Mushroom Decision Tree Project

Arianna Girotto

17 February 2025

1 Introduction

In this work, it will be analyzed a dataset containing 61,069 hypothetical mushrooms, with caps generated based on 173 real species (with 353 mushrooms per species). Each mushroom is classified as definitely edible or definitely poisonous.

The main goal of this study is to develop from scratch a Decision Tree and a Random Forest, along with their respective cross-validation techniques. The methodology implemented in this paper follows the structure below:

- **Preprocessing** – Importing necessary libraries, loading the dataset, splitting it into training and test sets, and performing data cleaning
- **Class NodeTree** – Defining the structure of tree nodes
- **Definition of the functions splitting criteria and best split** – Establishing the splitting criteria and identifying the best split
- **Class DecisionTree** – Implementing the decision tree algorithm
- **Definition of the function for cross-validation** – Defining the cross-validation technique
- **Class RandomForest** – Implementing the random forest algorithm
- **Definition of cross-validation** – Applying cross-validation to the random forest

This structure allows to evaluate the performance of the developed models and compare their effectiveness in determining the edibility of hypothetical mushrooms.

2 Preprocessing

The dataset was processed through a series of data cleaning and preparation steps to ensure an accurate and unbiased analysis. The main steps applied were: loading the dataset, splitting the total dataset into training and test sets, data cleaning, and separating the dependent variable from the independent ones.

First, the data was split into training (80%) and test (20%) sets using a random seed of 42 to ensure reproducibility. Only after this step it was performed the data cleaning to avoid data leakage and to ensure that the test set remained as separate as possible from the training set.

The methodology used to clean the training set is composed of three steps: the percentage of missing values per column was calculated and the ones with more than 40% missing values were dropped. The rows containing the remaining missing values were dropped and in the end the duplicate entries were eliminated. As a result, the training set was reduced from an initial size of (48,855 rows $\times$ 21 columns) to a final size of (29,664 rows $\times$ 15 columns).

After applying these transformations to the training set, the same operations were consistently applied to the test set: this approach ensures that the model is trained and tested on data with the same structure and characteristics, without introducing bias from the test set during the data cleaning process.

As a next step, it was verified that the train-test ratio remained stable after the data cleaning process and since the new proportion is [Train: 80.22%, Test: 19.78%], we can confirm that the balance was maintained between the two sets.

Continuing with the data preprocessing, it was performed a separation between the target variable ("class") and the features. The "class" column explicitly indicates whether a mushroom is edible or poisonous, while the remaining 14 columns contain the mushroom's characteristics.

Since the dataset analysis revealed that some columns are numerical (int64 or float64) while others are categorical (object), these were identified and stored in separate variables to allow for appropriate handling within the decision tree.

Finally, to ensure balance between the training and test sets, the distribution of the target variable was analyzed in both subsets: the distributions were found to be very similar (approximately 46% vs. 54%), confirming that the model can now be trained effectively.

3 Implementation of the Decision Tree

3.1 Definition of the Decision Tree Structure: the "TreeNode" Class

The first part of the implementation involves creating the TreeNode class, which represents the fundamental element of the decision tree structure. Each node can have two distinct roles: it can be either an internal node, responsible for splitting the data, or a leaf node, which is a terminal node that provides a prediction.

To handle these two cases, the TreeNode class has been designed with several key attributes:

- `is_leaf`: A boolean value indicating whether the node is a leaf or not.
- `test`: Represents the split condition for internal nodes, defined as a pair (feature, threshold).
- `left` e `right`: Pointers to the left and right child nodes, allowing for the recursive construction of the tree.
- `prediction`: The predicted class value, used exclusively in leaf nodes to return a final result.

This structure enables the hierarchical organization of data and allows decisions to be made by iterating through the tree, following the split conditions until a terminal node is reached.

3.2 Implementation of the Impurity Functions

After defining the basic structure of the decision tree, the necessary functions are implemented to calculate the splitting criterion, which is the measure used to determine the quality of a data split. In this implementation, three different methods have been considered to compute node impurity, each with specific characteristics.

The first method is the `Gini` impurity, defined by the formula:

$$G(y) = 2p\,(1-p)$$

where p represents the proportion of instances belonging to the positive class. This measure quantifies the probability that an instance is misclassified if assigned randomly based on class distribution. The higher the value, the greater the mixing of classes, indicating a less effective split.

The second method is `Psi3`, the scaled entropy, calculated as:

$$\Psi_3(y) = -2p\log_2(p) - 2\,(1-p)\log_2(1-p)$$

This function is similar to classical entropy but has a different scaling factor, which can make the criterion more sensitive in certain situations.

Finally, `Psi4` is introduced, a metric based on the square root, defined as:

$$\Psi_4(y) = \sqrt{p(1-p)}$$

To ensure flexibility in choosing the impurity criterion, the `select_impurity_function` function is implemented, allowing the selection of the desired method dynamically based on a specified parameter. This way, the model can be adapted to the specific needs of the problem at hand.

3.3 Dataset Division: `split_dataset` Function

After defining the impurity criteria, a function is implemented to split the dataset into two subsets based on a specific feature and a predefined threshold. This step is crucial for constructing the decision tree, as it determines how the data is separated at each node.

The splitting logic varies depending on the type of variable being considered:

- For *categorical variables*, the split is performed by comparing the feature value to a specific category. All instances that match this value are assigned to one subset, while the remaining instances go into the other.
- For *numerical variables*, the split is based on a threshold: instances with values less than or equal to the threshold are assigned to one subset, while those with values greater than the threshold are placed in the other.

This distinction between *categorical and numerical variables* ensures that the data is correctly separated based on its nature, allowing the tree to learn effectively and adapt to different types of datasets.

3.4 Choosing the Best Split: `best_split` Function

This function is responsible for identifying the best split condition that divides the dataset optimally, minimizing the impurity in the resulting subsets. The process unfolds through a series of systematic steps: at first for each feature in the dataset, the function explores the possible split thresholds.

For ***categorical features***, it considers all unique values as potential split thresholds. Using a membership test, the data is split based on whether each instance belongs to a specific category. This way, the data is divided according to the categories present in the feature.

For ***numerical features***, the function selects specific percentiles (the 25th, 50th, and 75th percentiles) as candidate thresholds. This approach helps avoid considering too many possible splits, focusing instead on meaningful thresholds that might be useful for separating the data effectively. For each candidate threshold, the dataset is split into **two subsets (left and right)**. If either subset turns out to be empty, that threshold is immediately discarded, as it wouldn't be useful for creating a meaningful separation.

Next, the weighted impurity for each split is calculated: impurity is measured based on the proportions of examples in each subset and their respective impurity, depending on the selected impurity function.

The best split is the one with the **lowest impurity**, meaning the separation that results in the most "pure" subsets.

This function plays a crucial role in the construction of the decision tree, as it determines the optimal data division at each node. By selecting the best splits, it progressively reduces the impurity, leading to the creation of the **terminal leaf nodes**.

3.5 Recursive Construction of the Tree: `DecisionTree` Class

After defining the impurity criteria, the `DecisionTree` class is implemented, which is at the core of both the construction and use of the decision tree. It defines the parameters, methods for building the tree, and methods for making predictions on the data.

Class Parameters

The `DecisionTree` class allows for the configuration of various parameters that influence the tree construction:

- `max_depth`: defines the maximum depth of the tree. This parameter helps prevent overfitting by imposing a limit on the maximum depth of the nodes. Once the tree reaches this depth, no further splits will be made.
- `min_samples_split`: sets the minimum number of instances required for a split to be considered. If the number of instances in a node is lower than this value, the split will not be performed, and the node will become a leaf.
- `impurity_type`: defines the type of impurity criterion to use for evaluating node splits. It can be `Gini`, `Psi3`, or `Psi4`, depending on the user's choice.

Tree Construction (fit and _build_tree)

The `fit(X, y)` method is the starting point for constructing the tree. When called, it invokes the recursive method `_build_tree(X, y, depth)` to create the decision tree: if all examples in the node belong to the same class, or if the number of instances is below `min_samples_split`, or if the maximum depth of the tree has been reached, a leaf node is created.

Otherwise, the best possible split is identified using the functions defined earlier. The tree is then recursively built on the left and right subsets to expand the nodes and create a branching structure that progressively divides the data.

Prediction (`predict` and `_predict_single`)

Once the tree is trained with the fit method, it can be used to make predictions on the data. The `predict(X)` method takes an input dataset, and for each example, it calls `_predict_single(x, node)`, which traverses the tree from the root to a leaf node, returning the predicted class for each example.

The `_predict_single(x, node)` method recursively traverses the tree, following the split conditions until it reaches a leaf. The class of the leaf node is then returned as the prediction for the example.

Performance Evaluation (accuracy and error)

To assess the quality of the model, two main methods are implemented:

- `accuracy(X, y)` calculates the percentage of correct predictions. This value provides a direct measure of the model's ability to classify examples correctly in the test set.
- `error(X, y)` calculates the error rate, which is the complement of accuracy, i.e., 1 minus accuracy. This value represents the fraction of incorrect predictions made by the model.

Both of these methods use 0-1 loss, which assigns a cost of 1 for each classification error (when the prediction differs from the true class) and 0 for each correct classification. This makes the calculation of error and accuracy intuitive, as it only depends on whether the model correctly predicted the class or not.

In summary, the `DecisionTree` class provides a complete implementation of a decision tree, where the tree is constructed recursively, and each node is split based on a selected impurity criterion. The built model can then be used to make predictions on the data, and performance is measured using accuracy and error rate calculated with 0-1 loss.

Cross-Validation Procedures: `cross_validation`

After implementing the decision tree, the next step was to implement the cross-validation procedure, essential for selecting the best hyperparameters and evaluating the model's performance reliably.

In this paper was implemented the basic Cross-Validation: this technique involves splitting the dataset into k-folds and the idea is that for each combination of hyperparameters, the model is trained on k-1 subsets of the data and tested on the last subset, known as the test fold, which for each were calculated accuracy and error (based on 0-1 loss).

At the end of the process, the performance averages over all folds are calculated, providing an overall evaluation for each combination of hyperparameters. This process provides a more robust estimate of the model's performance, reducing the risk of overfitting and offering useful insights for selecting the best parameters.

4 Results

In this section of the paper, the results of the implemented algorithm will be presented, analyzing its performance through different stages.

Initially, a decision tree with random parameters is trained on the training set and evaluated on the test set to obtain an initial estimate of its performance. This preliminary phase allows us to observe the model's behavior before optimization.

Subsequently, cross-validation is applied, enabling the exploration of different hyperparameter combinations and selecting the one that leads to the lowest test error: once the best hyperparameters are identified, the model is retrained with these optimal values and tested again.

This entire process ensures that the algorithm is as performant as possible, providing a good balance between predictive capability and generalization, while minimizing the risk of overfitting.

4.1 Creation and Initial Evaluation of the Decision Tree

In this initial phase, a first `DecisionTree` model is created with default parameters:

- `Maximum Depth (max_depth): 10`
- `Minimum Samples for Split (min_samples_split): 2`
- `Impurity Criterion: "gini"`

The model is trained on the training data using the method `tree.fit(X_train, y_train)`. Then, the model's performance is calculated on both the training set and the test set, yielding the following results:

- `Train Accuracy: 0.9500`
- `Train Error: 0.0500`
- `Test Accuracy: 0.9530`
- `Test Error: 0.0470`

These values provide an initial evaluation of the model's performance, offering a benchmark before the optimization phase. In particular, it can be observed that both accuracies are very high, with the test accuracy slightly higher than the training accuracy. This suggests that no overfitting or underfitting issues are evident from this initial analysis.

4.2 Optimization of the Decision Tree with Cross-Validation

To improve the model's performance and enable better generalization, a hyperparameter grid (`param_grid_dt`) was defined with the following options:

- `Max Depth: 10, 15`
- `Min Samples Split: 2, 10`
- `Impurity Criterion: "gini", "psi3", "psi4"`

Subsequently, a standard cross-validation (k=5) was executed using the function described earlier. The input features and response variable from the training set were passed, and the training set was divided into 5 folds.

The results were sorted by test error (from the bigger to the smaller) to quickly and intuitively identify the best hyperparameter combination. In Table 1, it can be seen the decision trees that were executed:

max_depth	min_samples_split	impurity	max_features	mean_train_accuracy	mean_test_accuracy	mean_train_error	mean_test_error
10	2	psi4	14	0.849713	0.849225	0.150287	0.150775
10	10	psi4	14	0.849713	0.849225	0.150287	0.150775
10	2	psi3	14	0.883078	0.878894	0.116922	0.121106
10	10	psi3	14	0.883052	0.878894	0.116948	0.121106
15	10	psi4	14	0.941842	0.938065	0.058158	0.061935
15	2	psi4	14	0.941952	0.938166	0.058048	0.061834
10	2	gini	14	0.952486	0.951281	0.047514	0.048719
10	10	gini	14	0.952385	0.951281	0.047615	0.048719
15	10	psi3	14	0.965161	0.962171	0.034839	0.037829
15	2	psi3	14	0.965405	0.962239	0.034595	0.037761
15	10	gini	14	0.994564	0.993088	0.005436	0.006912
15	2	gini	14	0.995053	0.993392	0.004947	0.006608

Table 1: Cross-Validation Results (Sorted by Test Error)

and it can be immediately evident that the best tree is the one with the following parameters

- `Max Depth: 15`
- `Min Samples Split: 2`
- `Impurity Type: gini`

Thus, using these hyperparameters, a final model was created, retrained on the entire training dataset, and tested on the main test set.

The final performance results are:

- `Train Accuracy: 0.9875`
- `Train Error: 0.0125`
- `Test Accuracy: 0.9884`
- `Test Error: 0.0116`

It is immediately noticeable that the results after optimization are significantly improved, especially with the reduction in test error, which enhances the tree's ability to generalize. This also confirms the absence of overfitting or underfitting.

5 `RandomForest` Class

The `RandomForest` class implements a random forest model for classification, using the decision tree algorithm that was implemented earlier in the paper. In addition to the class itself, a cross-validation function has been implemented, using parallelization to speed up the process.

`RandomForest` Class

The RandomForest class is initialized with several parameters:

- `n_trees`: the number of decision trees to include in the forest.

- `max_depth`: the maximum depth of each tree.
- `min_samples_split`: the minimum number of samples required to split a node.
- `impurity_type`: the impurity criterion used to determine the quality of splits in nodes; here, the splitting criteria described previously are used.
- `max_features`: the maximum number of features to consider when constructing each tree.

`fit` Method

In the `fit` method, the forest is trained. Each tree is built using the bootstrapping process, where samples are selected with replacement from the original dataset. Additionally, feature sampling is applied, where only a subset of features is randomly selected to train each individual tree. This process helps reduce the correlation between the trees in the forest. Each tree is an instance of the `DecisionTree` class, which is trained using a subset of data and features. Ultimately, the resulting forest is a list of trees, with each tree associated with the features selected during training.

`predict` Method

The `predict` method makes predictions on a new dataset. For each tree in the forest, a prediction is made, and the final prediction is determined by majority voting. In other words, each tree casts a vote, and the final class is the one that receives the most votes.

`accuracy` and `error` Methods

The accuracy method calculates the accuracy of the model by comparing the predictions to the true labels, while the error method returns the classification error, which is complementary to accuracy.

Cross-Validation

The `cross_validation_rf` function is an implementation of k-fold cross-validation for optimizing the model parameters and is similar to the one used for the decision tree. Specifically:

- `k`: the number of folds for cross-validation.
- `param_grid`: a grid of parameters containing all possible combinations of values for the model's parameters, which are the same parameters used to initialize the class.

For each combination of parameters in the grid, the function performs cross-validation on k-folds (in this case, 5), training the Random Forest model on each fold and calculating the performance (accuracy and error) on both the training and test sets. The results for each parameter combination are collected in a dictionary, which is then converted into a Pandas DataFrame. This

DataFrame provides an overview of the average performance for each hyperparameter combination, making it easier to select the best parameters.

Parallelization The `cross_validation_rf` function uses the joblib library to parallelize the execution of the computations, leveraging multiple CPU cores to speed up the evaluation of parameters. The n_jobs=-1 argument sets the number of parallel processes equal to the number of available cores, optimizing the execution time during grid search.

6 Results

The same approach that was used in the `DecisionTree` is applied now to the `RandomForest` algorithm: after an initial evaluation with random parameters, cross-validation is used to optimize the hyperparameters, and the model is retrained and tested with the best values. This ensures that the algorithm is as performant as possible, striking a good balance between predictive capability and generalization, and minimizing the risk of overfitting.

6.1 Initial Random Forest Model

In this initial phase, a first Random Forest model is created using default parameters:

- `Number of Trees: 5`
- `Max Depth: 10`
- `Min Samples Split: 2`
- `Impurity Type: gini`
- `Max Features: None`

The model is trained on the training data using the method `rf.fit(X_train, y_train)`. The performance of the model is then evaluated on both the training set and the test set, yielding the following results:

- `Initial Train Accuracy: 0.8877`
- `Initial Train Error: 0.1123`
- `Initial Test Accuracy: 0.8909`
- `Initial Test Error: 0.1091`

These values provide an initial evaluation of the model's performance, serving as a benchmark before the optimization phase. Both accuracies are relatively high, with the test accuracy being slightly higher than the training accuracy: this suggests that there is no clear indication of overfitting or underfitting at this point.

6.2 Random Forest Hyperparameter Optimization with Cross-Validation

To improve the model's performance and ensure better generalization, a hyperparameter grid (`param_grid_rf`) was defined with the following options:

- `Number of Trees: 5`
- `Max Depth: 10, 15`
- `Min Samples Split: 2, 10`
- `Impurity Type: "gini", "psi3", "psi4"`
- `Max Features: 3, 14`

Next, standard cross-validation (k=5) was performed using the previously described function, where the input features and the target variable from the training set were used, and the training set was divided into 5 folds. The results were sorted by test error (from the smaller to the bigger) to intuitively and quickly identify the best combination of hyperparameters. In Table 2 we can see the results from the different Random Forest models,

n_trees	max_depth	min_samples_split	impurity	max_features	mean_train_accuracy	mean_test_accuracy	mean_train_error	mean_test_error
5	15	2	gini	14	0.996612	0.996022	0.003388	0.003978
5	15	10	gini	14	0.996115	0.995179	0.003885	0.004821
5	15	2	psi3	14	0.992348	0.991032	0.007652	0.008968
5	15	10	psi3	14	0.992095	0.990964	0.007905	0.009036
5	10	10	gini	14	0.955983	0.955327	0.044017	0.044673
5	10	2	gini	14	0.955099	0.954956	0.044901	0.045044
5	10	2	psi3	14	0.935126	0.934828	0.064874	0.065172
5	15	2	psi4	14	0.923934	0.922151	0.076066	0.077849
5	10	10	psi3	14	0.921439	0.920802	0.078561	0.079198
5	15	10	psi4	14	0.911419	0.909980	0.088581	0.090020
5	15	2	gini	3	0.915506	0.884862	0.084494	0.115138
5	10	2	gini	3	0.868717	0.862610	0.131283	0.137390
5	10	10	gini	3	0.855874	0.850506	0.144126	0.149494
5	15	2	psi4	3	0.878249	0.850202	0.121751	0.149798
5	15	10	psi4	3	0.855351	0.833749	0.144649	0.166251
5	15	2	psi3	3	0.865279	0.831153	0.134721	0.168847
5	15	10	psi3	3	0.838800	0.825354	0.161200	0.174646
5	10	2	psi4	14	0.822830	0.822252	0.177170	0.177748
5	10	2	psi3	3	0.829083	0.821578	0.170917	0.178422
5	15	10	gini	3	0.850110	0.820432	0.149890	0.179568
5	10	10	psi3	3	0.825527	0.816554	0.174473	0.183446
5	10	10	psi4	14	0.812321	0.811598	0.187679	0.188402
5	10	10	psi4	3	0.808503	0.808800	0.191497	0.191200
5	10	2	psi4	3	0.795862	0.790762	0.204138	0.209238

Table 2: Cross-Validation Results (Sorted by Test Error)

and it is immediately evident that the best-performing model has the following parameters:

- `Number of Trees: 5`
- `Max Depth: 15`
- `Min Samples Split: 2`
- `Impurity Type: gini`
- `Max Features: 14`

6.3 Final Model Results (after Hyperparameter Tuning)

Using these optimized hyperparameters, a final model was trained on the entire training dataset and tested on the main test set. The final performance results are:

- `Final Train Accuracy: 0.9971`
- `Final Train Error: 0.0029`
- `Final Test Accuracy: 0.9958`
- `Final Test Error: 0.0042`

It is immediately apparent that the results have significantly improved compared to the initial model, particularly the test error, which has been substantially reduced. This demonstrates an enhanced ability of the Random Forest model to generalize, and confirms that there is no overfitting or underfitting present after optimization.