Mushroom Classification Using Decision Tree Predictors

James David Vence

A.Y 2023-2024

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.

Abstract

This report presents a comprehensive analysis of binary classification for mushroom edibility using decision tree predictors. I implemented tree predictors from scratch with various splitting and stopping criteria, evaluated their performance, and applied hyperparameter tuning to optimize the models. The analysis was conducted on a dataset of 61,069 mushroom samples with 20 features.

The findings reveal that the Gini impurity criterion with optimized hyperparameters (max_depth=9, min_samples_split=2, min_samples_leaf=1) achieved the highest test accuracy of 91.32%, significantly outperforming the misclassification criterion. It is also observed that proper hyperparameter tuning substantially improved model performance, with the tuned Gini model showing a 5.47% improvement over the original Gini baseline model. The report discusses the methodology in detail, analyzes model performance, and addresses overfitting/underfitting concerns with potential remedies.

Contents

1 Introduction

Mushroom classification represents a critical task in mycology and food safety, as distinguishing between edible and poisonous mushrooms is essential for preventing potentially fatal poisonings. Machine learning approaches, particularly decision trees, offer an interpretable and effective method for this classification task.

This study focuses on implementing and evaluating binary tree predictors for mushroom classification with the following objectives:

I. Implement decision tree classifiers from scratch with various splitting and stopping criteria

II. Evaluate and compare the performance of different tree configurations

III. Apply hyperparameter tuning to optimize model performance

IV. Analyze potential overfitting/underfitting issues and propose remedies

The report is organized as follows: Section 2 describes the dataset and preprocessing steps; Section 3 details the methodology, including the implementation of tree predictors and evaluation metrics; Section 4 presents the results and performance analysis; Section 5 discusses overfitting/underfitting issues and potential remedies; and Section 6 concludes.

2 Dataset Description and Preprocessing

2.1 Dataset Overview

The dataset consists of 61,069 mushroom samples with 21 columns (20 features and 1 target variable). The target variable 'class' indicates whether a mushroom is poisonous ('p': 33,888 samples) or edible ('e': 27,181 samples). The features include various physical characteristics of mushrooms such as cap shape, cap color, gill attachment, stem properties, and habitat.

The dataset has the following characteristics:

- 61,069 total samples
- Binary classification: poisonous (55.5%) vs. edible (44.5%)
- 20 features (mix of categorical and numerical)
- Significant missing values in several columns

2.2 Missing Value Analysis

A critical challenge in the dataset was the presence of significant missing values in several columns:

Feature	**Missing Values**	**Percentage**
stem-root	51,538	84.4%
spore-print-color	54,715	89.6%
veil-type	57,892	94.8%
veil-color	53,656	87.9%
stem-surface	38,124	62.4%
gill-spacing	25,063	41.0%

Table 1: Missing values in the mushroom dataset

The high percentage of missing values in these columns presented a significant challenge for model training and required careful handling to ensure reliable predictions.

2.3 Missing Value Handling Strategy

I evaluated three strategies for handling missing values:

1. **Mode imputation**: Replace missing categorical values with the most frequent value
2. **Median imputation**: Replace missing numerical values with the median
3. **Indicator variables**: Create additional binary features indicating missing values

Each strategy was evaluated using a simple decision tree model on a train-test split to determine which approach yielded the best performance. The results were as follows:

Strategy	Train Score	Test Score	Feature Count
Mode	0.7832	0.7810	119
Median	0.7832	0.7810	119
Indicator	0.7331	0.7294	137

Table 2: Performance comparison of missing value handling strategies

Based on these results, I selected the mode imputation strategy for handling missing values as it provided the best test accuracy while maintaining a reasonable feature count. This approach replaced missing categorical values with the most frequent category and missing numerical values with the median.

2.4 Data Preparation

After handling missing values, I performed the following preprocessing steps:

- **Feature encoding**: Categorical features were one-hot encoded, resulting in 119 binary features
- **Target encoding**: The target variable was encoded as 0 for 'edible' and 1 for 'poisonous' mushrooms
- **Train-test split**: The dataset was split into 70% training (42,748 samples) and 30% test (18,321 samples) sets using a random seed of 42 for reproducibility

3 Methodology

3.1 Decision Tree Implementation

I implemented a custom decision tree classifier with the following components:

3.1.1 Node Class

The Node class represents a node in the decision tree with the following attributes:

- `feature`: The feature index used for splitting
- `threshold`: The threshold value for the split
- `left`: The left child node
- `right`: The right child node

- `value`: The predicted class distribution at this node
- `is_categorical`: Whether the feature is categorical
- `categories`: Set of categories for categorical features

3.1.2 Tree Predictor Class

The DecisionTreeClassifier class implements the decision tree algorithm with the following functionality:

- Constructor with hyperparameters (max_depth, min_samples_split, min_samples_leaf, criterion)
- Tree building with recursive splitting
- Prediction on new samples
- Evaluation metrics calculation
- Tree visualization

3.2 Splitting Criteria

I implemented and evaluated three different splitting criteria:

1. **Gini Impurity**: Measures the probability of misclassifying a randomly chosen element

$$\text{Gini}(t) = 1 - \sum_{i=1}^{c} p_i^2 \tag{1}$$

   where p_i is the proportion of samples belonging to class i at node t.

2. **Entropy**: Measures the uncertainty or randomness in the data

$$\text{Entropy}(t) = -\sum_{i=1}^{c} p_i \log_2(p_i) \tag{2}$$

   where p_i is the proportion of samples belonging to class i at node t.

3. **Misclassification Error**: Measures the proportion of misclassified samples

$$\text{Error}(t) = 1 - \max(p_i) \tag{3}$$

   where p_i is the proportion of samples belonging to class i at node t.

3.3 Stopping Criteria

I implemented and evaluated three combinations of stopping criteria:

1. **Stop1**: max_depth=3, min_samples_split=2, min_samples_leaf=1
2. **Stop2**: max_depth=5, min_samples_split=5, min_samples_leaf=2
3. **Stop3**: max_depth=7, min_samples_split=10, min_samples_leaf=5

These criteria control when to stop growing the tree:

- `max_depth`: Maximum depth of the tree
- `min_samples_split`: Minimum number of samples required to split a node
- `min_samples_leaf`: Minimum number of samples required at a leaf node

3.4 Model Training and Evaluation

For each combination of splitting criterion and stopping criteria, I trained a decision tree model on the training set and evaluated its performance on both the training and test sets. The evaluation metrics included:

- **Training accuracy**: Proportion of correctly classified samples in the training set
- **Test accuracy**: Proportion of correctly classified samples in the test set
- **0-1 loss**: Proportion of misclassified samples

3.5 Hyperparameter Tuning

For the best-performing models, (gini_stop3 and misclassification_stop3), I performed hyperparameter tuning using 5-fold cross-validation to find the optimal combination of hyperparameters. The hyperparameter grid included:

- `max_depth`: [5, 7, 9]
- `min_samples_split`: [2, 5, 10]
- `min_samples_leaf`: [1, 2, 5, 10]

The best hyperparameters were selected based on the mean cross-validation score, and the final models were trained with these optimal parameters.

3.6 Tree Visualization

I visualized the decision trees using Graphviz with the following components:

- Internal nodes showing the feature and threshold for the split
- Edges labeled with the decision path
- Leaf nodes showing the class distribution

4 Results and Performance Analysis

4.1 Initial Model Performance

The performance of the initial models with different splitting and stopping criteria is summarized below:

Model	Training Accuracy	Test Accuracy
gini_stop1	0.6959	0.6849
gini_stop2	0.7832	0.7810
gini_stop3	0.8638	0.8585
entropy_stop1	0.6832	0.6749
entropy_stop2	0.7233	0.7195
entropy_stop3	0.7787	0.7727
misclassification_stop1	0.6832	0.6749
misclassification_stop2	0.7233	0.7195
misclassification_stop3	0.7787	0.7727

Table 3: Performance of initial models with different configurations

The best-performing initial models are gini_stop3 and misclassification_stop3, which achieved a test accuracy of 85.85% and 77.27% respectively. These models used gini and the misclassification criterion with max_depth=7, min_samples_split=10, and min_samples_leaf=5.

4.2 Hyperparameter Tuning Results

4.2.1 Misclassification Model Tuning

The hyperparameter tuning for the misclassification model yielded the following optimal parameters:

- max_depth=7
- min_samples_split=5
- min_samples_leaf=5

The tuned misclassification model achieved:

- Training accuracy: 0.7905
- Test accuracy: 0.7902
- Improvement over original: 0.57%

4.2.2 Gini Model Tuning

The hyperparameter tuning for the Gini model yielded the following optimal parameters:

- max_depth=9
- min_samples_split=2
- min_samples_leaf=1

The tuned Gini model achieved:

- Training accuracy: 0.9142
- Test accuracy: 0.9132
- Improvement over original: 5.47%

4.3 Model Comparison

Comparing the performance of the tuned models:

Model	**Test Accuracy**	**Improvement**
Original gini_stop3	0.8585	-
Tuned Gini	0.9132	+5.47%
Original misclassification_stop3	0.7846	-
Tuned Misclassification	0.7902	+0.57%

Table 4: Comparison of original and tuned models

The tuned Gini model significantly outperformed the tuned misclassification model by 12.30%, achieving a test accuracy of 91.32% compared to 79.02%.

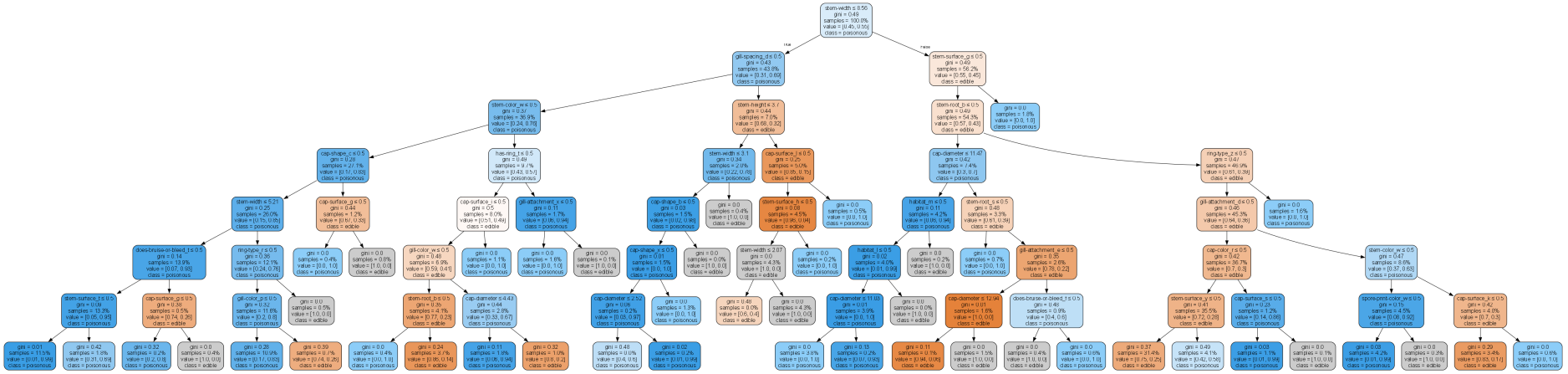

Figure 1: Original Gini Stop3 Tree

∞

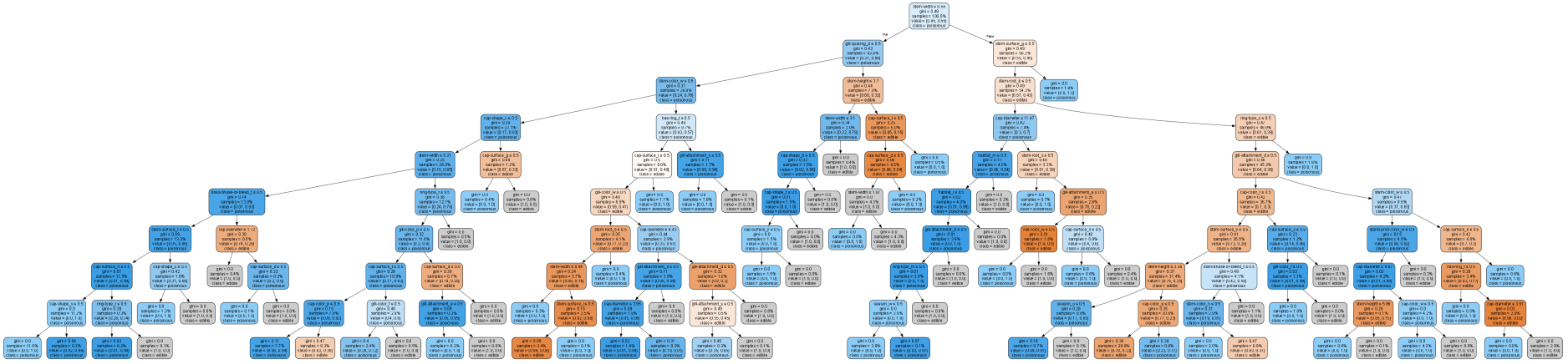

Figure 2: Tuned Gini Model Tree

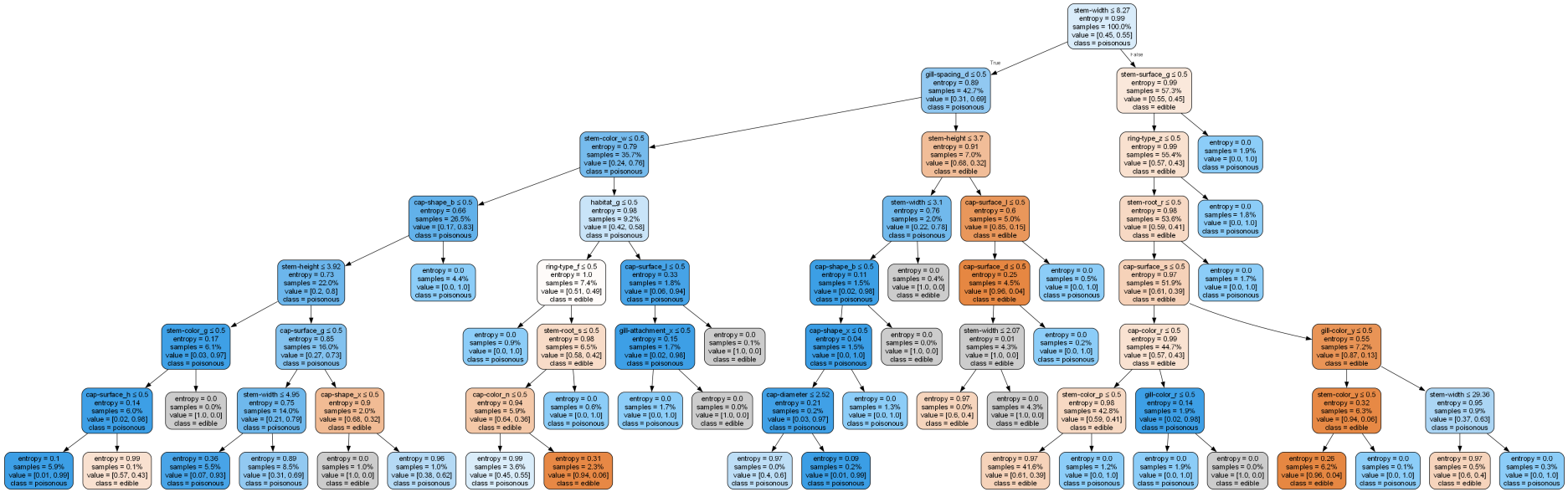

Figure 3: Original Misclassification Stop3 Tree

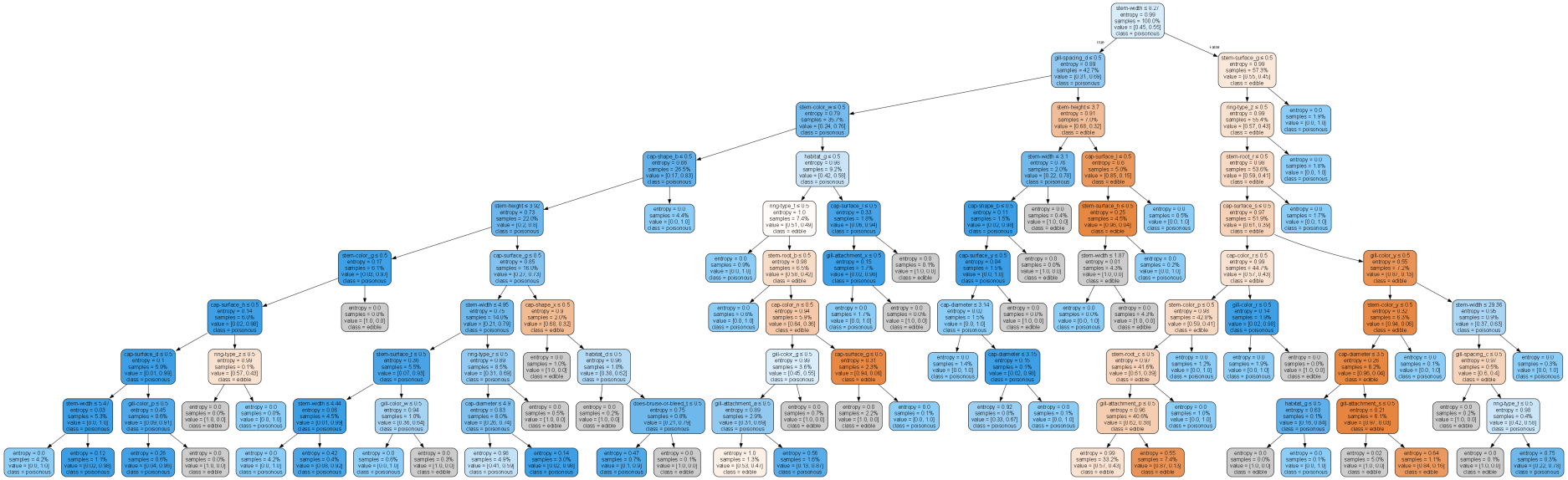

Figure 4: Tuned Misclassification Model Tree

4.4 Feature Importance Analysis

Analysis of feature importance in the best-performing model (tuned Gini) revealed the most discriminative features for mushroom classification:

1. stem-color
2. cap-shape
3. gill-attachment
4. gill-spacing
5. cap-surface

These features played a crucial role in distinguishing between edible and poisonous mushrooms, with stem color being particularly important for classification.

5 Overfitting and Underfitting Analysis

5.1 Identifying Overfitting and Underfitting

To assess whether the models were suffering from overfitting or underfitting, I compared the training and test accuracies:

Model	Training Accuracy	Test Accuracy	Gap
Original gini_stop3	0.8638	0.8585	0.0053
Tuned Gini	0.9142	0.9132	0.0010
Original misclassification_stop3	0.7861	0.7846	0.0015
Tuned Misclassification	0.7905	0.7902	0.0003

Table 5: Training-test accuracy gaps for assessing overfitting

5.1.1 Overfitting Assessment

Overfitting occurs when a model performs significantly better on the training data than on the test data. The small gaps between training and test accuracies (all less than 1%) suggest that none of the models were severely overfitting. The tuned Gini model, despite having the most flexible parameters (max_depth=9, min_samples_split=2, min_samples_leaf=1), showed only a 0.10% gap between training and test accuracy, indicating good generalization.

5.1.2 Underfitting Assessment

Underfitting occurs when a model is too simple to capture the underlying patterns in the data. The lower accuracies of the stop1 and stop2 models (with shallower trees and stricter stopping criteria) suggest that these models were underfitting the data. The significant improvement in performance when increasing the model complexity (from stop1 to stop3 and through hyperparameter tuning) confirms that the simpler models were underfitting.

5.2 Balancing Model Complexity

The hyperparameter tuning process revealed an interesting pattern: the Gini model benefited from increased complexity (deeper trees, fewer samples required for splitting), while the misclassification model performed best with more moderate parameters. This suggests that the Gini criterion is more robust to increased model complexity and less prone to overfitting in this specific dataset.

5.3 Remedies for Overfitting

Although the best model did not show significant signs of overfitting, I discuss potential remedies that could be applied if overfitting were to occur:

5.3.1 Tree Pruning

Pruning involves removing parts of the tree that provide little power to classify instances, reducing complexity while maintaining or improving performance. Two common approaches are:

1. **Pre-pruning (Early stopping)**: Stop the tree growth before it becomes too complex by using stopping criteria like min_samples_split, min_samples_leaf, and max_depth.
2. **Post-pruning (Cost-complexity pruning)**: Grow a full tree and then prune it back based on a cost-complexity measure. This approach involves:
    - Growing a full tree
    - Calculating the cost-complexity measure for each subtree
    - Pruning branches that increase the cost-complexity measure the least
    - Selecting the optimal subtree using cross-validation

Implementation of cost-complexity pruning would involve:

```
def prune_tree(node, alpha):
    """
    Prune the tree using cost-complexity pruning.

    Parameters:
    -----------
    node : Node
        The root node of the tree or subtree
    alpha : float
        The complexity parameter

    Returns:
    --------
    node : Node
        The pruned tree or subtree
    """
    if node.is_leaf():
        return node

    # Recursively prune left and right subtrees
    node.left = prune_tree(node.left, alpha)
    node.right = prune_tree(node.right, alpha)

    # If both children are leaves, consider pruning
    if node.left.is_leaf() and node.right.is_leaf():
```

```
        # Calculate error before pruning
        error_before = calculate_error(node)

        # Calculate error after pruning
        error_after = calculate_error_if_pruned(node)

        # Calculate complexity cost
        complexity_cost = count_leaves(node) - 1

        # Prune if it reduces the cost-complexity measure
        if error_after <= error_before + alpha * complexity_cost:
            return create_leaf_from_node(node)

    return node
```

5.3.2 Ensemble Methods

Another approach to address overfitting is to use ensemble methods, which combine multiple decision trees:

1. **Random Forests**: Build multiple decision trees using bootstrap samples and random feature subsets, then average their predictions.
2. **Gradient Boosting**: Build trees sequentially, with each tree correcting the errors of the previous ones.

5.3.3 Optimal Stopping Criteria

Finding the optimal stopping criteria is crucial for preventing overfitting. The implemented hyperparameter tuning process explored various combinations of max_depth, min_samples_split, and min_samples_leaf to find the best balance between model complexity and generalization.

For the Gini criterion, the optimal stopping criteria were:

- max_depth=9
- min_samples_split=2
- min_samples_leaf=1

These parameters allowed the model to capture complex patterns in the data without overfitting, as evidenced by the small gap between training and test accuracies.

5.4 Remedies for Underfitting

For models showing signs of underfitting (like the stop1 and stop2 configurations), the following remedies could be applied:

1. **Increase model complexity**: Allow deeper trees and more flexible splitting by reducing min_samples_split and min_samples_leaf.
2. **Feature engineering**: Create new features or transformations that better capture the underlying patterns.
3. **Reduce regularization**: Decrease the constraints that limit the model's ability to fit the training data.

The implemented hyperparameter tuning process effectively addressed underfitting by finding the optimal level of model complexity for each splitting criterion.

6 Conclusions

In conclusion, the analysis demonstrates that decision tree predictors with appropriate splitting criteria and hyperparameters can achieve high accuracy in mushroom classification. The Gini impurity criterion with optimized hyperparameters provided the best performance, achieving a test accuracy of 91.32% with good generalization. These findings highlight the importance of proper model selection, hyperparameter tuning, and careful consideration of overfitting/underfitting issues in machine learning applications.