Binary Classification of Mushrooms Using Decision Trees and Random Forests

Valeria Stighezza, 46021A*

Department of Computer Science "Giovanni degli Antoni", Universitá degli Studi di Milano

*Email: valeria.stighezza@studenti.unimi.it

Abstract

This study implements decision trees and random forests from scratch to classify mushrooms as edible or poisonous using the Mushroom dataset. After preprocessing to address non-relevant features and missing values, custom decision trees are built with impurity-based splitting criteria and optimized via k-fold cross-validation. Based on these models, an ensemble method is implemented by aggregating multiple decision trees through bootstrap sampling, resulting in a random forest model that achieves perfect test set accuracy, demonstrating the efficacy of decision trees and the enhanced performance achieved through ensemble methods based on them, as well as the critical role of hyperparameter tuning in identifying optimal model parameters.

Keywords: binary classification, decision tree, k-fold cross-validation, hyperparameter tuning, impurity, feature importance, random forest, bootstrap sampling, ensemble methods

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.

1. Introduction

The primary objective of this project is to develop tree-based predictors from scratch for binary classification, applied to the Mushroom dataset, to determine whether a mushroom is poisonous or edible. The Mushroom dataset, containing 61069 samples derived from 173 real species [5], provides a rich set of data for this classification challenge. This task involves constructing decision trees that utilize single-feature binary tests at internal nodes, exploring various criteria for splitting and stopping, and evaluating their performance using different metrics.

Section 7 presents a practical example of execution, showing results from two runs with different training-test splits to demonstrate the implementation's effectiveness.

Decision trees are a natural choice for this task due to their intuitive structure, which mirrors human decision-making by recursively partitioning the feature space based on single-feature tests [4]. They excel at handling both numerical and categorical data without requiring scaling, and their interpretability is useful for understanding data patterns. However, decision trees are susceptible to overfitting, especially with complex datasets, necessitating careful tuning of hyperparameters.

This report provides a comprehensive account of the project, detailing the pipeline: data preprocessing (Sec. 2), decision tree implementation from scratch (Sec. 3), hyperparameter tuning using cross-validation (Sec. 4), the construction of a random forest (Sec. 6) and the evaluation of all the implemented models (Sec. 5).

2. Data Preprocessing: Preparing the Mushroom Dataset

Effective model training begins with a clean and well-structured dataset. The Mushroom dataset requires preprocessing to address missing values, duplicates, and irrelevant features, ensuring that the subsequent models can learn meaningful patterns.

2.1 Splitting the Data

The first step involves dividing the dataset into an 80% training set (48856 samples) and a 20% test set (12213 samples) using a random shuffle, to ensure that both sets are representative of the overall distribution.

An analysis of the class distribution reveals a balanced dataset, with approximately 45% edible and 55% poisonous mushrooms. This balance eliminates the need for resampling techniques, simplifying the preprocessing workflow.

2.2 Handling Missing Values and Duplicates

The dataset contains features with significant missingness, including *stem-root*, *veil-type*, *veil-color*, and *spore-print-color*, where over 84% of values are absent. Retaining these features would introduce substantial noise, so they are dropped, reducing the feature set from 20 to 16.

For remaining features with missing values, these are substituted using the **mode** for categorical features (e.g., *gill color*) and the **median** for numerical features (e.g., *stem height*), calculated from the training set. The mode preserves categorical distributions, while the median is preferred to the mean because it is more robust to outlier effects in numerical data. The dedicated function `calculate_class_statistics()`, from the `utils` module, computes these statistics from the training set, and the function `fill_missing_values()` applies them to fill missing values in both training and test sets, ensuring consistency.

Duplicates in the training set (approx. 100 samples) are removed to prevent overfitting.

Scaling is skipped, as decision trees and random forests are scale-invariant, because they rely on relative comparisons rather than absolute distances.

This preprocessing prepares a clean dataset with 16 features, ready for model training.

3. Decision Tree Implementation: From Nodes to Full Trees

The cornerstone of this project is a custom-built decision tree classifier [1], designed to handle the Mushroom dataset's mixed feature types. The classifier has been implemented from scratch using two primary classes: `Node` and `TreePredictor`, which together define the tree's structure and behavior.

3.1 The Node Class

The `Node` class serves as the fundamental unit of the decision tree, representing either a decision point (internal node) or a terminal prediction (leaf).

Attributes:

- `feature_index` and `feature_name`: identify the feature used for splitting, facilitating interpretability.
- `threshold`: specifies the split condition, either a numerical threshold (e.g., `feature_value` ≤ 5.0) or a categorical value (e.g., `feature = "value"`).
- `left` and `right`: reference the child nodes corresponding to the outcomes of the split test.

- `leaf_value`: stores the predicted class (edible or poisonous) for leaf nodes.
- `categorical`: a boolean flag indicating whether the feature is categorical or numerical.
- additional metadata, such as `info_gain` (the reduction in impurity from the split), `most_freq_class` (the majority class at the node), and `criterion` (the splitting criterion, *"Gini", "Entropy", or "Squared Error"*), provide analytical insights.

Methods:

- `is_leaf()`: returns True if the node is a leaf (i.e., `leaf_value` is defined).
- `set_split_info()`: assigns metadata to a split node, including information gain, feature name, majority class and the splitting criterion, enabling detailed analysis of each decision point.
- `__str__`: generates a string representation of the node, detailing the feature, split condition, information gain, and majority class for decision nodes, or the predicted class for leaf nodes, distinguishing numerical ($\leq$) and categorical (=) splits.

This flexible structure allows the creation of both decision nodes, which partition the data, and leaf nodes, which deliver predictions, within a cohesive framework.

3.2 The TreePredictor Class

The `TreePredictor` class manages the tree construction and prediction.

Attributes:

- `max_depth`: tree's depth to control complexity and prevent overfitting.
- `min_samples_split`: the minimum number of samples required to split a node.
- `criterion`: the impurity measure for evaluating splits, with options including `"gini", "entropy"` or `"MSE"`.
- `root`: points to the topmost `Node` of the tree.
- `feature_importance_`: tracks each feature's contribution to impurity reduction, aiding interpretability.

Key Methods:

- `fit(X, y)`: initiates training by building the tree from the input data X (features) and labels y.
- `_grow_tree(X, y, depth)`: recursively constructs the tree, determining when to stop and how to split.
- `predict(X)`: traverses the tree for each sample in X to assign a class label.

Training begins with the `fit()` method, which takes a feature matrix X and target labels y and validates inputs, ensuring X is a DataFrame and y is a Series to handle mixed data types. Next, it extracts metadata: the feature names (*e.g., "cap-diameter"*), the number of features (16 after preprocessing), and the data type of each feature (`float` for numerical, `object` for categorical). These are stored in the class's attributes to guide subsequent operations. The method also initializes an array to track feature importance, which accumulates the contribution of each feature to impurity reduction during training.

Then, `fit()` method calls `_grow_tree()` on the training set at depth 0, starting the recursive construction of the tree, which is stored in the `root` attribute. The `_grow_tree()` method recursively builds the tree by deciding whether to create a leaf or split a node. It checks stopping conditions:

- if the **depth exceeds** `max_depth`,
- if the **number of samples is below** `min_samples_split`,
- if **all samples share the same class**,

in such cases, the method creates a leaf node with the majority class (computed via the helper method `_most_common_label`) as its predicted value.

Otherwise, the `_grow_tree()` method calculates the node's impurity using the specified criterion

- **Scaled entropy**

$$\psi(p) = \begin{cases} -\frac{p}{2}\log_2(p) - \frac{1-p}{2}\log_2(1-p) & \text{if } 0 < p < 1, \\ 0 & \text{if } p = 0 \text{ or } p = 1 \end{cases}$$

- **Gini impurity**

$$\psi(p) = 2 \cdot p \cdot (1-p)$$

- **Squared impurity**

$$\psi(p) = \sqrt{p \cdot (1-p)}$$

where term p represents the proportion of the positive class in the set of labels at a node. Then, the `_grow_tree()` method calls `_find_best_split()` to find the optimal split.

The `_find_best_split()` method evaluates each feature, testing thresholds for numerical features (*e.g., cap-diameter* ≤ 5.0) or categories for categorical ones (*e.g., gill-color = "white"*). For each potential split, it partitions samples into left and right subsets and the split's quality is computed using **information gain**, which is defined as the parent node's impurity minus the weighted average of the child nodes' impurities, where weights are the proportions of samples in each subset. Specifically, the impurity is calculated using the chosen criterion (*Scaled Entropy, Gini, Squared Error*).

$$\text{Info Gain} = \text{Impurity}_{\text{parent}} - \left(\frac{n_{\text{left}}}{n} \cdot \text{Impurity}_{\text{left}} + \frac{n_{\text{right}}}{n} \cdot \text{Impurity}_{\text{right}} \right)$$

The split with the highest gain is selected, and a `Node` is created with the feature, threshold, and categorical flag. The node's metadata, including information gain, majority class, and criterion, is set to record the split's context, and the feature's importance is updated.

`_grow_tree()` then splits the data and recurses on each subset, linking the resulting subtrees as the node's children, until one of the stopping conditions is reached.

Prediction is handled by `predict()`, which processes each sample in X using `_traverse_tree()`. This method navigates from the root, evaluating split conditions ($\leq$ for numerical, = for categorical) to follow the left or right child until reaching a leaf, returning its predicted class.

3.3 The TreeVisualizer Class

The `TreeVisualizer` class enhances interpretability by rendering the decision tree as a graphical structure, leveraging the `Graphviz` library to produce visual representations. To achieve the best resolution, the trees are saved in PDF format.

Attributes:

- `tree`: references the trained `TreePredictor` to visualize.
- `graph`: a `Graphviz Digraph` object for constructing the visualization.

Key Methods:

- `draw_tree(filename)`: generates and saves the tree visualization as a PDF file.
- `_add_nodes_edges(node, parent_name, edge_label)`: recursively adds nodes and edges to the `Graphviz` object, shaping leaf nodes as boxes and decision nodes as ellipses, with edge labels indicating split conditions (e.g., ≤ 5.0 for numerical, = "*value*" for categorical).
- `_get_node_label(node)`: creates labels for nodes, displaying the feature, threshold, and information gain for decision nodes, or the predicted class for leaf nodes.

4. Hyperparameter Tuning for Decision Trees

To ensure the decision tree generalizes well and achieves optimal performance, it's crucial to tune its hyperparameters: `max_depth`, `min_samples_split` and `criterion`.

4.1 Hyperparameter Tuning Without and With Cross-Validation

In this project, hyperparameter tuning is performed in two ways:

- **Direct Training**: for each combination of hyperparameters, the `TreePredictor` model is trained on the entire training set and evaluated on the test set, computing accuracy and 0-1 loss.
  → The hyperparameters achieving the best test set accuracy are selected.
- **K-Fold Cross-Validation** [3]: to ensure robust and reliable performance estimates, for each combination of hyperparameters, the training set is divided into $K = 5$ folds. The `TreePredictor` model is trained on 4 folds and validated on the remaining fold, with this process repeated for each fold, to obtain five validation accuracy and 0-1 loss scores. The sample indices are shuffled every time to ensure each fold reflects the dataset's diversity
  → The hyperparameters achieving the best average test set accuracy across all folds are selected as the optimal ones.

  This approach mitigates overfitting to a specific train-test split and offers an evaluation of the model behavior across different subsets of the data.

Both decision trees, one trained with the best hyperparameters from direct training and the other with those from cross-validation, are then trained on the full training set and tested on the test set to evaluate their performance, as described in Section 5.

Both trees are saved as PDFs in a graphical format using the `TreeVisualizer` class.

4.2 Grid Search for Tree Hyperparameter Tuning

Grid search is employed to evaluate all combinations of given hyperparameters under both tuning approaches. The grid of hyperparameter values is defined as:

- `max_depth` : [10, 25, 50]
- `min_samples_split` : [2, 10, 50]
- `criterion` : `["gini", "entropy", "MSE"]`

The best hyperparameters overall are determined by comparing the test set accuracy of the best configuration from direct training with the test set accuracy of the best configuration from K-fold cross-validation. If the direct training configuration achieves higher test set accuracy, it is selected as the optimal set; if the cross-validation configuration performs better, it is chosen instead. In cases where both configurations have identical test set accuracies (i.e. same best hyperparameters found), the direct training configuration is selected by default.

This approach leverages the strengths of both methods: direct training optimizes performance on the test set, while cross-validation ensures generalization across data subsets. The final selection, driven by the highest test set accuracy, maximizes the model performance.

It is obviously expected that direct training achieves higher test set accuracy, as it optimizes hyperparameters directly on the test set, in fact, beyond identifying the configuration with the highest test set accuracy, it is interesting to **examine whether both tuning methods select the same hyperparameters**. Such a convergence would suggest that the test set, used directly in the direct training approach, provides a level of generalization comparable to the multi-fold evaluation of cross-validation, offering valuable information about the test set's representativeness.

5. The Evaluation Class

The `Evaluation` class is designed to compute and visualize performance metrics for the decision tree and random forest models used in classifying mushrooms as edible or poisonous. It provides a standardized framework for evaluating model performance by calculating key metrics based on true and predicted labels from the training and test sets.

The class is initialized with true labels (`y_true`), predicted labels (`y_pred`), and optional parameters for correct and total predictions, enabling detailed performance reporting.

The `Evaluation` class computes the following metrics:

- **Accuracy**: the proportion of correct predictions, given by:

$$\text{Accuracy} = \frac{\sum_{i=1}^{n} \mathbb{1}(y_i = \hat{y}_i)}{n}$$

  where y_i is the true label, $\hat{y}_i$ is the predicted label, n is the total number of samples, and $\mathbb{1}(\cdot)$ is the indicator function (1 if true, 0 otherwise).
- **Precision**: the average precision across classes, where precision for a class c is:

$$\text{Precision}_c = \frac{\text{TP}_c}{\text{TP}_c + \text{FP}_c}, \quad \text{Mean Precision} = \frac{1}{|C|} \sum_{c \in C} \text{Precision}_c$$

  with TP_c as true positives, FP_c as false positives, and $|C|$ as the number of classes.
- **Recall**: the average recall across classes, where recall for a class c is:

$$\text{Recall}_c = \frac{\text{TP}_c}{\text{TP}_c + \text{FN}_c}, \quad \text{Mean Recall} = \frac{1}{|C|} \sum_{c \in C} \text{Recall}_c$$

  with FN_c as false negatives.
- **F1-Score**: The harmonic mean of mean precision and mean recall:

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

- **0-1 Loss**: The proportion of incorrect predictions:

$$\text{0-1 Loss} = \frac{\sum_{i=1}^{n} \mathbb{1}(y_i \neq \hat{y}_i)}{n}$$

The `print_report` and `plot_report` methods of the `Evaluation` class produce textual and graphical outputs, respectively, for a single dataset. These methods are designed to be flexible, allowing users to evaluate any dataset (e.g., training, test, or an external set).

For the specific purpose of this project, the `evaluate_and_plot_train_and_test` function was implemented to create unified visual reports that compare the performance of each model on both training and test sets within a single plot, to enhance the visual analysis of training and test errors. This function performs predictions on the training and test sets using the `predict_and_evaluate` method of the `TreePredictor` or `RandomForest` model, instantiates `Evaluation` objects for each set to compute metrics, and generates a subplot report with four panels: two confusion matrices (one for training, one for test) and two bar plots of metrics (accuracy, precision, recall, F1-score, and 0-1 loss) for both sets. The report is annotated with model parameters and the number of correct predictions.

Additional methods were implemented to streamline function execution and optimize evaluation workflows through parallel computation:

- the `fit_and_evaluate` function trains a decision tree for a given hyperparameter combination (e.g., `max_depth`, `min_samples_split`, `criterion`) and computes training and test accuracy and 0-1 loss, encapsulating these steps in a reusable procedure.
- The `plot_performance_vs_depth` function evaluates decision tree performance by varying the `max_depth` parameter, utilizing `joblib.Parallel` for efficient computation. Specifically, for each `max_depth` value, it trains and evaluates more decision trees in parallel across all logical cores, computing training and test accuracy and 0-1 loss.

6. Random Forest

Although decision trees perform well, this study aims to investigate whether predictive accuracy can be further improved through the use of ensemble techniques [2].

The `RandomForest` class builds on the `TreePredictor`, through a specialized `RandomTreePredictor` class, which extends `TreePredictor` to incorporate **feature subsampling** at each split. By combining multiple decision trees trained on **bootstrapped samples**, the random forest mitigates overfitting and captures complex patterns in the Mushroom dataset, achieving superior generalization for binary classification of mushrooms as edible or poisonous.

6.1 Random Forest Implementation

The `RandomForest` class constructs an ensemble of decision trees, each trained on a bootstrapped subset of the training data, and aggregates their predictions via **majority voting**.

Attributes:

- `n_estimators`: the number of trees in the forest, controlling the ensemble size.
- `max_depth`: the maximum depth of each tree, limiting complexity.
- `min_samples_split`: the minimum number of samples required to split a node.
- `criterion`: the impurity measure for splits, with options `"gini"`, `"entropy"`, or `"MSE"`.
- `max_features`: the number of features considered at each split, specified as `"sqrt"` (square root of total features), `"log2"`, a fixed integer or a fraction.
- `n_jobs`: the number of parallel jobs for training, with `-1` utilizing all logical cores of the machine.
- **random_state**: an integer seed that controls the randomness of bootstrap sampling and feature selection. Setting this value ensures that the random forest model is reproducible across different runs.
- `trees`: a list of trained `RandomTreePredictor` instances, stored after fitting.

Key Methods:

- `fit(X, y)`: trains the forest by constructing `n_estimators` trees in parallel, each on a bootstrapped sample of the training data X and labels y. Each tree is an instance of `RandomTreePredictor`, which modifies `TreePredictor`'s `_find_best_split` method to evaluate only a random subset of features, enhancing tree diversity. The number of features considered at each split is determined by `max_features`, typically set to $\sqrt{n_{\text{features}}}$ for `"sqrt"` (for the Mushroom dataset is $\sqrt{16} = 4$).
- `predict(X)`: predicts class labels by aggregating individual tree predictions through majority voting.
- `predict_and_evaluate(X, y)`: computes predictions, correct prediction count, total predictions, and 0-1 loss for evaluation.

Training is parallelized using `joblib.Parallel` with `n_jobs=-1`, distributing the construction of `n_estimators` trees across all logical cores of the machine, significantly reducing computation time.

For a forest with T trees, the training process can be expressed as:

$$\text{Train } T \text{ trees } \{h_t(X_b, y_b)\}_{t=1}^{T} \text{ in parallel, where } (X_b, y_b) \text{ is a bootstrapped sample.}$$

For a sample x, the forest's prediction is:

$$\hat{y}(x) = \text{mode}\{h_t(x)\}_{t=1}^{T},$$

where $h_t(x)$ is the prediction of the t-th tree, and mode selects the most frequent class (edible or poisonous). **This voting mechanism ensures robust predictions by averaging out individual tree errors.**

Feature importance is computed by averaging the `feature_importance_` scores across all trees, where each tree's score is the the cumulative information gain from splits involving each feature.

6.2 Hyperparameter Tuning for Random Forests

Hyperparameter tuning for the random forest is performed through a grid search, following the same approach used for decision trees (Section 4), to identify the optimal configuration for `n_estimators`, `max_depth`, `min_samples_split`, `criterion` and `max_features`. The process is implemented in the `perform_grid_search` function from the `hyperparameter_tuning` module.

The hyperparameter grid is defined as:

- `n_estimators`: [10, 20], controlling the number of trees.
- `max_depth`: [30, 50], limiting tree complexity.
- `min_samples_split`: [2, 10], setting the minimum number of samples required to split a node.
- `criterion`: [`"gini"`, `"entropy"`, `"MSE"`], specifying the impurity measure.
- `max_features`: [`"sqrt"`], fixing the number of features considered per split to $\sqrt{n_{\text{features}}}$.

This grid results in $2 \times 2 \times 2 \times 3 \times 1 = 24$ combinations. Each combination is evaluated using the **direct training** approach, where the model is trained on the full training set and evaluated directly on the test set. The best hyperparameter set is selected based on the highest test set accuracy.

Due to computational resource and time constraints, K-fold cross-validation was not used during hyperparameter tuning. However, as shown in the final experimental evaluation (Section 7), direct training achieves high accuracy, demonstrating that cross-validation is not essential for effective hyperparameter selection in this setting.

The grid search is parallelized using `joblib.Parallel` with `n_jobs=-1`, distributing the construction of multiple random forests and the respective evaluation of hyperparameter combinations across all available cores, significantly accelerating the search process.

For the random forest trained with the best hyperparameters, performance metrics (accuracy, precision, recall, F1-score, 0-1 loss, and confusion matrices) are computed using the `Evaluation` class. Visual reports are generated through `evaluate_and_plot_train_and_test`, producing four-panel subplots (two confusion matrices and two bar plots) for both the training and test sets, as detailed in Subsection 5. Feature importance is analyzed to identify the most relevant features, with results discussed in Section 7.

7. Experimental Evaluation

This section presents the results from two distinct executions (Run 1 and Run 2) of the decision tree and random forest classifiers applied to the Secondary Mushroom dataset. Each run uses a different training/test split due to random shuffling, allowing us to assess the robustness and consistency of the models.

The analysis compares the performance of three decision trees with different splitting criteria (entropy, Gini, and MSE), two hyperparameter-tuned decision trees (one with direct training and one with 5-fold cross-validation), plots of accuracy and 0-1 loss versus maximum depth, a standard random forest, and a random forest with hyperparameter tuning.

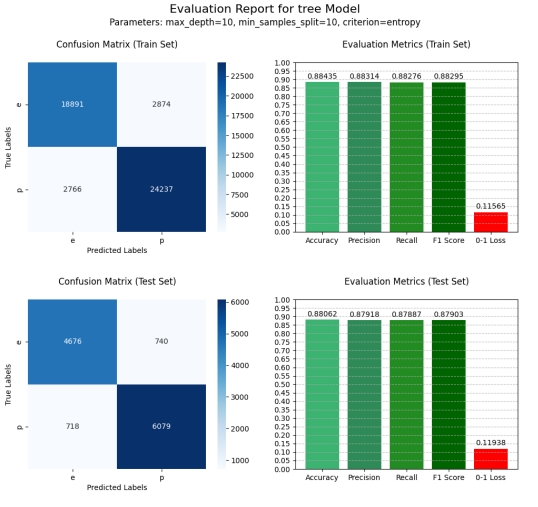
7.1 Decision Trees with Different Splitting Criteria

In both runs, three decision trees were trained with fixed `max_depth=10`, fixed `min_sample_split=10` and distinct splitting criteria: `entropy`, `Gini`, and `MSE`.

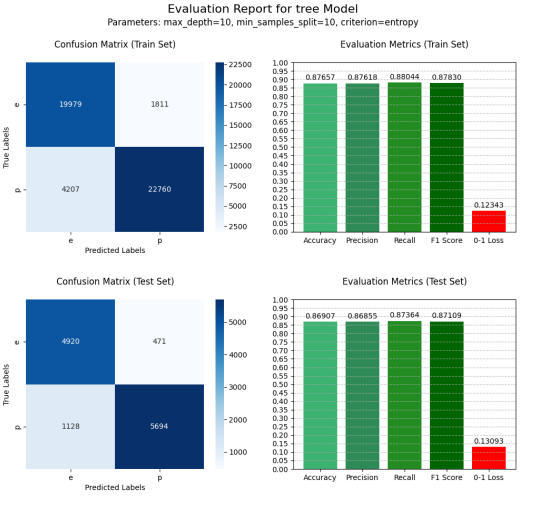
Performance metrics (accuracy, precision, recall, F1-score, and 0-1 loss) and feature importance were evaluated on both training set (top panel) and test test (bottom panel).

7.1.1 Using Scaled Entropy - Fig. 1

- Run 1
  - Top features: stem-width, cap-surface, ring-type
  - Train Accuracy: 88.43%,
  - Test Accuracy: 88.06%
- Run 2:
  - Top features: stem-width, gill-attachment, cap-surface
  - Train Accuracy: 87.65%,
  - Test Accuracy: 86.90%



(a) Run 1 details

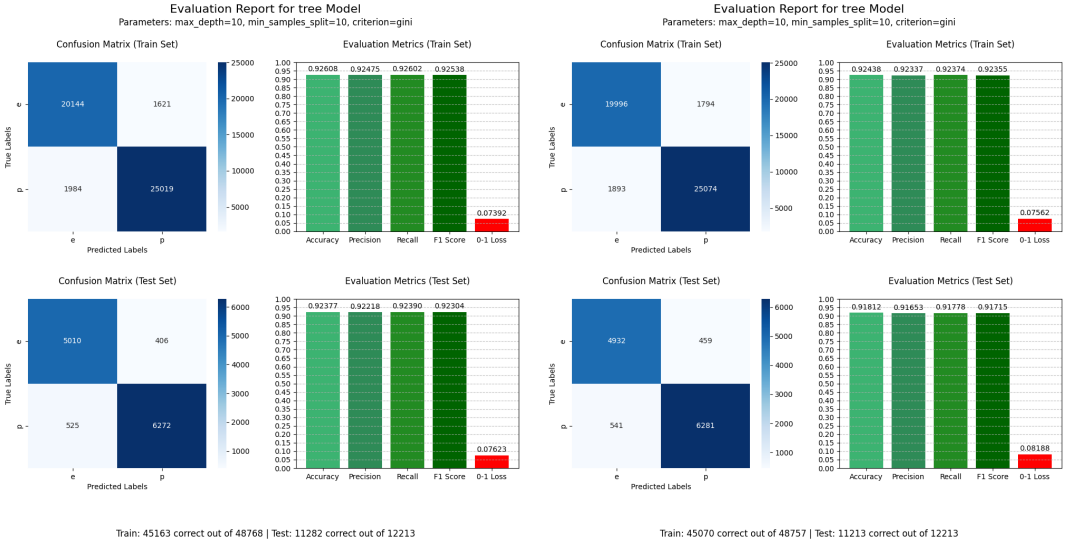


(b) Run 2 details

Figure 1. Tree - Scaled Entropy

7.1.2 *Using Gini Impurity - Fig. 2*

- Run 1 (tree at fig. 4):
  - Top features: cap-surface, gill-attachment, stem-width
  - Train Accuracy: 92.60%,
  - Test Accuracy: 92.37%
- Run 2:
  - Top features: cap-surface, stem-width, cap-diameter
  - Train Accuracy: 92.43%,
  - Test Accuracy: 91.81%



(a) Run 1 details

(b) Run 2 details

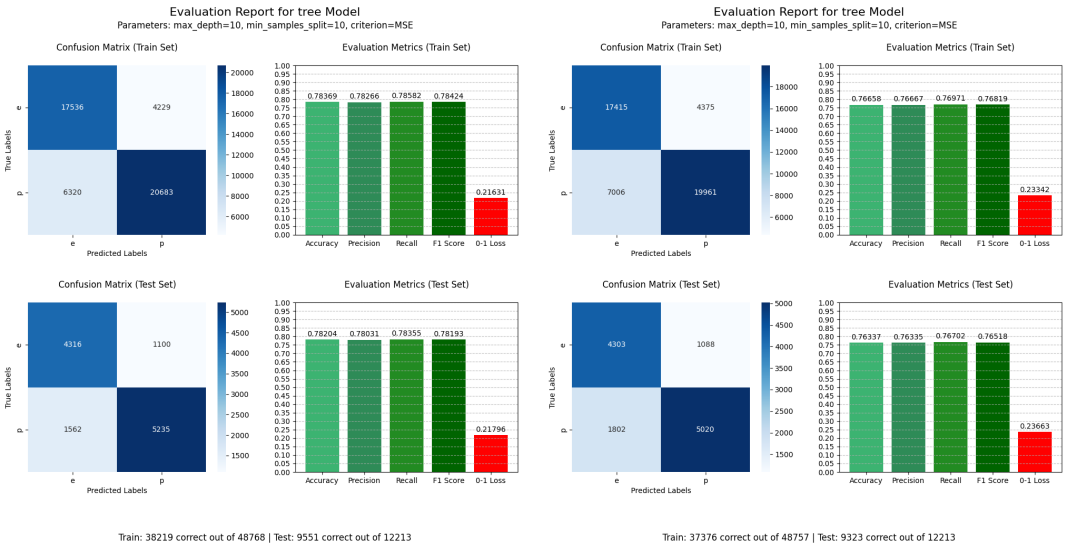
Figure 2. Tree - Gini Impurity

7.1.3 *Using Squared Impurity - Fig. 3*

- Run 1:
  - Top features: cap-surface, stem-width, stem-height
  - Train Accuracy: 78.36%,
  - Test Accuracy: 78.20%
- Run 2:
  - Top features: cap-surface, stem-width, stem-color
  - Train Accuracy: 76.65%,
  - Test Accuracy: 76.37%

7.1.4 *Decision Trees with Different Splitting Criteria - Analysis of Results*

With a low depth (`max_depth=10`) and small minimum sample split (`min_samples_split=10`), Gini impurity emerges as the best-performing criterion across both runs, consistently achieving the highest accuracy on both training and test sets. It outperforms scaled entropy, which achieves



(a) Run 1 details **(b)** Run 2 details

Figure 3. Tree - Squared Impurity

moderate results, and squared impurity, which suggests it's less effective for this task under these constraints.

In Run 2, all methods do slightly worse, suggesting the data split matters a bit.

The features *cap-surface* and *stem-width* are always among the three most important ones across methods and runs.

Each tree was saved graphically as a PDF in the `imgs` directory, with an example shown in Fig. 4.

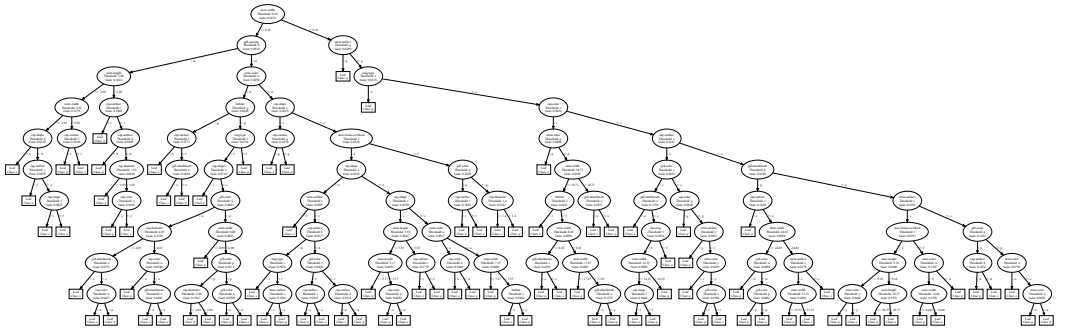


Figure 4. Graphical Tree Example - Gini Impurity

7.2 Hyperparameter-Tuned Decision Trees

Hyperparameter tuning was conducted using grid search, comparing direct training (no cross-validation) and 5-fold cross-validation.

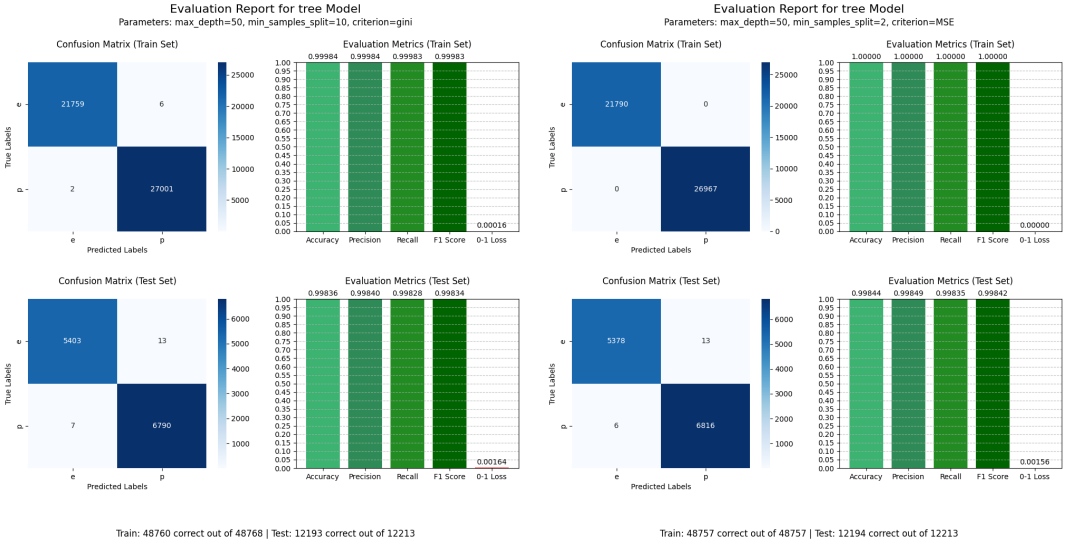
Both tuned trees were saved graphically as a PDF in the `imgs` directory.

7.2.1 *Using Direct Training - Fig. 5*

- Run 1
  - Best hyperparameters: `max_depth = 50`, `min_sample_split = 10`, `criterion = gini`
  - Top features: gill-attachment, cap-shape, cap-surface
  - Train Accuracy: 99.98%,
  - Test Accuracy: 99.84%
- Run 2:
  - Best hyperparameters: `max_depth = 50`, `min_sample_split = 2`, `criterion = MSE`
  - Top features: cap-surface, gill-attachment, cap-diameter
  - Train Accuracy: 1.00%,
  - Test Accuracy: 99.84%

7.2.2 *Using 5-Fold Cross Validation - Fig. 6*

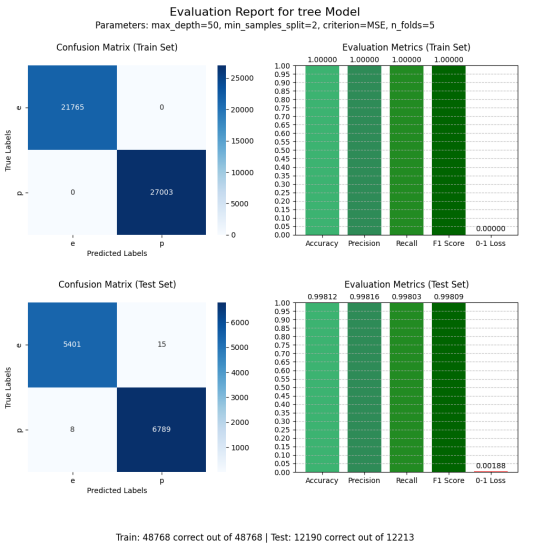
- Run 1
  - Best hyperparameters: `max_depth = 50`, `min_sample_split = 2`, `criterion = MSE`
  - Top features: cap-diameter, cap-surface, stem-width
  - Train Accuracy: 1.00%,
  - Test Accuracy: 99.81%
- Run 2:
  - Best hyperparameters: `max_depth = 50`, `min_sample_split = 2`, `criterion = MSE`
  - Top features: cap-surface, gill-attachment, cap-diameter
  - Train Accuracy: 1.00%,
  - Test Accuracy: 99.84%



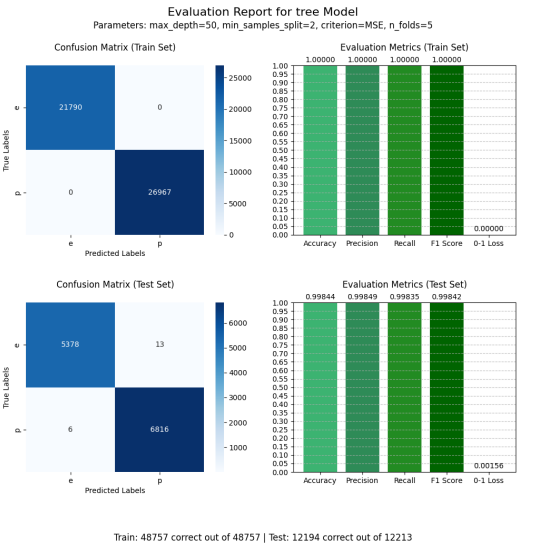
(a) Run 1 details

(b) Run 2 details

Figure 5. Tuned Tree - Direct Training



(a) Run 1 details



(b) Run 2 details

Figure 6. Tuned Tree - 5-Fold Cross Validation

7.2.3 Hyperparameter-Tuned Decision Trees - Analysis of Results

Both direct training and 5-fold CV achieve good performance, with test accuracies ranging from 99.81% to 99.84% across runs. Direct training slightly outperforms CV in Run 1 (99.84% vs. 99.81%) and matches it in Run 2 (99.84%), suggesting that direct training may better exploit split-specific patterns, while CV ensures robust generalization. The high training accuracies (99.98% to 100%) indicate potential overfitting, but test results confirm strong generalization.

Hyperparameters: all models select `max_depth=50`.

- In Run 2, direct training and CV align perfectly (`min_samples_split=2, criterion=MSE`).
- In Run 1, they differ: direct training uses `min_samples_split=10` and `gini`, while CV picks `min_samples_split=2` and `MSE`, with `min_samples_split` remaining low (≤ 10).

CV shows consistent hyperparameters across runs, unlike direct training.

Key features (*cap-surface, gill-attachment*) vary slightly in ranking but remain prominent.

It is worth noting that in the initial evaluation of decision trees with fixed parameters (`max_depth = 10`, `min_samples_split = 10`), *MSE was the worst-performing impurity measure*, significantly underperforming both Gini and scaled entropy in terms of accuracy. However, in the hyperparameter-tuned setting, where depth increases (up to 50) and the minimum sample split is reduced (as low as 2), *MSE emerges as the most effective criterion*, consistently selected by grid search in both direct training and cross-validation. This shift suggests that MSE, although less effective under restrictive model settings, may better capture impurity patterns when the tree structure is allowed to grow deeper and split more. In this context, **hyperparameter tuning not only improves performance but also reshapes the relative effectiveness of impurity measures.**

7.3 Accuracy and 0-1 Loss vs. Max Depth

Plots of accuracy and 0-1 loss versus maximum depth (varying from 2 to 50, step 3) were generated using the best hyperparameters from direct training.

Both experimental runs (fig. 7 and 8) reveal a clear saturation point in model performance with respect to tree depth. In the initial range, increasing `max_depth` leads to substantial improvements in both training and test accuracy, as well as a rapid reduction in 0-1 loss. However, in Run 1, starting from a `max_depth` of 32, and in Run 2, from depth 35, neither the training error nor the test error shows further improvement. This indicates that the model has already captured the relevant data structure, and increasing the depth beyond these values brings no additional benefit. Deeper trees only increase model complexity without gains in accuracy.

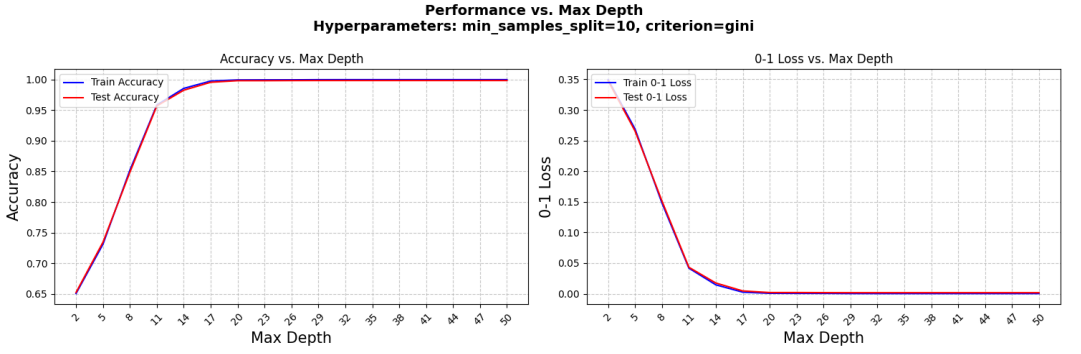


Figure 7. Run 1 - Accuracy and 0-1 Loss vs. Max Depth

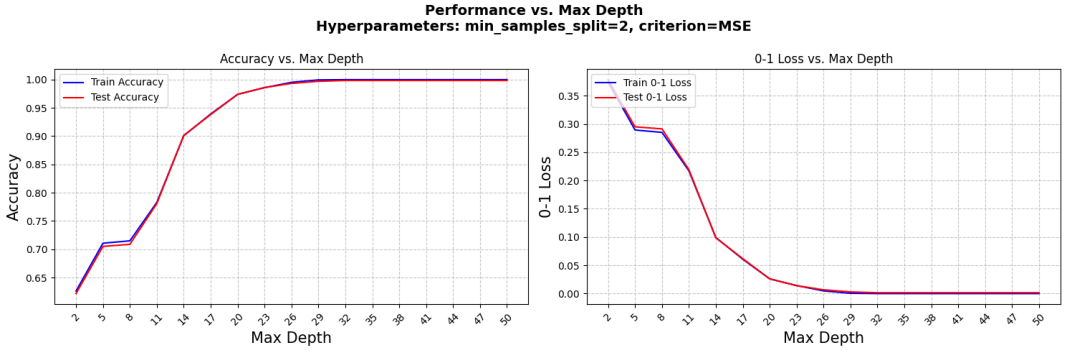


Figure 8. Run 2 - Accuracy and 0-1 Loss vs. Max Depth

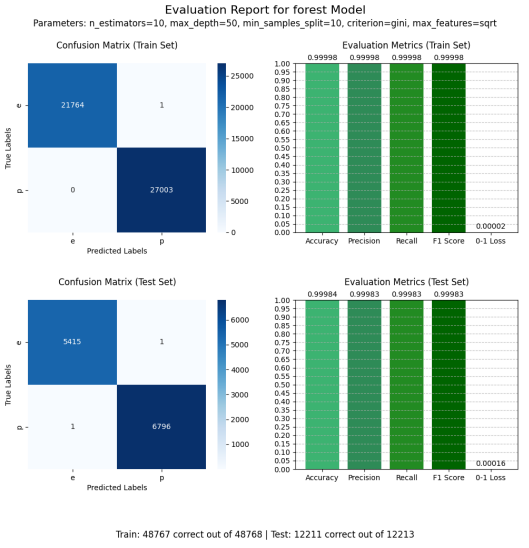
The plots in fig. 7 and 8 show that training and test metrics remain closely aligned across all depths. This tight correspondence indicates strong generalization, with no evident overfitting even at higher depths.

7.4 Random Forest - Fig. 9

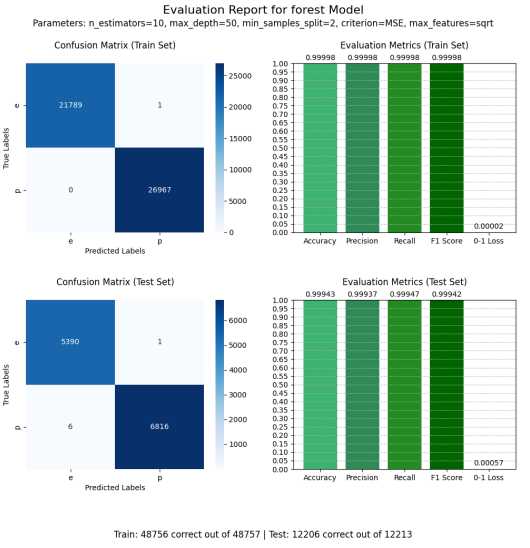
A random forest was trained using the best hyperparameters identified from decision tree tuning (`max_depth=50, min_samples_split` and `criterion`), with default settings `n_features=sqrt` and `n_estimators=10`. Performance metrics and feature importance were evaluated.

- Run 1:
  - Top features: stem-width, cap-surface, gill-attachment
  - Train Accuracy: 99.99%,
  - Test Accuracy: 99.98%

- Run 2:
  - Top features: stem-width, gill-attachment, stem-height
  - Train Accuracy: 99.99%,
  - Test Accuracy: 99.94%



(a) Run 1 details



(b) Run 2 details

Figure 9. Random Forest

7.4.1 Random Forest - Analysis of Results

The random forest achieves near-perfect performance, with test accuracies of 99.98% in Run 1 (2 errors) and 99.94% in Run 2 (7 errors), significantly outperforming single decision trees due to its ensemble approach.

Key features like *stem-width* and *gill-attachment* dominate in both runs, though Run 2 prioritizes *stem-height* over *cap-surface*, reflecting split-specific feature interactions.

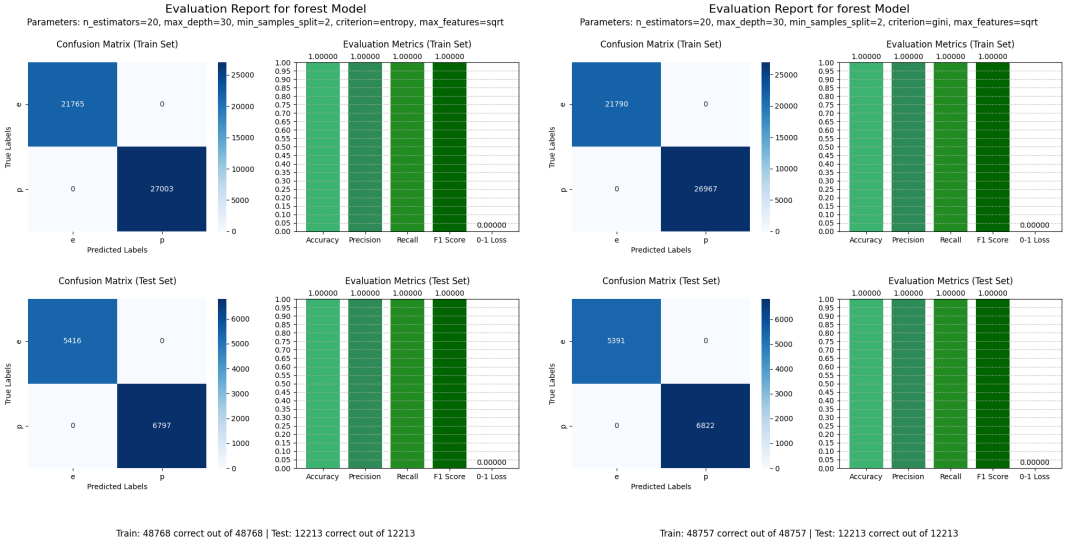
The slightly lower accuracy observed in Run 2 suggests either a more challenging train-test split or suboptimal hyperparameters. To address this, hyperparameter tuning is performed specifically for the random forest, with the goal of improving performance and potentially eliminating residual errors.

7.5 Hyperparameter-Tuned Random Forest - Fig. 10

A grid search optimized the random forest's hyperparameters, selecting the best configuration based on test accuracy.

- Run 1:
  - Best hyperparameters: `n_estimators = 20`, `max_depth = 30`, `min_sample_split = 2`, `criterion = entropy`, `max_features = sqrt` $= \sqrt{16} = 4$
  - Top features: stem-width, gill-attachment, cap-surface
  - Train Accuracy: 1.00%,
  - Test Accuracy: 1.00%

- Run 2:
  - Best hyperparameters: `n_estimators = 20`, `max_depth = 30`, `min_sample_split = 2`, `criterion = gini`, `max_features = sqrt` $= \sqrt{16} = 4$
  - Top features: stem-width, cap-surface, gill-color
  - Train Accuracy: 1.00%,
  - Test Accuracy: 1.00%



(a) Run 1 details

(b) Run 2 details

Figure 10. Tuned Random Forest

7.5.1 Hyperparameter-Tuned Random Forest - Analysis of Results

The hyperparameter-tuned random forest achieves perfect performance, with 100% train and test accuracy in both Run 1 and Run 2, eliminating the residual errors seen in the previous random forests (2 errors in Run 1 and 7 in Run 2) and indicating that tuning mitigates split-specific challenges. This improvement highlights the **effectiveness of performing a dedicated grid search directly on the random forest**, as opposed to relying on the hyperparameters initially optimized for the decision tree.

Hyperparameter Impact: increasing `n_estimators` from 10 (fig. 9) to 20 (fig. 10) enhances ensemble robustness, addressing the previous challenging random forest (99.94% accuracy), while `max_depth=30` (less than 50) may prevent unnecessary complexity and overfitting.
The `criterion` (potentially shifting between gini and entropy) shows no substantial impact, suggesting that the choice of impurity measure is less critical for this dataset, while a moderate `max_depth` and a low `min_sample_split` are most relevant hyperparameters.

Feature importance shifts slightly between Run 1 and Run 2, but confirm the dominance of *stem-width* and *cap-surface*. The perfect performance in Run 2, despite its prior 7 errors, highlights the **tuning's success in generalizing across splits**.

The perfect performance in Run 2, despite its prior 7 errors, highlights the tuning's success in generalizing across splits.

8. Conclusion

This study demonstrates the effectiveness of custom-built tree-based models for classifying mushrooms as edible or poisonous using the Secondary Mushroom dataset.

Initial decision trees with **fixed parameters** (`max_depth=10, min_samples_split=10`) reveal **Gini impurity as the best splitting criterion**, achieving test accuracies of 92.37% (Run 1) and 91.81% (Run 2), outperforming scaled entropy (88.06%, 86.90%) and squared impurity (MSE), which performs poorly at 78.20% and 76.37%.

Notably, MSE, the least effective criterion in these shallow trees, shows a significant improvement in preformance after hyperparameter tuning. **With deeper trees** (`max_depth=50`) **and smaller splits** (`min_samples_split=2` or 10), **MSE becomes the dominant choice** in three out of four tuned decision tree models (test accuracies: 99.81%, 99.84%), highlighting its strength in capturing complex patterns when given sufficient flexibility.

However, no further performance gains are observed beyond `max_depth=32-35`, as shown in accuracy and 0-1 loss plots, indicating that **excessive depth adds complexity without benefit**.

The standard random forest, using decision tree-tuned hyperparameters, achieves test accuracies of 99.98% (Run 1, 2 errors)and 99.94% (Run 2, 7 errors), showcasing the power of ensemble methods.

Hyperparameter tuning for the random forest eliminates all errors, achieving 100% accuracy in both runs by increasing `n_estimators` to 20 and reducing `max_depth` to 30, demonstrating the critical role of dedicated ensemble tuning.

Features like *stem-width, cap-surface and and gill-attachment* consistently drive predictions across models and runs.

These results affirm the robustness of tree-based models, the necessity of hyperparameter tuning, and the superiority of random forests for achieving perfect classification on this dataset.

References

[1] Cesa-Bianchi N *Tree Predictors*. https://cesa-bianchi.di.unimi.it/MSA/Notes/treepred.pdf. Lecture notes for the course "Statistical Methods for Machine Learning", University of Milan. 2024.

[2] Google Developers *Decision Forests | Machine Learning - Google for Developers*. https://developers.google.com/machine-learning/decision-forests. 2025.

[3] Mehryar Mohri AR and Talwalkar A *Foundations of Machine Learning*, 2nd ed. MIT Press, 2018. Available at https://cs.nyu.edu/~mohri/mlbook/.

[4] Shalev-Shwartz S and Ben-David S *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014. Available at https://www.cs.huji.ac.il/~shais/UnderstandingMachineLearning/.

[5] Wagner D, Heider D and Hattab G Mushroom data creation, curation, and simulation to support classification tasks. *Scientific Reports* **11** (2021). https://api.semanticscholar.org/CorpusID:233241809.

Appendix 1. File Organization

The code is structured into modular Python files, each handling a specific component of the binary classification task for the Secondary Mushroom dataset.

Below is a brief overview of the files and their key functionalities:

- `utils.py`:
  Purpose: manages data preprocessing tasks.
  Key Functions:

  - `split_train_test`: splits the dataset into n% training and m% test sets with optional shuffling.
  - `calculate_class_statistics`: computes medians (numerical) and modes (categorical) for missing value imputation.
  - `fill_missing_values`: imputes missing values using medians and modes.
  - `NaN_summary`: summarizes missing values in datasets.
  - `reset_indices`: resets DataFrame indices after splitting.
- `models.py`:
  Purpose: implements decision tree and random forest classifiers, along with visualization.
  Key Classes and Functions:
  - `Node`: represents tree nodes (split or leaf) with attributes like `feature_index, threshold` and methods like `is_leaf()`.
  - `TreePredictor`: builds decision trees with methods like `fit()`, `predict()`, and `_grow_tree()`, supporting criteria (entropy, Gini, MSE).
  - `RandomTreePredictor`: extends TreePredictor for feature subsampling.
  - `RandomForest`: constructs an ensemble of trees with `fit()` (parallelized) and `predict()` (majority voting).
  - `TreeVisualizer`: renders trees as PDFs using `Graphviz`.
  - `print_feature_importance`: displays sorted feature importance for trees and forests.
- `hyperparameter_tuning.py`:
  Purpose: performs grid search for hyperparameter optimization.
  Key Functions:
  - `kfold`: generates indices for k-fold cross-validation.
  - `evaluate_model`: evaluates a model for given hyperparameters.
  - `perform_grid_search`: conducts grid search with optional cross-validation, parallelized for trees and random forests.
- `evaluation.py`:
  Purpose: evaluates and visualizes model performance.
  Key Classes and Functions:
  - `Evaluation`: computes metrics (*accuracy, precision, recall, F1-score, 0-1 loss*) with methods like `print_report()` and `plot_report()`.
  - `evaluate_and_plot_train_and_test`: generates unified reports with confusion matrices and metric plots.
  - `fit_and_evaluate`: trains and evaluates trees for specific hyperparameters.
  - `plot_performance_vs_depth`: plots accuracy and 0-1 loss versus `max_depth`, using parallel computation.
- `main.py`:
  Purpose: executes the pipeline, coordinating preprocessing, model training, tuning, evaluation, and visualization.
  Key Functionality:
  - Loads the Mushroom dataset (UCI or local CSV).
  - Executes preprocessing (splitting, dropping high-missingness columns, imputing, removing duplicates).
  - Trains and evaluates decision trees (entropy, Gini, MSE), hyperparameter-tuned trees (direct and 5-fold CV), and random forests (standard and tuned).
  - Saves visualizations (graphical trees and their performance plots) in the `imgs` directory.

The `imgs` directory stores graphical outputs, such as decision tree PDFs and evaluation plots, organized by run.