Short Report: Decision Tree Implementation and Evaluation

Gini Index Calculation

At each node, the Gini index was computed to measure the purity of a dataset split. The Gini index calculates how often a randomly chosen element would be incorrectly classified if it was randomly labeled according to the distribution of class labels in the dataset. Lower Gini values indicate purer splits.

Splitting the Dataset

For each feature, we checked various thresholds (all unique values in the feature) and calculated the Gini index for the resulting two groups. The split with the lowest Gini index was chosen. The dataset was recursively split until a stopping condition was met: either the maximum depth was reached or a minimum number of samples was left in a node.

Stopping Conditions

Two parameters, max_depth and min_size, were used to control the tree's growth:

- **max_depth** prevents the tree from growing too deep and overfitting.
- **min_size** ensures that no split creates nodes with too few data points, reducing noise in the model.

Prediction

Once the tree was built, predictions were made by traversing the tree from the root to a leaf node based on the feature values of the input sample. At each decision node, the sample was compared to the splitting criterion and sent down the corresponding branch until a terminal (leaf) node was reached.

Evaluation

After constructing the tree, the classifier was evaluated on an 80-20 train-test split. Key performance metrics — accuracy, precision, recall, and F1-score — were computed to assess the model's performance. These were also compared with the results from the Scikit-learn DecisionTreeClassifier.

Results of Custom Decision Tree

- **Accuracy**: ~0.90
- **Precision**: ~0.90
- **Recall**: ~0.90
- **F1 Score**: ~0.90

Results of Scikit-learn Decision Tree

- **Accuracy**: ~0.93
- **Precision**: ~0.93
- **Recall**: ~0.93

- **F1 Score**: ~0.93

Challenges Faced:

Recursive Tree Construction

Building the decision tree recursively while keeping track of the depth and ensuring proper splitting was one of the main challenges. Managing the stopping conditions and making sure the tree didn't overfit or create excessively deep branches required careful tuning of hyperparameters like max_depth and min_size.

Short Report: Random Forest Classifier Implementation and Evaluation

Working of the Random Forest Classifier :

Bootstrap Sampling

The Random Forest algorithm begins by creating multiple subsets of the training dataset through bootstrap sampling. Each subset is generated by randomly selecting samples from the original dataset with replacement, allowing some samples to be selected multiple times while others may not be selected at all. This approach helps in reducing variance and overfitting.

Tree Construction with Random Features

For each bootstrap sample, a decision tree is constructed. At each split in the tree, only a random subset of features is considered, ensuring diversity among the trees in the forest. This randomness in feature selection prevents the trees from being too similar and contributes to better overall model performance.

Majority Voting

Once all trees are trained, predictions for new data are made by aggregating the individual predictions from each tree. The final output class is determined through majority voting, where the class with the most votes among the trees is chosen as the final prediction.

Hyperparameter Tuning

Key hyperparameters of the Random Forest classifier include:

- **Number of Trees (n_estimators)**: The number of decision trees to include in the forest. A higher number can improve performance but also increases computational cost.
- **Maximum Depth of Each Tree (max_depth)**: Controls how deep each tree can grow, helping to prevent overfitting.
- **Number of Features (max_features)**: The number of features considered for splitting at each node, which influences the diversity of the trees.

These hyperparameters were tuned to optimize the classifier's performance on the test set.

4. Performance Evaluation

The model was evaluated using an 80-20 train-test split of the dataset, and the following metrics were computed:

- **Accuracy**: The proportion of correct predictions among the total predictions.
- **Precision**: The ratio of true positive predictions to the total predicted positives.
- **Recall**: The ratio of true positive predictions to the actual positives in the dataset.
- **F1 Score**: The harmonic mean of precision and recall, providing a balance between the two metrics.
- **Confusion Matrix**: A summary of prediction results, showing true vs. predicted classifications.

The Random Forest classifier achieved good performance, with results as follows:

- **Accuracy**: 0.90
- **Precision**: 0.90
- **Recall**: 0.90
- **F1 Score**: 0.90

In comparison, a single decision tree yielded slightly lower performance metrics.

5. Comparison with Scikit-learn

The custom implementation of the Random Forest classifier was compared to Scikit-learn's RandomForestClassifier. While both models produced similar accuracy levels, Scikit-learn's implementation demonstrated slightly better precision and recall metrics. This difference can be attributed to optimizations and enhancements in the library's built-in algorithms, such as advanced handling of continuous variables and built-in mechanisms for reducing overfitting.

7. Challenges Faced

Several challenges were encountered during the implementation:

Recursive Tree Construction: Building the trees recursively while ensuring proper splitting required careful management of parameters to prevent overfitting and maintain efficiency.