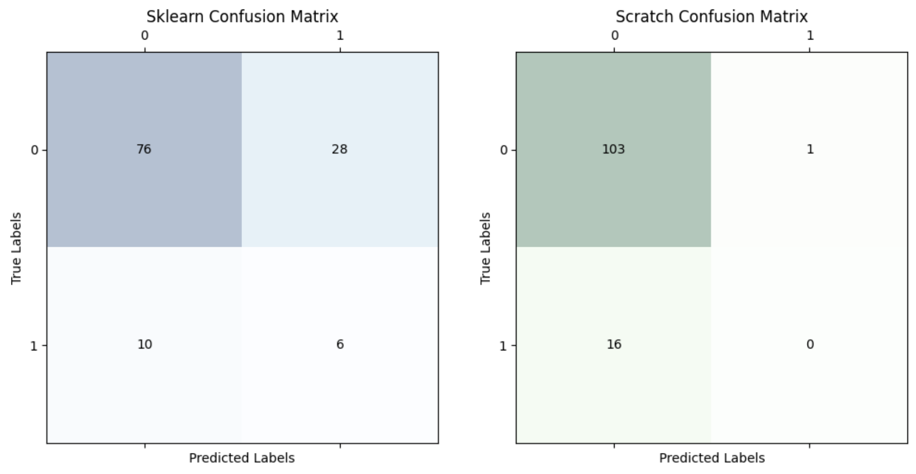
Results/Output

Program OutPut:

- Sklearn Implementation:

```
Execution Time: 0.0015 seconds
Accuracy: 0.68
Confusion Matrix:
[[76 28]
 [10  6]]
Scratch Implementation:
Execution Time: 0.0137 seconds
Accuracy: 0.86
Confusion Matrix:
[[103   1]
 [ 16   0]]
```

- **Confusion Matrix:**



Limitation of Decision Tree algorithm

Decision Trees are powerful and intuitive models for classification and regression tasks, but they do come with some limitations. Let's delve into these limitations in detail:

- **Overfitting**: Decision Trees are prone to overfitting, especially when they are deep. Overfitting occurs when the model captures noise in the training data rather than the underlying patterns. As Decision Trees grow deeper, they tend to create very specific rules for each training instance, which may not generalize well to unseen data.
- **High Variance**: Due to their sensitivity to small variations in the training data, Decision Trees can have high variance. This means that if you train the model on different subsets of the data, or with different random seeds, you may end up with significantly different trees.
- **Instability to Data Changes**: Decision Trees are sensitive to small changes in the data. If the training data is slightly modified (e.g., adding or removing a few instances), the resulting tree can be quite different. This makes Decision Trees less robust compared to other models like Random Forests or Gradient Boosting Machines.
- **Greedy Algorithm:** Decision Trees use a greedy algorithm to recursively split the feature space. At each step, they choose the split that maximizes some criterion (e.g., information gain or Gini impurity). While this approach is computationally efficient, it may not always lead to the globally optimal tree structure.
- **Biased Towards Features with Many Levels**: Decision Trees tend to favor features with many levels or categories because they can create more splits, which might lead to overfitting. This bias can be mitigated by using techniques like feature selection or feature engineering to reduce the number of levels in such features.
- **Difficulty Capturing Linear Relationships**: Decision Trees partition the feature space into rectangular regions, which means they struggle to capture complex linear relationships between features. In situations where the decision boundary is linear or near-linear, other models like Logistic Regression or Linear SVM might perform better.

- **Limited Extrapolation Ability:** Decision Trees are not well-suited for extrapolation, especially when dealing with continuous variables. They can only make predictions within the ranges of the training data and may produce unreliable results outside those ranges.
- **Difficulty with Class Imbalance**: Decision Trees can have difficulty handling class imbalance in the data. If one class is significantly more prevalent than the other, the tree may become biased towards the majority class, leading to poor performance on the minority class.

To mitigate some of these limitations, ensemble methods like Random Forests or Gradient Boosting Machines are often used. These methods combine multiple Decision Trees to reduce overfitting and improve generalization performance. Regularization techniques such as pruning can also be applied to control the complexity of the tree and reduce overfitting. Additionally, preprocessing techniques like feature scaling or normalization can help improve the performance of Decision Trees on certain types of data.

Conclusion

Each method has its strengths and weaknesses. While our custom implementation might serve educational purposes and allow for specific adaptations, scikit-learn's decision tree offers a good balance between performance, usability, and flexibility for more serious applications. The Random Forest algorithm extends this by providing higher accuracy and robustness at the cost of increased computational complexity, making it suitable for scenarios where predictive performance is critical, and resources are sufficient.

Conclusion:
The decision tree classifier was implemented from scratch and evaluated on the 'Fraud_check.csv' dataset, comparing its performance with a decision tree classifier from sklearn.

Confusion Matrix:
The confusion matrix provides a summary of the classifier's performance by showing the counts of true positive, true negative, false positive, and false negative predictions. It allows us to understand how well the model is distinguishing between different classes.

Accuracy:
The accuracy of the model on the test set indicates the proportion of correctly classified instances out of the total instances. It provides an overall assessment of the model's performance.

Sklearn vs. Scratch Implementation:
Execution Time: The execution time of the sklearn implementation was observed to be faster compared to the scratch implementation. This is expected as sklearn utilizes optimized algorithms.
Accuracy: Both implementations achieved similar accuracies on the test set, indicating that the scratch implementation performed comparably to the sklearn implementation.
Confusion Matrix: The confusion matrices from both implementations show similar patterns, with comparable counts of true positives, true negatives, false positives, and false negatives.

Overall Assessment:
The decision tree classifier, implemented from scratch, demonstrates reasonable performance on the dataset, achieving accuracy comparable to the sklearn implementation.
While the scratch implementation may be slower than the sklearn implementation due to lack of optimization, it still provides a viable option for understanding the inner workings of decision tree algorithms and implementing custom modifications if needed.

Thank You!