# Tree Predictors for Binary Classification

Amirreza Dashti Genave

December 2024

## Declaration

I/We declare that this material, which I/We now submit for assessment, is entirely my/our 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/our work. I/We 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/us or any other person for assessment on this or any other course of study.

## Abstract

This project focuses on implementing tree predictors for binary classification from scratch, specifically considering the Mushroom dataset. The primary goal is to know whether mushrooms are poisonous using decision tree algorithms. These trees use single-feature binary tests for decision-making at internal nodes, considering both numerical thresholds and categorical membership tests. This report demonstrates the methodology, hyperparameter tuning, and experimental results to evaluate the model's effectiveness and mitigate potential issues such as overfitting.

## 1 Introduction

This report provides an analysis of tree-based predictors for binary classification, specifically focusing on the task of determining whether mushrooms are poisonous using the Mushroom dataset as the training dataset. Tree predictors, or decision trees, are known for their simplicity and effectiveness in classification tasks. They work by iteratively partitioning data based on specific criteria, and this makes them a good choice for this task.

The study is designed to provide a deep understanding of the project's methodology and principal results. It begins with an explanation of the research question and the methodology used to examine the binary classification task. The report then describes the Mushroom dataset, showing its features.

Next, the data preprocessing steps are explained. These steps were critical in preparing the dataset for training the model, ensuring that the data was consistent and accurate before it was fed into the decision tree model. Following this, the design and implementation of the tree predictor are explained, including the criteria used for data splitting and the stopping conditions set to prevent overfitting.

The report also discusses hyperparameter tuning and cross-validation techniques. These optimization methods were applied to improve model performance, and cross-validation was used to assess the reliability of the model. The experimental results are presented and analyzed, focusing on the model's performance and efforts to address potential issues like overfitting and underfitting.

Finally, the report concludes with a summary of the experimental results and provides recommendations for further work.

## 2 Research Question and Methodology

The primary objective of this project is to design a tree predictor, specifically a decision tree, to classify mushrooms as either poisonous or edible. By analyzing features within the Mushroom dataset, the decision tree aims to provide accurate and interpretable predictions based on the structure of the data.

A decision tree is a supervised learning model that works by splitting data into subsets based on specific decision criteria. It consists of a root node, internal nodes, and leaf nodes. The root node represents the starting point, where the first decision is made, while internal nodes split the data further based on conditions. Finally, leaf nodes provide the outcome, which in our case is the classification of mushrooms. Each split within the tree is based on a single feature, using thresholds for numerical attributes or membership tests for categorical attributes.

The methodology of this project involves several steps. First, the implementation of the decision tree structure, focusing on its main components, such as nodes, splitting criteria, and stopping conditions. Next, a training process is performed to ensure the model learns patterns effectively from the data. Finally, hyperparameter tuning and validation are applied to optimize the model's performance and mitigate overfitting or underfitting.

Through this methodology, the project tries to provide insights into the construction and evaluation of a decision tree, as well as its effectiveness in solving a binary classification problem.

# 3 Dataset

The dataset used in this project is the Mushroom dataset, which consists of 61,069 samples of mushrooms described by various features. The mushrooms are classified into two main categories: edible and poisonous. The data includes both numerical and categorical features. The dataset contains 20 columns, with some features having missing values. A brief overview of the dataset's columns and the number of non-null values for each feature is presented in the table below:

| **Index** | **Feature Name** | **Non-Null Values** |
|---|---|---|
| 0 | cap-diameter | 61069 |
| 1 | cap-shape | 61069 |
| 2 | cap-surface | 46949 |
| 3 | cap-color | 61069 |
| 4 | does-bruise-or-bleed | 61069 |
| 5 | gill-attachment | 51185 |
| 6 | gill-spacing | 36006 |
| 7 | gill-color | 61069 |
| 8 | stem-height | 61069 |
| 9 | stem-width | 61069 |
| 10 | stem-root | 9531 |
| 11 | stem-surface | 22945 |
| 12 | stem-color | 61069 |
| 13 | veil-type | 3177 |
| 14 | veil-color | 7413 |
| 15 | has-ring | 61069 |
| 16 | ring-type | 58598 |
| 17 | spore-print-color | 6354 |
| 18 | habitat | 61069 |
| 19 | season | 61069 |

Table 1: Overview of the Mushroom Dataset Features

This dataset includes both numerical features (e.g., cap diameter, stem height, and stem width) and categorical features (e.g., cap shape, cap color, gill color). Some of the features, such as "stem-root," "veil-type," and "spore-print-color," have a significant number of missing values. These missing values need to be handled appropriately during data preprocessing to ensure the dataset's suitability for training our decision tree model.

## 3.1 Data Preprocessing

Before training the decision tree model, the dataset experiences a preprocessing phase to handle missing values and ensure balanced classes. This process includes two thresholds to deal with null values and class imbalance.

First, we introduce two thresholds:

- **null_threshold:** This threshold is used to identify features (columns) that have a significant number of missing values. Features with missing values exceeding this threshold are removed from the dataset to maintain the quality of the data.
- **imbalance_threshold:** This threshold checks if the dataset is imbalanced in terms of the distribution of the classes. If the class distribution is significantly skewed, we need to use another approach to maintain the removed features.

Next, for the remaining features, if any row still contains at least one missing value after the feature removal step, it is dropped. This ensures that the final dataset is complete, with no missing values.

After these preprocessing steps, the dataset consists of 13 features and 58,598 records. The class distribution is as follows:

- **Class 0 (Edible):** 25,769 samples (43.98%)
- **Class 1 (Poisonous):** 32,829 samples (56.02%)

The chart below shows the class distribution:

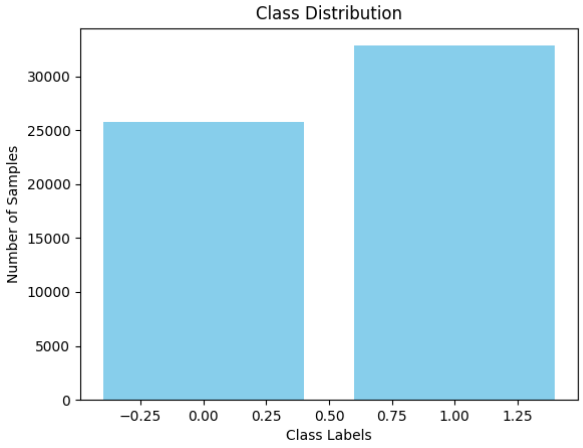


Figure 1: Class Distribution in the Mushroom Dataset

# 4 Model

In this section, we describe the implementation of a decision tree model built from scratch, demonstrating its components, functionality, and mechanisms that

control its construction and prediction processes. The model employs a recursive tree-building strategy, uses impurity calculations, and includes stopping criteria to balance accuracy and generalization. Below, we cover its core elements and important characteristics.

## 4.1 The `Node` Class

The backbone of the implementation is the `Node` class, which represents individual nodes within the tree. Each node has information about its splitting feature and threshold (for numerical features) or categories (for categorical features), as well as pointers to its left and right child nodes. For leaf nodes, it stores the predicted class (`value`). A convenient method, `is_leaf_node()`, identifies whether a node is a leaf.

## 4.2 The `DecisionTree` Class

The decision tree is managed by the `DecisionTree` class, which encapsulates several parameters that control the tree's growth and prevent overfitting:

- `max_depth`: The maximum allowable depth of the tree.
- `min_samples_split`: The minimum number of samples required to perform a split.
- `min_impurity_decrease`: The minimum impurity reduction needed to justify a split.
- `criterion`: The impurity measure, which can be either `gini` or `entropy`.

These parameters work together to ensure that the decision tree balances accuracy and generalization.

## 4.3 Tree Construction

Tree construction begins with the `_build_tree()` method, a recursive function that splits the data into subsets at each node. The method evaluates stopping criteria at every level to decide whether further splitting is necessary:

- The tree reaches the maximum depth (`max_depth`).
- All samples at the node belong to the same class.
- The number of samples at the node is below `min_samples_split`.
- The impurity reduction achieved by the split is less than `min_impurity_decrease`.

If any of these conditions are met, the node becomes a leaf, and the majority class among its samples is assigned using the `_most_common_label()` method.

### 4.4 Splitting and Impurity Calculations

To identify the optimal split at each node, the `_best_split()` function evaluates potential splits for both numerical and categorical features. It relies on helper methods:

- `_split_numeric()`: Evaluates thresholds for splitting numerical features.
- `_split_categorical()`: Generates subsets of categories for categorical features using the `_all_subsets()` method.

Impurity is calculated using the selected `criterion`:

- `_gini()`: Computes Gini impurity.
- `_entropy()`: Computes entropy.

The `_weighted_impurity()` function combines the impurity values of the left and right child nodes, weighted by their respective sample sizes. This ensures that splits are evaluated based on their overall impact on data purity.

### 4.5 Prediction Process

Prediction is carried out by the `_traverse_tree()` function, a recursive method that traverses the tree from root to leaf for each input sample. At each node, the traversal direction is determined by the feature value:

- For numerical features, the value is compared to the threshold.
- For categorical features, membership in the category subset is checked.

The process continues until a leaf node is reached, and the value stored in the leaf is returned as the predicted class.

### 4.6 Key Characteristics

The important characteristic of this decision tree implementation is that it relies on recursion for both tree construction and prediction. Additionally, the model emphasizes robustness and flexibility by handling both numerical and categorical features. This custom decision tree combines recursive logic, impurity-based splitting, and parameterized growth control to deliver a balanced and efficient model.

## 5 Hyperparameter Tuning

In this section, we describe our approach to optimize the hyperparameters of the decision tree model and analyze the accuracy. The main goal of hyperparameter tuning is to identify the best set of parameters that balance the model's performance on both training and validation datasets, ensuring good generalization and minimizing overfitting.

The key hyperparameters tuned include:

- **Criterion**: Determines the impurity measure used for splitting the nodes. Two options are considered: `gini` (Gini impurity) and `entropy`.
- **Max Depth**: Controls the maximum depth of the decision tree. A range of depths from 8 to 19 is evaluated.
- **Min Samples Split**: Specifies the minimum number of samples required to perform a split.
- **Min Impurity Decrease**: Represents the minimum decrease in impurity required to allow a node split.

A **K-Fold Cross-Validation** strategy with 2 folds is applied to evaluate the model's performance. For each combination of the hyperparameters, the model is trained and evaluated on both the training and validation datasets, and average accuracies are recorded for analysis.

## 5.1 Results and Analysis

The plot below shows the **train accuracy** and **cross-validation accuracy** for each hyperparameter combination. A few important observations can be made:

1. **General Trend**: As the **max depth** increases, the model's training accuracy tends to improve and approach 100%. This is expected, as deeper trees can capture more complexity in the training data. However, the cross-validation accuracy does not always follow this trend, indicating potential overfitting.
2. **Criterion Impact**: The results for `entropy` are easier to interpret as it experiences less fluctuations. It follows a predictable pattern as the tree becomes deeper.

   Overall, `entropy` appears to achieve slightly higher validation accuracy in deeper trees.
3. **Best Hyperparameter Combination**: After careful analysis, the chosen hyperparameter combination is:

   `max_depth=14, min_samples_split=5, min_impurity_decrease=0.01, criterion=entropy`.

   This combination keeps a balance between high accuracy and avoiding overfitting. The cross-validation accuracy is close to the maximum observed on the validation set, while the training accuracy remains high but does not reach 100%. Specifically:

   - **Max Depth of 14**: Limits the tree's complexity, preventing excessive overfitting observed at larger depths (e.g., 18 or 19), while still capturing sufficient data patterns for good performance.

- **Min Samples Split of 5**: Ensures that nodes are not split unnecessarily, maintaining a balance between model generalization and complexity.
- **Min Impurity Decrease of 0.01**: Controls the minimum required improvement in impurity, avoiding small, insignificant splits that could lead to overfitting.
- **Entropy Criterion**: Provides slightly better performance compared to Gini in this case, as observed from the results.

This configuration achieves high validation accuracy while maintaining a moderate depth, making it a robust choice for generalization to unseen data.

4. **Overfitting Behavior**: For very large depths (e.g., `max_depth=19`), the training accuracy reaches 100%, but the validation accuracy oscillations, suggest that deeper trees start to overfit to the training data without improving generalization.

## 5.2 Conclusion

Through hyperparameter tuning, we found the combination of `max_depth = 14`, `min_samples_split = 5`, `min_impurity_decrease = 0.01`, and `criterion = entropy` as the optimal configuration. This combination ensures high accuracy on the validation set while keeping overfitting under control. The observed results emphasize the importance of using cross-validation to assess generalization and to avoid relying solely on training accuracy when selecting hyperparameters.

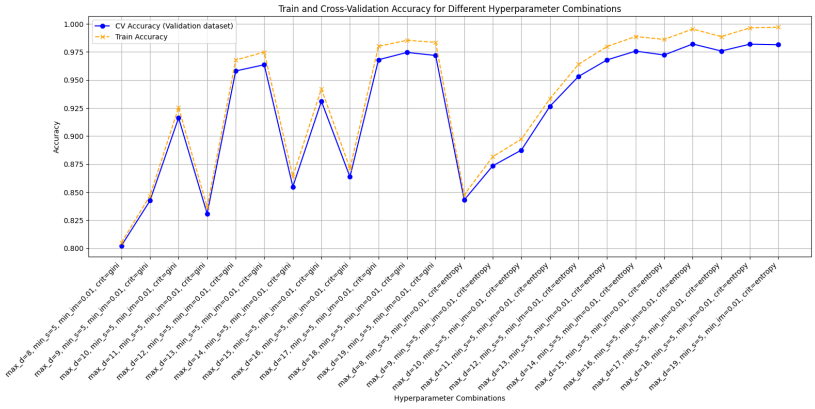


Figure 2: Train and Cross-Validation Accuracy for Different Hyperparameter Combinations

# 6 Experimental Results

In this section, we present the experimental results obtained by training the decision tree model with the selected hyperparameters:

`max_depth=14, min_samples_split=5, min_impurity_decrease=0.01, criterion=entropy`.

Unlike the hyperparameter tuning phase, no cross-validation was performed this time. The training was conducted on the full training dataset, and the results were evaluated on both the training and test datasets. Below, we discuss the results in detail.

## 6.1 Performance Metrics

The model achieved the following accuracy scores:

- **Training Accuracy:** 0.9700
- **Test Accuracy:** 0.9624

The high training accuracy indicates that the model effectively learned the patterns in the training data. The slight drop in test accuracy suggests that the model generalizes well to unseen data, with minimal overfitting.

Additionally, we calculated detailed classification metrics on the test dataset:

| Class | Precision | Recall | F1-Score | Support |
|---|---|---|---|---|
| 0 | 0.94 | 0.98 | 0.96 | 5103 |
| 1 | 0.98 | 0.95 | 0.97 | 6617 |
| **Accuracy** | 0.96 (11720 samples) | | | |
| **Macro Avg** | 0.96 | 0.96 | 0.96 | 11720 |
| **Weighted Avg** | 0.96 | 0.96 | 0.96 | 11720 |

The metrics indicate strong performance for both classes. The `precision` and `recall` scores are well-balanced, with class 1 (positive class) achieving slightly better `precision`. This balance shows that the model performs well across different evaluation aspects.

## 6.2 Training Error

The training error, measured as the 0-1 loss (the proportion of misclassified samples), is reported as:

**Training Error:** 0.0300

This shows a very low error rate. Only 3% of the training samples were misclassified. This low training error, combined with high test accuracy, confirms that the selected hyperparameters enabled the model to effectively capture the structure of the data while maintaining good generalization.

### 6.3 Discussion

Overall, the results demonstrate the robustness of the decision tree model with the chosen hyperparameters. The balance between training and test accuracy, as well as the detailed classification metrics, shows the model's ability to handle both the training data and unseen test data effectively. The low training error further emphasizes the quality of the training process, ensuring minimal misclassification within the dataset.

## 7 Conclusion

This study demonstrates the effectiveness of custom decision tree predictors in solving binary classification problems using interpretable and structured models. By tuning hyperparameters and addressing issues like overfitting, the decision tree achieves high accuracy and robustness. The recursive implementation effectively handles diverse data types and ensures scalability for complex tasks.

Future work could explore advanced pruning techniques and ensemble methods like random forests to improve model generalization further. Additionally, integrating more sophisticated handling of missing values and class imbalances would enhance the versatility of the tree predictors for real-world datasets.