Statistical Methods for Machine Learning

Tree predictors for binary classification of Mushroom Dataset

Mina Beric

January 2025

UNIVERSITÀ
DEGLI STUDI
DI MILANO

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.

Contents

1 Introduction

This project aims to develop a custom binary tree classifier to predict whether a mushroom is edible or poisonous. The project proceeds in several stages, starting with the exploration and preprocessing of the dataset, followed by the construction of the classifier, and concluding with model evaluation.

The dataset used in this project was analyzed through Exploratory Data Analysis (EDA) and preprocessed to prepare it for training. The dataset was then divided into training and testing sets, with 80% of the data used for training and 20% for testing, ensuring that the split maintained the class distribution through stratified sampling.

The classifier was built by first defining a `Node` class, which represents both decision and leaf nodes. The `Decision_Tree` class was then developed to construct the decision tree using recursive algorithms. The tree-building process starts by evaluating potential splits based on feature thresholds that maximize information gain, measured using criteria such as scaled entropy, Gini index, or squared impurity. Each decision node in the tree represents a feature and its threshold, while leaf nodes contain the final prediction value (edible or poisonous).

To fine-tune the model, hyperparameter optimization was performed using random search with stratified k-fold cross-validation. The parameters optimized during this process include the maximum depth of the tree, the minimum number of samples required to split a node, the criterion used to evaluate splits, and the minimum impurity decrease for stopping the growth of the tree. Stratified k-fold cross-validation ensures that each fold maintains the same class distribution, providing more reliable evaluation metrics.

Finally, the performance of the decision tree model was evaluated by calculating key metrics such as accuracy, precision, recall, and F1-score on both the training and test sets. These metrics allow for a comprehensive assessment of the model's ability to classify mushrooms correctly as either edible or poisonous, providing insights into its generalization and robustness.

The following sections of this report detail each stage of the project, from data analysis and model construction to hyperparameter tuning and evaluation.

2 Dataset

The dataset used for this project is the *Secondary Mushroom Dataset*, specifically the `secondary_data.csv`, obtained from the UCI Machine Learning Repository. It consists of 61,069 hypothetical mushroom observations. Each mushroom is classified as either definitely edible, definitely poisonous, or of unknown edibility (which has been combined with the poisonous class for this analysis).

It includes various features that describe the physical characteristics of mushrooms, such as cap diameter, cap shape, stem height, color, and more. The target variable is the classification of the mushroom as edible or poisonous. The following table describes the features present in the dataset, along with their

roles, types, and possible values.

Variable Name	Role	Type	Description
class	Target	Categ.	Classification of mushroom (edible=e or poisonous=p)
cap-diameter	Feat.	Num	Diameter of the cap, expressed in centimetres
cap-shape	Feat.	Categ.	bell=b, conical=c, convex=x, flat=f, sunken=s, spherical=p, others=o
cap-surface	Feat.	Categ.	fibrous=i, grooves=g, scaly=y, smooth=s, shiny=h, leathery=l, silky=k, sticky=t, wrinkled=w, fleshy=e
cap-color	Feat.	Categ.	brown=n, buff=b, grey=g, green=r, pink=p, purple=u, red=e, white=w, yellow=y, blue=l, orange=o, black=k
does-bruise-or-bleed	Feat.	Categ.	bruises-or-bleeding=t, no=f
gill-attachment	Feat.	Categ.	adnate=a, adnexed=x, decurrent=d, free=e, sinuate=s, pores=p, none=f, unknown=?
stem-height	Feat.	Num	Height of the stem, in centimetres
stem-width	Feat.	Num	Width of the stem, in millimetres
stem-color	Feat.	Categ.	Same values as cap-color
has-ring	Feat.	Categ.	ring=t, none=f
ring-type	Feat.	Categ.	cobwebby=c, evanescent=e, flaring=r, grooved=g, large=l, pendant=p, sheathing=s, zone=z, scaly=y, movable=m, none=f, unknown=?
habitat	Feat.	Categ.	grasses=g, leaves=l, meadows=m, paths=p, heaths=h, urban=u, waste=w, woods=d
season	Feat.	Categ.	spring=s, summer=u, autumn=a, winter=w

Table 1: Variable descriptions of the mushroom dataset.

2.1 Preprocessing

The original dataset was inspected, revealing several missing values across multiple columns. To address this, columns with excessive missing data, such as `stem-root`, `stem-surface`, `veil-type`, `veil-color`, and `spore-print-color`, were dropped.

For columns with fewer missing entries, such as `cap-surface`, `gill-attachment`, and `gill-spacing`, The missing values were imputed with the most frequent value (mode) of each respective column. This approach was selected because removing all rows with missing values and duplicates would have reduced the dataset to only 36.2% of its original size. In the context of decision trees, using a significantly smaller dataset can increase the risk of overfitting, as the model might learn noise or irrelevant patterns. By retaining more data through imputation, the aim is to enhance the model's ability to generalize better to unseen data. Additionally, given the small proportion of duplicates, they were

identified and removed to maintain the dataset's integrity, leading to a slight reduction in size from 61,069 to 60,923.

Categorical variables were encoded using one-hot encoding to transform them into binary features. This method was chosen because, while decision trees can handle both numerical and categorical data, one-hot encoding simplifies the tree-building process by enabling binary decisions based on the presence or absence of specific categories. This ensures no assumptions are made about ordinal relationships between categories, thereby preserving the categorical nature of the data. Additionally, the first category of each feature was dropped to prevent multicollinearity, which is particularly crucial to avoid redundant features that could lead to overfitting. Finally, the relevant columns were converted to integer data types. The resulting dataset, now referred to as `data_encoded`, has the following structure:

class	**stem-height**	**stem-width**	**cap-shape_c**	**cap-shape_f**	**...**
1	16.95	17.09	0	0	...
1	17.80	17.74	0	0	...
1	15.77	15.98	0	1	...

Table 2: Sample of the encoded dataset showing selected columns.

The dataset was then divided into independent variables (`X`) and the target variable (`y`), with the `class` column serving as the target. Following this, `data_encoded` was split into training and test sets using an 80/20 ratio, ensuring stratification to preserve the class distribution. This process yielded 48,738 training samples and 12,185 test samples, which were saved in pickle format for later use during hyperparameter tuning.

2.2 Explanatory Data Analysis

Class proportions seem pretty balanced, therefore no balancing techniques were performed.

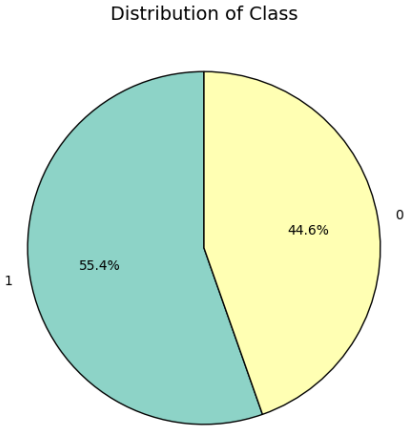


Figure 1: Target class distribution: edible vs. poisonous mushrooms

Distribution of some of the categorical and numerical variables w.r.t. the *class*:

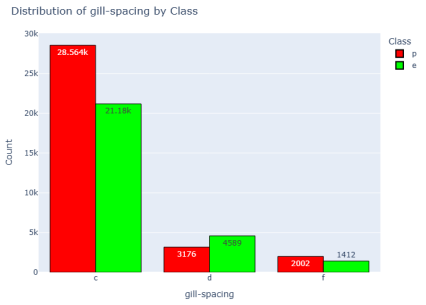


Figure 2: Gill spacing by class.

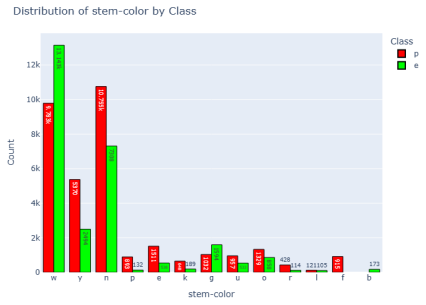


Figure 3: Stem color by class.

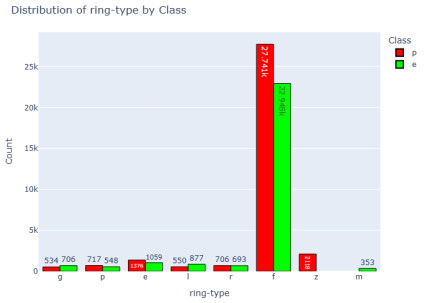


Figure 4: Ring type by class.

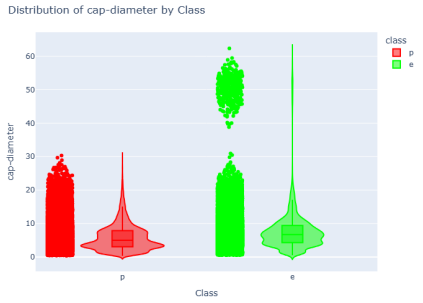


Figure 5: Cap diameter by class.

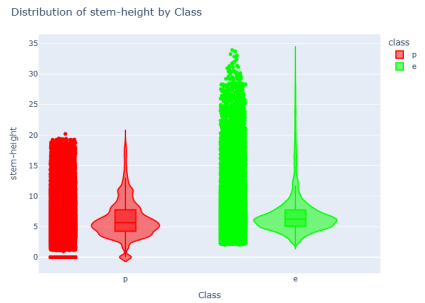


Figure 6: Stem height by class.

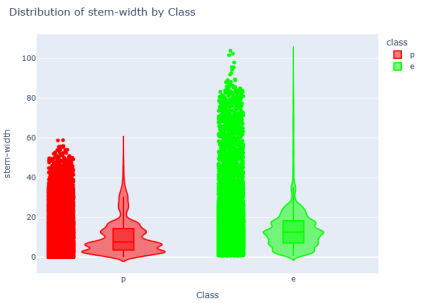


Figure 7: Stem width by class.

3 How the model works

In this section, an explanation of how the decision tree operates is provided. The following steps are taken:

- The dataset starts at the root node.
- At each node, the algorithm selects the optimal feature and threshold for splitting based on a the chosen criterion (e.g., scaled entropy, gini, or Squared Impurity).
- The data is divided into two subsets (left and right) based on the threshold, and the process continues recursively for each child node.
- When a stopping criterion is met, the node becomes a leaf and is assigned the most common label of the remaining examples.

3.1 Splitting criteria

The primary goal of a classification tree is to partition the data into smaller, more uniform groups. This partitioning process is driven by the following three

commonly used splitting criteria, particularly suited for binary classification:

- **Scaled Entropy**: It is a modified measure of impurity specifically tailored for binary classification problems. The formula is:

$$H(S) = -2p\log_2(p) - 2(1-p)\log_2(1-p)$$

  where p is the proportion of the positive class in the set S. The scaled entropy ensures that impurity is always zero when the set contains only one class.

- **Gini Impurity**:

$$Gini(S) = 2p(1-p)$$

  where p is the proportion of the positive class in the set S. Like scaled entropy, the Gini criterion is used to calculate impurity before and after a split, and the best split minimizes this value.

- **Squared Impurity**:

$$\text{Squared Impurity}(S) = \sqrt{p(1-p)}$$

  where p is the proportion of the positive class in the set S. This criterion aims to minimize impurity at each split and works effectively for creating balanced partitions.

3.2 Stopping Criteria

Stopping criteria are essential for preventing overfitting by ensuring that the tree does not grow indefinitely. In this implementation, the tree growth is stopped when any of the following conditions is met:

- **max_depth:** The concept of tree depth involves monitoring the depth value of a newly created node. If the depth of the node is greater than or equal to the specified hyperparameter set during training, the node is automatically converted into a leaf. This leaf is assigned a label corresponding to the most common label among the examples routed to that node.

- **Class purity**: Splitting stops when all samples in the current node belong to the same class (i.e., the number of unique labels in the data is one). In this case, the node is deemed pure, and further division is redundant.

- **min_samples_split**: It defines the minimal number of samples that are needed to split a node. If the number of samples in a node is less than min_samples_split, the node will not be split and it will turn into a leaf node.

- **min_impurity_decrease**: The minimum decrease in impurity required for a split to occur.

These criteria help ensure that the tree is not overly complex, which could lead to overfitting, or too simple, which could lead to underfitting.

3.3 Traversal and Prediction

Once the tree is built, predictions are made by traversing the tree from the root to the leaf nodes. For each test sample, the algorithm follows the decisions made at each node based on the feature values, until it reaches a leaf node. The class label of the leaf node is then returned as the predicted label for the sample.

4 Implementation

The following section includes the detailed explanation on the actual implementation of the binary decision tree model, which consists of two primary classes: `Node` class and `Decision_Tree` class.

4.1 Node Class

The `Node` class represents a single node in the decision tree. Each node can either be a decision node or a leaf node. The constructor of the class takes several arguments:

- `feature`: The feature on which the node splits.
- `threshold`: The threshold used for splitting the data at the node.
- `left`: The left child node of the current node.
- `right`: The right child node of the current node.
- `value`: The value assigned to the leaf node (used for classification in case of a leaf).

The `is_leaf` method checks if a node is a leaf, which is the case when the `value` is not `None`.

4.2 Decision Tree Class

The `Decision_Tree` class contains the core logic for building and traversing the tree. It includes several methods for growing the tree, calculating impurity measures, and selecting the best splits.

Attributes:

- `max_depth`: Maximum depth of the tree.
- `min_samples_split`: Minimum number of samples required to split a node.
- `min_impurity_decrease`: Minimum impurity reduction required to split a node.
- `criterion`: The criterion for calculating impurity (`"scaled_entropy"`, `"gini"`, or `"squared_impurity"`).

Key Methods:

1. `fit`:

Algorithm 1 Fit Method

Require: Features X, target y, optional feature names
Ensure: Train the decision tree model

Set $n\_features$ to use all features or a subset
Call `_grow_tree(`X, y`)` to construct the decision tree

2. `_grow_tree`: Recursively grows the decision tree by splitting nodes.

Algorithm 2 Recursive _grow_tree Method

Require: Features X, target y, depth $d = 0$
Ensure: Decision tree node

$n\_samples, n\_features \leftarrow$ shape of X
$n\_labels \leftarrow$ number of unique values in y
$parent\_impurity \leftarrow$ Compute impurity based on the selected criterion (e.g., entropy, Gini, squared impurity)
if $d \geq$ max depth $\vee$ $n\_labels = 1 \vee n\_samples <$ min samples split $\vee$ $parent\_impurity <$ min impurity decrease **then**
    $leaf\_value \leftarrow$ most common label in y
    return Leaf node with $leaf\_value$
end if
$feat\_idxs \leftarrow$ randomly select a subset of features from total features
$best\_feature, best\_thresh \leftarrow$ Find the best split using _best_split(X, y, feat_idxs)
$left\_idxs, right\_idxs \leftarrow$ _split(X[:, best_feature], best_thresh)
Build left and right subtrees:
$left\_child \leftarrow \_grow\_tree(X[left\_idxs, :], y[left\_idxs], d + 1)$
$right\_child \leftarrow \_grow\_tree(X[right\_idxs, :], y[right\_idxs], d + 1)$
return Node with $best\_feature$, $best\_thresh$, $left\_child$, and $right\_child$

3. `_best_split`: Identifies the best feature and threshold for splitting the data maximizing the information gain.

Algorithm 3 _best_split Method

Require: Features X, target y, subset of feature indices $feat\_idxs$
Ensure: Optimal feature index and threshold for the split

```
Initialize variables for the best gain, split feature index, and threshold
for each feature index in feat_idxs do
    Extract the column corresponding to the feature index from X
    Determine the unique threshold values from the feature column
    for each threshold in the unique thresholds do
        Calculate the information gain using _information_gain method
        if the current gain is higher than the best gain then
            Update the best gain, split feature index, and threshold
        end if
    end for
end for
Return the feature index and threshold that provide the best gain
```

4. `_information_gain`: Calculates the information gain for a given split based on the chosen impurity measure (`scaled_entropy, gini, or squared impurity`)

   Information gain is calculated as the reduction in entropy after a split. The formula is the following:

$$\text{Information Gain} = H(S) - \left(\frac{|S_{\text{left}}|}{|S|} H(S_{\text{left}}) + \frac{|S_{\text{right}}|}{|S|} H(S_{\text{right}}) \right)$$

   where S_{left} and S_{right} are the subsets of the data after the split.

Algorithm 4 _information_gain Method

Require: Target y, X_{column}, threshold value
Ensure: Compute the information gain for a potential split

```
Calculate parent impurity using the chosen criterion (e.g., entropy, Gini, squared impurity)
Split the data into left and right subsets based on the threshold
if either left or right subset is empty then
    Return 0 as the information gain
end if
Compute the impurity for each child node (left and right) using the chosen criterion
Compute the weighted sum of child impurities based on proportion of samples in each subset
Return the difference between parent impurity and child impurities
```

5. `_traverse_tree`: Traverses the tree from root to a leaf node to make predictions for a single sample.

Algorithm 5 _traverse_tree Method

Require: Sample x, current node
Ensure: Predict the class label for a single sample
if node is a leaf (checked using `is_leaf()`) **then**
    Return the value stored in the leaf node
else
    if feature value at the node's split $\leq$ threshold **then**
        Traverse the left child (recursively call _traverse_tree on `node.left`)
    else
        Traverse the right child (recursively call _traverse_tree on `node.right`)
    end if
end if

6. `predict`: Predicts labels for an input dataset by calling `_traverse_tree` for each sample.

Algorithm 6 Predict Method

Require: Dataset X
Ensure: Predict class labels for all samples
Return a list of predictions by applying `_traverse_tree` to each sample

5 Hyperparameter Tuning and Cross Validation

Hyperparameter tuning is essential in machine learning to optimize model performance. For decision trees, once the best hyperparameters are found, they must be set before training to obtain the best performance. Without proper tuning, the model might overfit or underfit, leading to suboptimal results.

However, evaluating hyperparameter choices using a single train-test split can lead to biased or unreliable results due to variability in the data. Cross-validation (CV) is a robust technique designed to address these challenges.

Cross-validation divides the dataset into multiple subsets (or folds) and systematically trains and evaluates the model across these folds. Each subset serves as a test set once, while the remaining data is used for training. This approach ensures that the model's performance is evaluated on different portions of the data, mitigating the risk of overfitting to a specific train-test split.

By averaging performance metrics across all folds, cross-validation provides a more reliable estimate of a model's generalization ability. Additionally, it enables efficient use of the entire dataset for both training and validation, which is especially important when data is limited. This balanced evaluation frame-

work is essential for comparing different hyperparameter configurations fairly and identifying the optimal combination that maximizes predictive accuracy.

In the project a slight variation of Cross Validation was used, the **Stratified k-fold cross-validation.** This variation from the original one ensures that class distributions are preserved in each fold, making the evaluation robust and reliable.

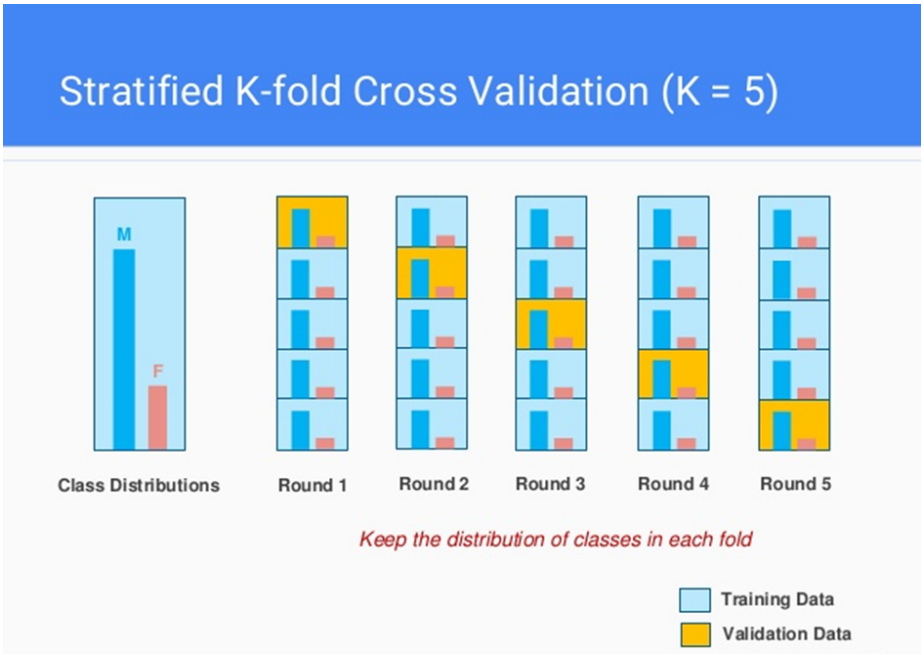


Figure 8:
`https://medium.com/@ompramod9921/cross-validation-623620ff84c2`

While grid search provides an exhaustive approach to explore all possible combinations of hyperparameters, it was found to be computationally expensive.[1]

Instead, **Randomized grid search** was used and offers an efficient alternative by sampling a subset of parameter combinations. It was actually found that Randomized grid search often outperforms standard grid search due to two key advantages:

1. *Independent Limit Allocation*: Unlike grid search, which uses fixed limits for each hyperparameter, random search adjusts limits based on the distribution of the search space, making it more effective when hyperparameters are not uniformly distributed.

2. *Easy Parallelization*: Random search enables flexible resource allocation

[1]The algorithm was still running after 6 hours, even when a smaller grid of parameters was introduced

and parallelization, improving efficiency and effectiveness, especially for longer searches where grid search may not yield better results.[2]

5.1 Implementation of Randomized Grid Search with Stratified K-Fold

To make hyperparameter tuning efficient and scalable, a custom function was implemented using randomized grid search and stratified k-fold cross-validation, called `random_search_cv`.[3]

Algorithm 7 Randomized Search with Stratified K-Fold Cross-Validation

Require: Feature matrix X, target vector y, `model_class`, `param_distributions`, `cv`, scoring metrics , `n_iter`, `n_jobs`, `random_state`, `saved_results_file`

Ensure: `best_params`, `best_score`, `results`

Initialize `best_params` $\leftarrow$ `None`, `best_score` $\leftarrow -\infty$, and load previous results if available
Define the primary metric to optimize (e.g., `accuracy`)
Generate `n_iter` random parameter combinations from `param_distributions`
for all parameter combinations **do**
    Initialize fold scores for each metric in `scoring`
    for all folds generated by `cv.split(X, y)` **do**
        Split data into training $(X_{\text{train}}, y_{\text{train}})$ and validation $(X_{\text{val}}, y_{\text{val}})$ sets
        Initialize and fit the model with parameter combinations
        Predict on X_{val} and evaluate using `scoring` metrics
        Store fold scores
    end for
    Compute mean score across folds for each metric
    if current mean score of primary metric $>$ `best_score` **then**
        Update `best_params` and `best_score`
    end if
    Append results and save to `saved_results_file`
end forreturn `best_params`, `best_score`, `results`

To make it more clearer, the hyperparameter tuning process follows these steps:

1. Define the search space, specifying ranges or possible values for each hyperparameter.

[2]Source: `https://www.geeksforgeeks.org/how-to-tune-a-decision-tree-in-hyperparameter-tuning/`

[3]"Using 3 splits in the stratified K-Fold cross-validation provided a balance between computational efficiency and model evaluation robustness. Increasing k to 5 would have imposed significant computational costs (after 7 hours the algorithm was still running) but with negligible improvement in performance metrics, making k=3 the optimal choice for hyperparameter tuning in this project.

2. Randomly sample a predefined number of hyperparameter combinations from the search space.

3. Perform stratified k-fold cross-validation for each combination:

   - Split the dataset into training and validation folds while preserving class distribution.
   - Train the decision tree on the training set with the sampled parameters.
   - Evaluate the model on the validation set using predefined metrics.

4. Track the best-performing combination based on the primary metric (e.g., accuracy).

5. Save results incrementally to avoid data loss during execution.

5.2 Evaluation Metrics

To assess the performance of the decision tree model, the following metrics are utilized:

Accuracy: The ratio of correctly predicted instances to the total instances, which provides a straightforward measure of the overall performance and for that reason it is used as primary metrics of comparison in the search for the best hyperparameters.

$$\text{Accuracy} = \frac{\text{True Positives} + \text{True Negatives}}{\text{Total Instances}}$$

Precision: The proportion of correctly predicted positive instances out of all instances predicted as positive, indicating the model's reliability in classifying positive cases:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

Recall The proportion of actual positive instances correctly predicted by the model, reflecting its sensitivity:

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

F1 Score The harmonic mean of precision and recall, balancing both metrics and providing a single measure of a model's effectiveness:

$$\text{F1} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

Accuracy alone can be misleading, especially in imbalanced datasets, where the majority class (edible mushrooms) may dominate. A model that always

predicts the majority class could still achieve high accuracy without correctly identifying the minority class (poisonous mushrooms).

That is the reason why other metrics where incorporated in the custom built `random_search_cv` function. Precision and recall are used because:

- **Precision** ensures that when the model predicts a mushroom as poisonous, it is truly poisonous, minimizing false positives.
- **Recall** ensures that as many poisonous mushrooms as possible are identified, minimizing false negatives.

Finally, the **F1 score**, as the harmonic mean of precision and recall, offers a balanced measure of both.

By incorporating accuracy, precision, recall, and F1 score, a more robust model is created given that it is not only overall accurate but also safe (minimizing false negatives) and practical (minimizing false positives).

6 Results and Analysis

Through the hyperparameter tuning done with the custom `random_search_cv` function, the best hyperparameters found after the search are as follows:

- **Max Depth**: 20
- **Minimum Impurity Decrease**: 0.01
- **Criterion**: 'entropy'
- **Minimum Samples Split**: 3

These parameters were selected based on the highest accuracy achieved during cross-validation. After determining the best parameters, the model was trained on the entire training dataset, and its performance was evaluated on both the training and test sets.

6.1 Evaluation of Model Performance

The evaluation metrics for the final model on both the training and test datasets are summarized below:

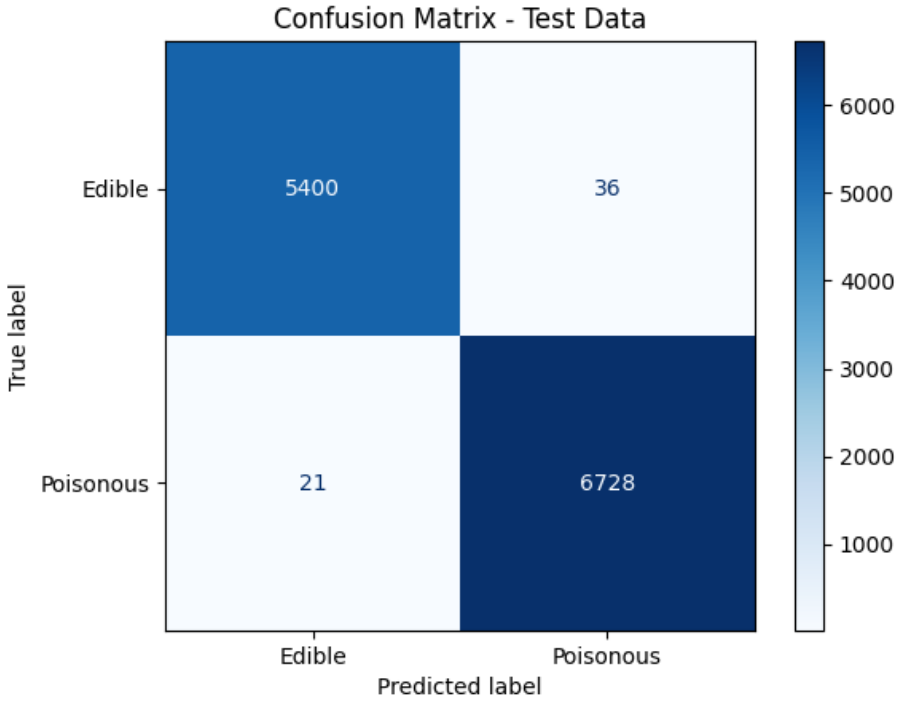
Dataset	Accuracy	Precision	Recall	F1 Score
Training	0.9979	0.9974	0.9988	0.9981
Test	0.9953	0.9947	0.9969	0.9958

Table 3: Evaluation Metrics for Training and Test Sets

The model performed well on both datasets, with high values across all evaluation metrics. Accuracy, precision, recall, and F1 scores for both datasets

are close to each other, indicating a balanced model that minimizes both false positives and false negatives.

The **confusion matrix** for the test set is presented below:



The model demonstrates excellent performance, with very few misclassifications.

Zero-one loss, which measures the proportion of incorrect predictions, was also evaluated on both the training and test datasets:

- **Training Error**: 0.00209 (0.21%)
- **Test Error**: 0.00467 (0.47%)

Quite low for both training and test datasets. This discrepancy between training and test error is normal and suggests that the model is generalizing well, with only a slight increase in error on the unseen test data.

6.2 Model Overfitting or Underfitting

To assess whether the model is overfitting or underfitting the following was considered:

- **Training vs. Test Accuracy**: The accuracy on the training dataset (99.79%) is slightly higher than that on the test dataset (99.53%). However, this difference is minimal, indicating that the model has not memorized the training data and is generalizing well to the test set.
- **Precision, Recall, and F1 Scores**: The precision, recall, and F1 scores on both the training and test sets are very close to each other, further supporting the idea that the model is not overfitting. If the model were overfitting, we would expect a larger disparity between these metrics on the training and test sets.
- **Zero-One Loss**: The low training and test error values indicate that the model is fitting the data well without overfitting. The increase in test error (from 0.21% to 0.47%) is minimal, which suggests the model's generalization is strong.

Based on these observations, the model is **not overfitting or underfitting**. Instead, it is performing well and generalizing effectively to unseen data.

7 Conclusion

In conclusion, the decision tree model built for the binary classification problem of identifying edible versus poisonous mushrooms has demonstrated very good performance. The combination of high accuracy, precision, recall, and F1 score, along with minimal errors in the confusion matrix and low zero-one loss, indicates that the model is both safe and practical for this classification task. The results show that the model is well-tuned, effectively handling both the training and test datasets without overfitting or underfitting, making it a robust solution for the problem at hand.

The full implementation of this project, can be found on my GitHub repository: `https://github.com/MinaBeric/`.

8 References

References

[1] UCI Machine Learning Repository, *Secondary Mushroom Dataset*. https://archive.ics.uci.edu/dataset/848/secondary+mushroom+dataset.

[2] GeeksforGeeks, *How to Tune a Decision Tree in Hyperparameter Tuning*, https://www.geeksforgeeks.org/how-to-tune-a-decision-tree-in-hyperparameter-tuning/.

[3] Shai Shalev-Shwartz and Shai Ben-David, *Understanding Machine Learning: From Theory to Algorithms*, Cambridge University Press, 2014.

[4] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar, *Foundations of Machine Learning (2nd edition)*, MIT Press, 2018.