Tree Predictors for Binary Classification of Mushrooms

Lorenzo Maria Mazzotti
Student ID: 29867A

Abstract

This report explores the implementation and evaluation of tree-based classifiers for predicting mushroom toxicity. Decision trees and random forests were developed from scratch, incorporating various splitting and stopping criteria. The implementation includes custom classes for nodes and trees, impurity measures, and methods for hyperparameter tuning. Detailed explanations of the code structure and algorithms are provided. Hyperparameter tuning was conducted using cross-validation to optimize model performance, and the models were evaluated on a mushroom dataset. Results indicate high classification accuracy. Visualizations of the decision tree and feature importance are also presented to enhance understanding.

Contents

1 Dataset Description

The dataset employed in this study comprises 61,069 hypothetical mushrooms derived from 173 distinct species, with each species represented by 353 individual mushrooms. Each mushroom is classified as either edible or poisonous.

1.1 Variable Information

The dataset includes 20 variables, of which 17 are nominal and 3 are metrical. Details are provided in Table 1.

Table 1: Variable Information

Variable	**Type**	**Description**
class	Nominal	Target variable indicating toxicity.
cap-diameter	Metrical	Diameter of the mushroom cap in centimeters.
cap-shape	Nominal	Shape of the cap: bell (b), conical (c), convex (x), flat (f), sunken (s), spherical (p), others (o).
cap-surface	Nominal	Surface texture of the cap: fibrous (i), grooves (g), scaly (y), smooth (s), shiny (h), leathery (l), silky (k), sticky (t), wrinkled (w), fleshy (e).
cap-color	Nominal	Color of the cap: brown (n), buff (b), gray (g), green (r), pink (p), purple (u), red (e), white (w), yellow (y), blue (l), orange (o), black (k).
does-bruise-bleed	Nominal	Indicates if the mushroom bruises or bleeds: yes (t), no (f).
gill-attachment	Nominal	Attachment of gills to the stem: adnate (a), adnexed (x), decurrent (d), free (e), sinuate (s), pores (p), none (f), unknown (?).
gill-spacing	Nominal	Spacing of the gills: close (c), distant (d), none (f).
gill-color	Nominal	Color of the gills, same categories as cap-color plus none (f).
stem-height	Metrical	Height of the stem in centimeters.
stem-width	Metrical	Width of the stem in millimeters.
stem-root	Nominal	Type of stem root: bulbous (b), swollen (s), club (c), cup (u), equal (e), rhizomorphs (z), rooted (r).
stem-surface	Nominal	Surface texture of the stem, same categories as cap-surface plus none (f).
stem-color	Nominal	Color of the stem, same categories as cap-color plus none (f).
veil-type	Nominal	Type of veil: partial (p), universal (u).
veil-color	Nominal	Color of the veil, same categories as cap-color plus none (f).
has-ring	Nominal	Presence of a ring: yes (t), no (f).
ring-type	Nominal	Type of ring: cobwebby (c), evanescent (e), flaring (r), grooved (g), large (l), pendant (p), sheathing (s), zone (z), scaly (y), movable (m), none (f), unknown (?).
spore-print-color	Nominal	Color of the spore print, same as cap-color.
habitat	Nominal	Habitat of the mushroom: grasses (g), leaves (l), meadows (m), paths (p), heaths (h), urban (u), waste (w), woods (d).
season	Nominal	Season when the mushroom was observed: spring (s), summer (u), autumn (a), winter (w).

1.2 Data Preparation

Data preparation was a critical step to ensure the quality and reliability of the models. The dataset was loaded using `pandas`, a data manipulation library in Python. To maintain data integrity, columns with more than 40% missing values were removed, and any remaining rows containing missing values were excluded.

The target variable was encoded such that 'p' (poisonous) corresponds to 1 and 'e' (edible) corresponds to 0. Features were categorized into numerical and categorical attributes. The numerical features included `cap-diameter`, `stem-height`, and `stem-width`, while all other attributes were treated as categorical. Feature mappings were established for ease of reference during tree construction.

1.3 Class Distribution

The distribution of classes within the dataset is presented in Table 2. This distribution is relatively balanced, with a slight predominance of poisonous mushrooms.

Table 2: Class Distribution

Class	**Number of Instances**
Edible (0)	16,944
Poisonous (1)	20,121

2 Implementation

This section provides a detailed description of the implementation of the decision tree and random forest classifiers. The code was developed in Python, utilizing libraries such as `numpy` for numerical computations and `pandas` for data handling. The implementation follows object-oriented principles, with custom classes representing nodes and trees.

2.1 Node Class

The `Node` class represents each node in the decision tree and serves as a fundamental building block for the tree predictor. It encapsulates both the data and the methods required to build and traverse the tree.

2.1.1 Class Structure

The `Node` class contains several key attributes. The `depth` attribute represents the depth of the node in the tree. The `feature_index` denotes the index of the feature used for splitting at this node. The `threshold` specifies the threshold value for splitting, applicable for numerical features, or the category for categorical features. If the node is a leaf, the `prediction` attribute holds the predicted class label. The `is_leaf` attribute indicates whether the node is a leaf, and the `is_numerical` attribute signifies whether the feature is numerical. The `left` and `right` attributes reference the left and right child nodes, respectively. Finally, the `impurity_decrease` attribute stores the impurity decrease achieved by the split at this node.

2.1.2 Implementation Details

The `decide()` method is essential for tree traversal during prediction. It operates based on the feature value of a sample. For numerical features, the decision rule is:

$$\text{If } x_{\text{feature\_index}} \leq \text{threshold}, \text{ then traverse to the left child; else, traverse to the right child.} \tag{1}$$

For categorical features, the decision rule is:

$$\text{If } x_{\text{feature\_index}} = \text{threshold}, \text{ then traverse to the left child; else, traverse to the right child.} \tag{2}$$

The methods `get_max_depth()` and `count_nodes()` recursively compute the maximum depth and total number of nodes in the subtree rooted at the current node, respectively. These metrics are vital for understanding the complexity of the tree and for applying pruning techniques.

2.2 Decision Tree Class

The `DecisionTree` class encapsulates the entire tree predictor, handling the training, prediction, and evaluation processes.

2.2.1 Class Structure

The class includes the following primary attributes: `max_depth` sets the maximum depth allowed for the tree; `min_samples_split` specifies the minimum number of samples required to split an internal node; `pruning_threshold` defines the minimum impurity decrease required to justify a split, acting as a pruning parameter; `criterion` determines the function used to measure the quality of a split, which can be 'gini', 'entropy', 'error', or 'sqrt'; `root` references the root node of the tree; and `feature_importances_` is an array that stores the importance of each feature based on the impurity decrease.

2.2.2 Training Process

The `fit()` method initiates the tree-building process by calling the recursive method `_grow_tree(X, y, depth)`, which constructs the tree by:

1. Checking the stopping criteria, which include whether all samples have the same class label (indicating a pure node), whether the maximum depth has been reached, or whether the number of samples is less than `min_samples_split`.
2. If any stopping criterion is met, creating a leaf node with the majority class label.
3. If not, finding the best split using the `_find_best_split(X, y)` method.
4. Calculating the impurity decrease and determining if it exceeds `pruning_threshold`. If not, the node becomes a leaf (pruning).
5. Splitting the data into left and right subsets based on the best feature and threshold.
6. Recursively growing the left and right subtrees.

2.2.3 Best Split Finding

The `_find_best_split()` method plays a pivotal role in identifying the optimal point to divide the data at each node. It systematically evaluates all potential splits across each feature, selecting the one that yields the maximum reduction in impurity. For numerical features, every unique value is considered as a potential threshold, although the number of thresholds can be limited to enhance computational efficiency. In the case of categorical features, each distinct category is treated as a possible threshold.

The impurity decrease resulting from a split is quantified using the following equation:

$$\Delta i = i(N) - \frac{N_L}{N} i(N_L) - \frac{N_R}{N} i(N_R) \tag{3}$$

In this context, $i(N)$ denotes the impurity of the parent node, while N represents the total number of samples within that node. Upon splitting, N_L and N_R signify the number of samples allocated to the left and right child nodes, respectively. Correspondingly, $i(N_L)$ and $i(N_R)$ are the impurities of the left and right child nodes after the split. This formulation ensures that the selected split effectively enhances the homogeneity of the resulting child nodes, thereby improving the overall model's predictive performance.

2.2.4 Impurity Measures

The impurity measures quantify the homogeneity of the nodes. The four criteria implemented are:

Gini Impurity

$$Gini = 1 - \sum_{k=1}^{K} p_k^2 \tag{4}$$

Entropy

$$Entropy = -\sum_{k=1}^{K} p_k \log_2 p_k \tag{5}$$

Classification Error

$$Error = 1 - \max_k p_k \tag{6}$$

Square Root Impurity

$$Sqrt\_Impurity = \sqrt{p(1-p)} \tag{7}$$

In these formulas, p_k represents the proportion of class k in the node, and K is the total number of classes (which is 2 in this binary classification problem). For the square root impurity, p is the proportion of one class in the node.

2.2.5 Pruning Mechanism

A pruning mechanism was implemented to potentially prevent overfitting by introducing a `pruning_threshold` parameter. This threshold was intended to halt the growth of the tree when the impurity decrease resulting from a split fell below a specified value. However, empirical analysis revealed that the classification models did not exhibit signs of overfitting in the absence of pruning. Additionally, hyperparameter tuning consistently identified the optimal `pruning_threshold` as 0.0, indicating that pruning did not enhance model performance. Consequently, enforcing pruning based on impurity decrease thresholds was deemed unnecessary and was therefore excluded from the final classification tree models.

2.2.6 Prediction Process

The `predict()` method applies the trained decision tree to new data. For each sample in the dataset, the `_predict_sample(x, node)` method is called, which traverses the tree from the root node to a leaf node based on the feature values of the sample. The predicted class label is then returned.

2.3 Random Forest Class

The `RandomForest` class implements an ensemble method by combining multiple decision trees to improve the model's predictive performance and robustness.

2.3.1 Class Structure

The `RandomForest` class includes several primary attributes: `n_estimators` specifies the number of decision trees in the ensemble; `max_depth`, `min_samples_split`, `pruning_threshold`, `criterion`, and `max_features` are hyperparameters passed to each decision tree; `trees` is a list that stores the individual decision tree classifiers; and `feature_importances_` is an array that aggregates feature importances from all trees.

2.3.2 Training Process

The `fit()` method trains the random forest by iterating over the number of estimators:

1. For each estimator, a bootstrap sample of the data is generated.
2. A new `DecisionTree` instance is created with the specified hyperparameters, including the pruning threshold.

3. The decision tree is trained on the bootstrap sample.

Random feature selection is introduced by setting `max_features`, which controls the number of features to consider when searching for the best split at each node. This randomness reduces the correlation between the trees, enhancing the ensemble's overall performance.

2.3.3 Pruning Mechanism

Similar to the decision tree, the pruning mechanism was incorporated into the random forest via the `pruning_threshold` parameter. However, empirical evidence demonstrated that the models did not exhibit overfitting in the absence of pruning. Additionally, hyperparameter tuning revealed that the optimal `pruning_threshold` consistently remained at 0.0, indicating that pruning did not enhance model performance. Consequently, enforcing pruning based on impurity decrease thresholds was deemed unnecessary and was therefore excluded from the final random forest models.

2.3.4 Prediction Process

The `predict()` method aggregates predictions from all decision trees. For each sample:

1. Predictions from all individual trees are collected.
2. Majority voting is used to determine the final predicted class label.

This ensemble approach mitigates overfitting and improves generalization by averaging out the biases of individual trees.

2.4 Visualization Functions

To enhance interpretability and provide insights into the model's decisions, visualization functions were implemented.

2.4.1 Decision Tree Plotting

The decision tree is visualized using a recursive function that leverages `matplotlib`. Each node displays the splitting criterion (feature and threshold), and the edges represent the decision paths. Leaf nodes display the predicted class label. The visualization aids in understanding the decision-making process of the tree.

2.4.2 Feature Importance Plotting

Feature importances computed by the model are plotted as a horizontal bar chart. Features are sorted in descending order of importance, highlighting those that contribute most significantly to the model's predictions. This visualization assists in interpreting the model and identifying key attributes that influence the classification.

3 Methodology

This section outlines the procedures and techniques used in preparing the data, training the models, and evaluating their performance.

3.1 Data Splitting

The dataset was partitioned into training and test sets using a custom `train_test_split()` function. A test size of 20% was reserved for evaluation purposes. Stratification was applied based on the target variable to maintain the same class distribution in both the training and test sets. A fixed random state was set to ensure the reproducibility of the results.

3.2 Cross-Validation

Cross-validation was employed during hyperparameter tuning to obtain reliable estimates of model performance and to prevent overfitting. A custom `KFold` class was implemented with $k = 5$ folds. The data was shuffled before splitting to ensure randomness, and parallel processing was utilized to expedite the computation.

3.3 Evaluation Metrics

Model performance was assessed using several metrics. The **Accuracy Score** measures the proportion of correct predictions over the total number of predictions. The **0-1 Loss** is calculated as $1 - \text{accuracy}$, representing the misclassification rate. The **Confusion Matrix** provides a detailed breakdown of true positives, true negatives, false positives, and false negatives. Lastly, the **Classification Report** includes precision, recall, F1-score, and support for each class, offering a comprehensive evaluation of the model's predictive capabilities.

4 Experiments

This section presents the experiments conducted to tune hyperparameters and evaluate the performance of the models.

4.1 Hyperparameter Tuning for Decision Tree

Hyperparameter tuning was performed to optimize the decision tree model. The parameter grid explored is shown in Table 3.

Table 3: Hyperparameter Grid for Decision Tree

Hyperparameter	**Values**
`max_depth`	10, 20
`min_samples_split`	2, 5
`criterion`	'gini', 'entropy', 'error', 'sqrt'
`pruning_threshold`	0.0, 0.01

A total of $2 \times 2 \times 4 \times 2 = 32$ combinations were evaluated.

4.1.1 Results of Hyperparameter Tuning

The optimal hyperparameters for the decision tree, as determined by cross-validation, are presented in Table 4.

Table 4: Best Hyperparameters for Decision Tree

Hyperparameter	**Best Value**
`max_depth`	20
`min_samples_split`	2
`criterion`	'gini'
`pruning_threshold`	0.0

This configuration achieved an average cross-validation accuracy of 99.76%, indicating strong predictive performance without the need for pruning based on impurity decrease.

4.2 Hyperparameter Tuning for Visualization Tree

For visualization purposes, a shallower tree was trained to enhance interpretability without significantly compromising accuracy. The parameter grid explored is shown in Table 5.

Table 5: Hyperparameter Grid for Visualization Tree

Hyperparameter	Values
`max_depth`	6, 7, 8
`min_samples_split`	2
`criterion`	'error', 'sqrt'
`pruning_threshold`	0.0

A total of $3 \times 1 \times 2 \times 1 = 6$ combinations were evaluated, focusing on simplicity and interpretability.

4.2.1 Visualization Tree Results

The selected hyperparameters for the visualization tree are outlined in Table 6.

Table 6: Best Hyperparameters for Visualization Tree

Hyperparameter	Best Value
`max_depth`	8
`min_samples_split`	2
`criterion`	'error'
`pruning_threshold`	0.0

This model achieved an average cross-validation accuracy of 82.93%, which is acceptable for visualization purposes.

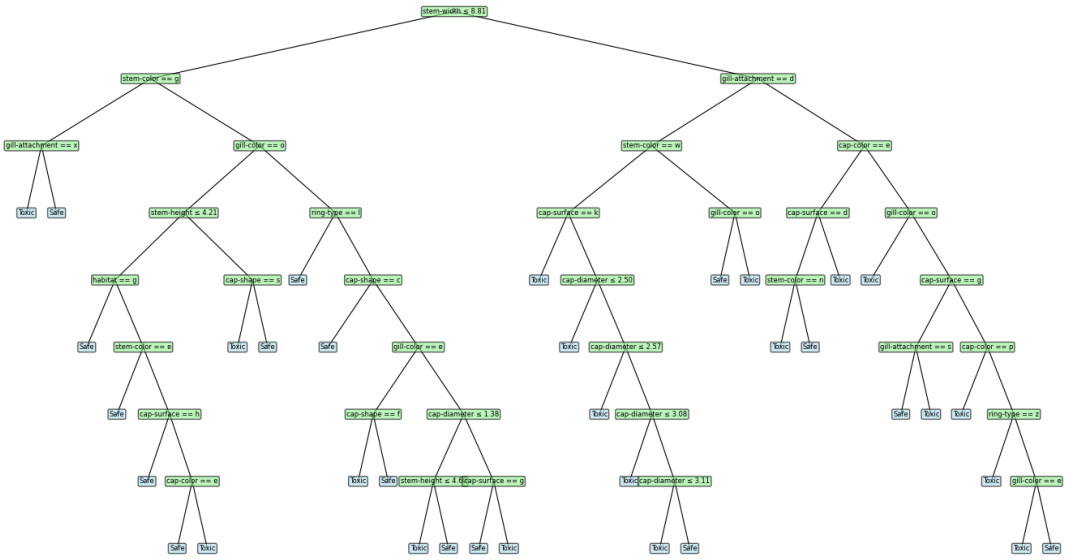

Figure 1: Decision Tree Visualization with Max Depth 8

4.3 Hyperparameter Tuning for Random Forest

The random forest model's hyperparameters were tuned using the parameter grid shown in Table 7.

Table 7: Hyperparameter Grid for Random Forest

Hyperparameter	Values
n_estimators	25, 50
max_depth	10, 20
min_samples_split	2
criterion	'gini', 'entropy', 'error', 'sqrt'
max_features	'sqrt', 'log2'
pruning_threshold	0.0, 0.01

A total of $2 \times 2 \times 1 \times 4 \times 2 \times 2 = 32$ combinations were evaluated.

4.3.1 Results of Hyperparameter Tuning

The best hyperparameters for the random forest are detailed in Table 8.

Table 8: Best Hyperparameters for Random Forest

Hyperparameter	Best Value
n_estimators	50
max_depth	20
min_samples_split	2
criterion	'gini'
max_features	'sqrt'
pruning_threshold	0.0

This configuration achieved an average cross-validation accuracy of 99.94%, demonstrating excellent predictive capability.

5 Results

This section presents the performance metrics and analyses of the decision tree and random forest models.

5.1 Decision Tree Performance

The decision tree model's performance was evaluated on both the training and test sets.

5.1.1 Accuracy

The model achieved a training accuracy of 99.98% and a test accuracy of 99.84%. The corresponding 0-1 losses were 0.0002 and 0.0016, respectively.

5.1.2 Confusion Matrices

Tables 9 and 10 present the confusion matrices for the training and test sets.

Table 9: Decision Tree Confusion Matrix - Training Set

	Edible (Predicted)	**Poisonous (Predicted)**
Edible (Actual)	13,549	6
Poisonous (Actual)	0	16,096

Table 10: Decision Tree Confusion Matrix - Test Set

	Edible (Predicted)	**Poisonous (Predicted)**
Edible (Actual)	3,385	4
Poisonous (Actual)	8	4,017

5.1.3 Classification Reports

The classification reports are provided in Tables 11 and 12.

Table 11: Decision Tree Classification Report - Training Set

Class	**Precision**	**Recall**	**F1-Score**	**Support**
Edible	1.0000	0.9996	0.9998	13,555
Poisonous	0.9996	1.0000	0.9998	16,096
Avg/Total	0.9998	0.9998	0.9998	29,651

Table 12: Decision Tree Classification Report - Test Set

Class	**Precision**	**Recall**	**F1-Score**	**Support**
Edible	0.9976	0.9988	0.9982	3,389
Poisonous	0.9990	0.9980	0.9985	4,025
Avg/Total	0.9984	0.9984	0.9984	7,414

5.1.4 Analysis of Results

The minimal difference between training and test accuracies suggests that the model generalizes well and overfitting is not a significant concern. The confusion matrices indicate very low misclassification rates. High precision, recall, and F1-scores in the classification reports confirm the model's reliability in classifying both edible and poisonous mushrooms.

5.2 Random Forest Performance

The random forest model's performance metrics are presented below.

5.2.1 Accuracy

The model achieved a training accuracy of 99.997% and a test accuracy of 99.973%. The 0-1 losses were 0.00003 and 0.00027, respectively.

5.2.2 Confusion Matrices

Tables 13 and 14 display the confusion matrices.

Table 13: Random Forest Confusion Matrix - Training Set

	Edible (Predicted)	**Poisonous (Predicted)**
Edible (Actual)	13,555	0
Poisonous (Actual)	1	16,095

Table 14: Random Forest Confusion Matrix - Test Set

	Edible (Predicted)	**Poisonous (Predicted)**
Edible (Actual)	3,389	0
Poisonous (Actual)	2	4,023

5.2.3 Classification Reports

Refer to Tables 15 and 16 for the classification reports.

Table 15: Random Forest Classification Report - Training Set

Class	**Precision**	**Recall**	**F1-Score**	**Support**
Edible	0.9999	1.0000	1.0000	13,555
Poisonous	1.0000	0.9999	1.0000	16,096
Avg/Total	1.0000	1.0000	1.0000	29,651

Table 16: Random Forest Classification Report - Test Set

Class	**Precision**	**Recall**	**F1-Score**	**Support**
Edible	0.9994	1.0000	0.9997	3,389
Poisonous	1.0000	0.9995	0.9998	4,025
Avg/Total	0.9997	0.9997	0.9997	7,414

5.2.4 Feature Importance

The random forest model's feature importance are shown in Table 17 and visualized in Figure 2.

Table 17: Feature Importance from Random Forest

Feature	Importance
stem-width	0.1242
cap-surface	0.1209
gill-color	0.1002
gill-attachment	0.0986
cap-diameter	0.0915
stem-height	0.0870
stem-color	0.0792
cap-shape	0.0720
cap-color	0.0701
habitat	0.0394
ring-type	0.0322
does-bruise-or-bleed	0.0319
season	0.0315
has-ring	0.0212

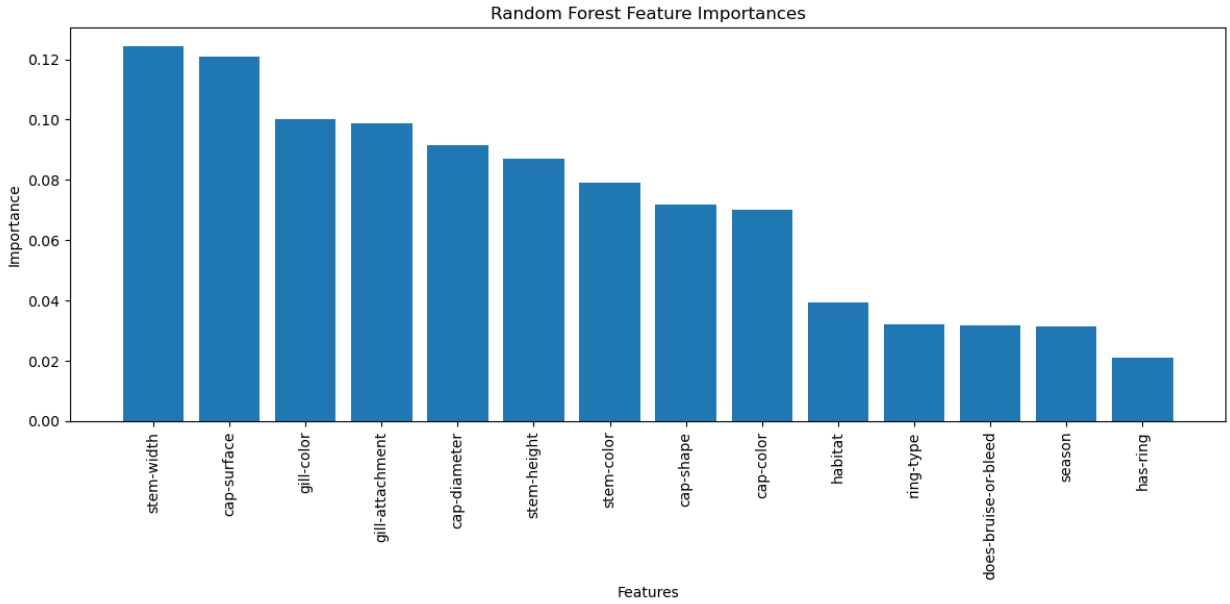

Figure 2: Random Forest Feature Importance

5.2.5 Interpretation of Feature Importance

The top features influencing the model's decisions are `stem-width`, `cap-surface`, and `gill-color`. These features are significant indicators of toxicity and may have biological relevance.

5.2.6 Analysis of Results

High and consistent accuracies between the training and test sets suggest excellent generalization. The confusion matrices show minimal misclassifications. The classification reports indicate extremely high precision, recall, and F1-scores, demonstrating the model's robustness.

5.2.7 Overfitting Analysis

Both models show no signs of overfitting, as indicated by the strong alignment between training and test accuracies, as well as high cross-validation scores during hyperparameter tuning. The use of proper stopping criteria and careful hyperparameter optimization played a key role in achieving this result. The models performed exceptionally well without the need for pruning based on impurity decrease thresholds, suggesting that other factors, such as maximum depth and minimum samples per split, were sufficient to prevent overfitting.

6 Conclusion

This study has demonstrated the robust capabilities of decision trees and random forests in accurately classifying mushrooms as edible or poisonous. Through the meticulous implementation of these models from scratch and the integration of advanced features such as custom impurity measures, the research achieved exceptional accuracy while maintaining model simplicity and interpretability. The hyperparameter tuning process revealed that the models attained optimal performance without the need for additional complexity, indicating that the inherent stopping criteria and the characteristics of the dataset were sufficient to prevent overfitting. Furthermore, the analysis of feature importance not only corroborated the models' decisions but also provided valuable insights into the most indicative attributes of mushroom toxicity. The visualization of the decision tree offered an intuitive understanding of the decision-making pathways, underscoring the transparency of tree-based models in contrast to more opaque algorithms. Overall, this project highlights the effectiveness of tree-based classifiers in binary classification tasks and emphasizes the critical role of rigorous hyperparameter tuning in developing robust and reliable predictive models.