Tree predictors for binary classification

Mattia Pantò
Dipartimento di Informatica, Università degli studi di Milano
Email: mattia.panto@studenti.unimi.it

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/our 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

This report documents the work done to implement a binary decision tree. The project was developed in Python and includes the main classes and functions for training, tuning hyperparameters, and testing the model. A greedy algorithm was implemented, which starts from the root node and develops the tree in depth. The model uses a single feature for each node and was trained to classify records in a mushroom dataset as poisonous or edible. Finally, the model's performance was analyzed and compared with that of a random forest ensemble method.

For simplicity, many names of classes, methods and attributes used in this work correspond to those used in the scikit-learn[4] library.

2 Project Structure

The main classes for creating and managing tree predictors are contained in the 'tree.py' module. This module contains:

- the 'Node' class for managing the tree nodes.
- the 'TreePredictor' class for creating the decision tree and managing the training and prediction phases.
- 'RandomForest' class for creating and managing random forest(RF) using the implemented predictor trees.

In the 'ML_tools.py' module there are the implemented classes and methods useful for the development of this project. It contains:

- Functions for Compunting the metrics used for performance evaluation.
- Functions for the hyperparameter tuning phase such as grid search and cross-validation.
- Classes and methods for the preprocessing phase.

3 Decision Trees

To create an instance of 'TreePredictor' it is necessary to specify a stopping criterion and a splitting criterion. The stopping criterion is defined by the parameters 'max_depth' and 'min_samples_leaf'. The termination of the growth of the tree is managed in this way:

- Tree expansion ends if a leaf node is pure, meaning all data points that reach that leaf have the same label.
- The 'max_depth' hyperparameter defines how deep a decision tree can go. When a node reaches this depth, its expansion stops.
- The 'min_samples_leaf' hyperparameter defines the minimum number of training samples that must be routed to a leaf. If splitting a node results in any leaf having fewer samples than this threshold, the split is not performed, and the node becomes a leaf.

The splitting criterion is specified as a string parameter. The following impurity measures have been implemented:

- gini: $\psi_1(p) = 2p(1-p)$
- scaled_entropy: $\psi_2(p) = -\frac{p}{2}\log_2(p) - \frac{1-p}{2}\log_2(1-p)$
- sqrt_impurity: $\psi_3(p) = \sqrt{p(1-p)}$

In addition, the parameters 'num_candidate_attributes' and 'random_state' can be specified to reduce the number of candidate attributes for the split by extracting a random subset for each node. These parameters were used only for creating random forests.

3.1 Thresholds for Continuous Features and Categorical Features

Continuous and categorical attributes are treated differently. Before the decision tree is created, the type of features in the training set is analyzed. A function generates a dictionary with one element for each feature. Each element is associated with a pair of values:

1. A list that can contain:
   - For categorical attributes all categories related to the feature.
   - For continuous features a certain number of numeric thresholds. To compute them, the attribute values are sorted in ascending order, then the thresholds are computed as the average between each pair of adjacent values. If the 'num_thresholds' parameter is specified, it is possible to reduce the total number of thresholds returned, selecting only those that are at multiple positions of a step calculated as the ratio between the total number of instances and num_thresholds.
2. A Boolean value that can be 'True' if the attribute is categorical, 'False' otherwise.

3.2 Construction of the Decision Tree

The decision tree is initialized with its root node. During training, the tree expands, creating two leaves for each node. For each internal node, a tuple is assigned that contains the information to build a decision function. This function is different depending on the type of attribute that needs to be evaluated (continuous or categorical). Tree expansion is a recursive process.

The construction of the tree predictor is started by calling the 'train()' method which requests the training dataset as input. A recursive process is started by calling function '_grow_tree()'. This function first evaluates the stopping criteria and, if at least one of these conditions is satisfied, the recursion terminates by returning a leaf node with the 'is_leaf' parameter set to 'True'. When a leaf node is returned, the associated label is computed through a majority voting among the examples routed to that leaf. Finally, a counter that tracks the sum of errors made by the leaves of the tree is updated by adding the number of misclassified samples. In this way, at the end of construction, it is possible to calculate the training error by dividing the sum of the errors by the total number of instances.

If the termination criterion is not satisfied, the best split function is computed and the '_grow_tree()' function is called for the left branch and the right branch. This function, in addition to the dataset routed to the current node, also requires a parameter that keeps track of the depth of each node.

If the tree has not already been terminated, an internal node with an associated decision function is returned.

To find the decision function that maximizes the gain, the 'find_best_split()' function was implemented. This calculates the gain obtained from a split of the leaf ℓ into leaves ℓ' and ℓ'', using an impurity function ψ:

$$\psi(p) - (\alpha\psi(q) + (1-\alpha)\psi(r))$$

where $p = \frac{N_\ell^+}{N_\ell}$, $q = \frac{N_{\ell'}^+}{N_{\ell'}}$, $r = \frac{N_{\ell''}^+}{N_{\ell''}}$ e $\alpha = \frac{N_{\ell'}}{N_\ell}$.

For each leaf, the gain is computed by testing all the decision function that act on each attribute and on each threshold/category defined in the dictionary structure described above. In the case of categorical attributes, the decision function evaluates whether the attribute of a datapoint belongs exactly to the category; for continuous attributes it is verified that the value of the attribute is less than a threshold.

If the 'feature_subset_size' parameter is specified and is an integer less than or equal to the number of attributes, the evaluation is done only on a subset of randomly extracted features. This operation is managed by a list of blocked features that is extracted for each node. Finally, the following values are returned:

- A tuple of three values containing the information relating to the best decision function. The first value is a string indicating the feature to be evaluated, the second is the category or threshold and can be a char (if the feature is categorical) or a float (if the feature is continuous), and the third is a boolean indicating whether the feature is categorical.

- The indices of the elements routed in the left branch and those routed in the right branch.

3.3 Predictions with Decision Tree

If the decision tree has been created, the 'predict_datapoint()' function predicts the label of a datapoint starting at the root and moving through the nodes of the tree. For each node it is evaluated whether it is a leaf node; if it is, the function returns the label associated with the leaf.

If the current node is not a leaf, the node's 'make_decision()' function is used to decide whether to move to the left or right branch. If the decision function returns 'True' it is routed to the left branch, otherwise to the right branch and repeating the cycle until a leaf node is reached.

To obtain predictions on a set of datapoints, the 'predict()' function is used. This function computes a prediction for each datapoint and returns a list containing the predictions.

3.4 Efficency

The time needed to build the decision tree depends on:

1. The maximum depth and stopping criteria used.
2. The number of thresholds considered for continuous attributes and the number of categories for categorical attributes.
3. The size of the training set.
4. The distribution of the training set data.

In particular, to find the best decision function, the implemented algorithm computes the gain obtained by evaluating all the splits involving all the attributes and all the respective thresholds. Each datapoint must also be routed to the correct branch. This operation is done with a complexity of $\mathcal{O}(d \cdot n \cdot m)$, where d is the number of attributes, n the number of data points and m the maximum number of thresholds or categories for an attribute. Furthermore, the number of nodes of each level increases exponentially with respect to the depth of the level, and therefore with a maximum depth i the construction algorithm has a complexity of $\mathcal{O}(2^i \cdot d \cdot m \cdot n)$.

The complexity for the prediction, however, increases linearly with respect to the depth of the tree, which determines the maximum number of nodes visited during a prediction. For this reason the complexity is $\mathcal{O}(i)$.

4 Random Forest

Using the tree predictor structure described in the previous section, the 'RandomForest' class was created to implement an ensemble method. In this project, decision trees were trained on bootstrap samples generated by sampling with replacement from the training set. The size of each generated bootstrap sample is the same as the original training set. To reduce the risk of overfitting, the possibility of limiting the number of candidate attributes for the split has been introduced. If 'num_candidate_attributes' is specified when creating tree predictors, a subset of attributes of that dimension is extracted for evaluation at each node.

To create the random forest (RF), in addition to the 'num_candidate_attributes' parameter, the following must be specified:

- The parameters to be used for the creation of decision trees (in this case each tree is expanded without using pruning techniques).
- The boolean parameter 'compute_oob_score'. If this is True, at the end of the creation of the random forest, the out-of-bag error is computed. This metric was used when tuning the 'num_candidate_attributes' hyperparameter.
- The 'random_state' parameter is used to make stochastic operations reproducible.

To speed up the training process, the creation of the trees was parallelized using the ProcessPoolExecutor class from concurrent.futures module. The random forest is created by calling the 'train()' method, for which it is necessary to specify the training set, the maximum number of thresholds for the evaluation of numerical attributes, the number of trees and the number of concurrent processes used.

The 'random_state' passed during the instantiation of the 'RandomForest' class is used to generate a list of random states (with a size equal to the number of trees in the RF) necessary to guarantee reproducibility in the bootstrap sample generation and candidate attribute extraction phases. During the creation of the RF, the generated trees are stored in an internal list, and for each tree a list of the indices of the data points belonging to the out-of-bag set is computed and stored.

When a new sample needs to be classified, a prediction is computed from all the decision trees that make up the random forest, and through a majority voting system, the predicted label is returned. To facilitate the computation of the oob score, tree indices that can vote for the final prediction can be selected.

5 Experimental Setup

5.1 Dataset

The dataset used to build the decision tree is the 'Secondary Mushroom dataset'[6], which contains 61068 samples described by 20 attributes, divided into continuous and categorical. The dataset is composed of 353 simulated mushroom for each of the 173 species considered. Each mushroom is classified as edible (e) or poisonous (p), with a ratio of 0.45 (e) and 0.55 (p). All records have at least one unknown attribute, but only some attributes have missing values. The dataset contains some duplicate datapoints. However, these duplicates were kept in the dataset, since in this specific case they could represent different instances with the same attributes.

For the construction of the decision trees, the dataset was divided into training and test sets[1] with a ratio of 0.7 and 0.3; therefore using 42748 samples for the training set and 18321 for the test set.

5.2 Preprocessing

Removing instances with missing attributes is not a viable solution. Therefore, it was decided to handle these cases by assigning a mean for continuous attributes and a mode for categorical attributes; this operation is handled by the 'MissingValuesImputer()' class. The mean and mode are computed on the training set and stored in an internal dictionary

[1] In the project, a random state of 24 was used to shuffle the dataset before splitting it.

structure. Datapoints in the validation set, test set or new data assume the stored values in cases where the attribute value is missing.

Finally, the labels were mapped to boolean values, with 'e' as True and 'p' as False. No other preprocessing operations was necessary on the data. Tree predictor is not sensitive to feature scaling and, due to the way the algorithm has been implemented, no categorical attribute encoding operation is necessary.

5.3 First Evaluation of Fully Expanded Trees

Before hyperparameter tuning, fully expanded decision trees were created, arriving at pure leaf nodes without applying other termination criteria. The resulting tree using 'gini' as splitting criterion has a maximum depth of 26 levels with 211 leaf nodes. Using 'scaled_entropy' the expansion stops after 22 levels with a total of 180 leaf nodes, while with 'sqrt_impurity' the tree reaches depth 30 with 202 leaf nodes. Trees of this type all have an accuracy on the training set of 100% and on the test set of 99.853% with 'gini', 99.864% with 'scaled_entropy' and 99.836% with 'sqrt_impurity'. The high accuracy on the test set indicates that even a fully expanded tree is not particularly prone to overfitting. However, the model could be further improved by considering only the most informative attributes and optimizing the choice of hyperparameters.

5.4 Feature Selection

The samples in the dataset are characterized by 20 attributes, and it is assumed that not all of them have the same discriminative power. To evaluate the importance of each attribute in the classification, the mutual information between each attribute and the target variable was computed. The mutual information of two random variables is a measure of the mutual dependence between the two variables [1]. For discrete random variables, it is calculated as follows:

$$I(X;Y) = H(X) + H(Y) - H(X,Y) = \sum_{x,y} p(x,y) \log \frac{p(x,y)}{p(x)p(y)}$$

To compute the mutual information, the 'mutual_info_classif()' function present in the scikit-learn library was used. In Fig.1 the results obtained.

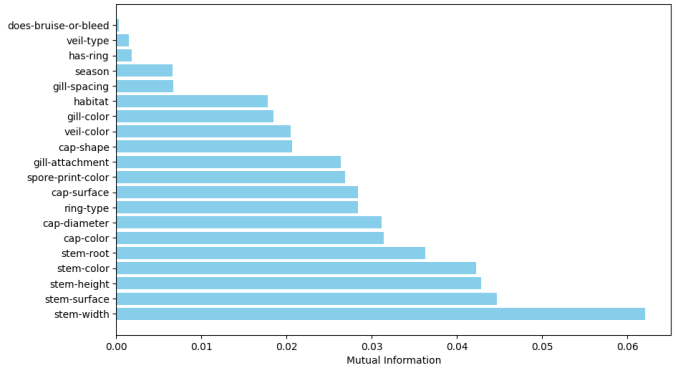


Figure 1: Comparison of mutual information values computed between dataset attributes and the target variable.

The 'stem-width' attribute appears to be the most informative, while the 'has-ring', 'veil-type' and 'does-bruise-or-bleed' attributes show a low correlation with mushroom edibility. The impact of removing these attributes was evaluated by doing a 4-fold cross-validation on the training set data considering a generic decision tree with 'max_dept'=26, 'min_samples_leaf'=1 and 'splitting criterion'="scaled_entropy". From the results obtained shown in Tab.1, it was decided not to remove any features.

Table 1

Removed features	CV_score
-	**99.854%**
'does-bruise-or-bleed'	99.798%
'veil-type','does-bruise-or-bleed'	99.798%
'has-ring','veil-type','does-bruise-or-bleed'	99.789%

5.5 Hyperparameter Tuning for Tree Predictor

Given the large number of instances in the dataset, nested cross-validation was not used for hyperparameter tuning. Instead, a cross-validation (CV) was performed on the training set with 4 folds. At each CV cycle, the dictionary used to replace missing values is recomputed on the training folds and applied to the samples of the validation fold. In this way the samples in the validation fold do not depend on those in the training folds.

To select the hyperparameters that maximize accuracy, a grid search was performed (The computational time for the grid search was reduced by limiting the total number of thresholds to be analyzed for continuous attributes to 50).

Grid search is handled by a function that requires as parameters: the class of the predictior to be analyzed, the training data, and a grid of parameters. The grid of parameters is a dictionary that associates each hyperparameter with a list of values to be tested. The 'GridSearchCV()' function makes a CV to estimate the risk of the predictors obtained from each combination of parameters. At the end, the function returns a list of combinations sorted by validation accuracy (other metrics such as precision, recall, and f1 score are also returned).

The range of hyperparameter values was chosen based on a general observation of the predictors' performance and considering the maximum depth obtained using fully expanded trees. For instance, maximum depths lower than 20 were not considered, as they would have led to underfitting. Specifically, the following were considered:

- "max_depth" : [20, 22, 24, 26, 28, 30]
- "min_samples_leaf" : [1, 2, 5, 10]
- "splitting_criterion" : ["gini", "scaled_entropy", "sqrt_impurity"]

The hyperparameters that obtained a higher validation accuracy were found to be:

- "max_depth" : 28
- "min_samples_size" : 1
- "splitting_criterion" : "scaled_entropy"

Using these hyperparameters, a validation accuracy of 99.878% was obtained; the same accuracy is obtained with a maximum depth of 30. The same hyperparameter configuration also optimizes the precision, recall and f1_score metrics. So the predictor with optimized hyperparameters is the same one obtained by letting the tree grow until pure leaf nodes are reached. In this case, the performance obtained with 'scaled_entropy' is often better than the other two impurity measures.

5.6 Hyperparameter Tuning for Random Forest

For the creation of the RF it was decided to use fully expanded trees without using pruning techniques. The splitting criterion used is 'scaled_entropy'. In the implemented RF, the number of attributes to be evaluated for the split of each node can be reduced by specifying the 'num_candidate_attributes' parameter; a standard number of candidate attributes for classification problems is $\sqrt{d}$ with d attributes. Therefore it was decided to consider 5 attributes for computing the decision function for each node.

In general, increasing the number of trees in an RF does not reduce the accuracy of the model. For this reason, the choice of the size of the RF was made by evaluating the trade-off between performance obtained and the computational times. To choose the number of trees, the out-of-bag(OOB) accuracy obtained with RFs of different sizes were compared[2]. As shown in Fig.2, by increasing the number of trees the OOB accuracy grows until it reaches the maximum value. Using RF with dimensions greater than 25 results in OOB accuracy of 100% with all seeds tested. The RF considered is made up of 30 trees as it was considered of sufficient size to obtain good results while maintaining acceptable computational times.

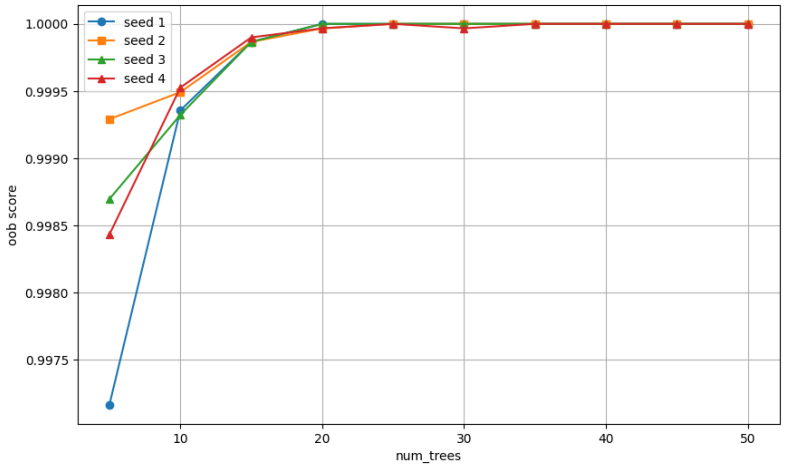


Figure 2: OOB accuracy values as the number of trees increases by comparing RFs obtained using 4 seeds.

6 Results

6.1 Visualization of Decision Trees

Fig.3 shows the first levels of the tree with optimized hyperparameters and trained on the training set. The third node at the second level immediately becomes a leaf node, since all data points routed to that node are labeled False. The stem-width, gill-spacing and stem-surface features are therefore those that best characterize the mushroom. Many of the attributes with a high value of mutual information, are evaluated by the decision function in the first nodes of the tree. In Fig.4 a visualization of the decision boundary of the first layer with a subset of examples from the training set. The optimized decision tree have an average of 202 training samples per leaf.

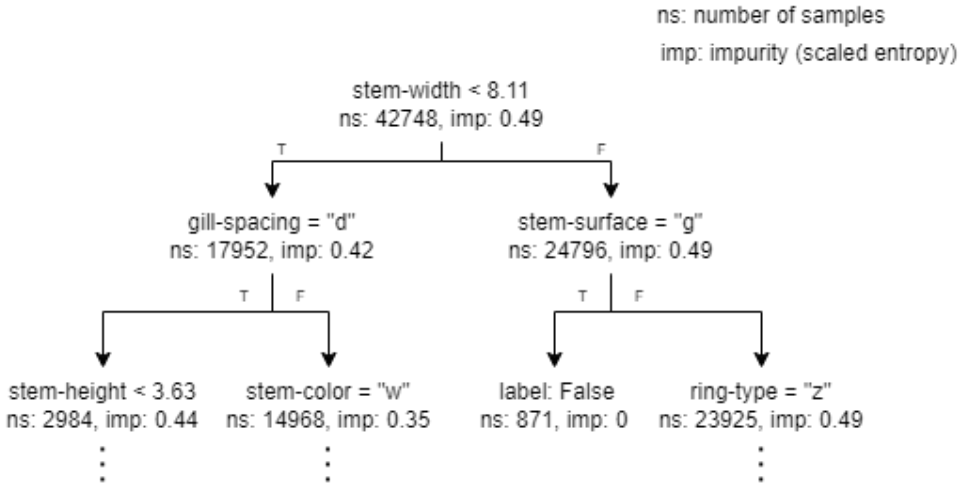


Figure 3: Root node, first and second layers of a decision tree trained with optimized hyperparameters on the training set.

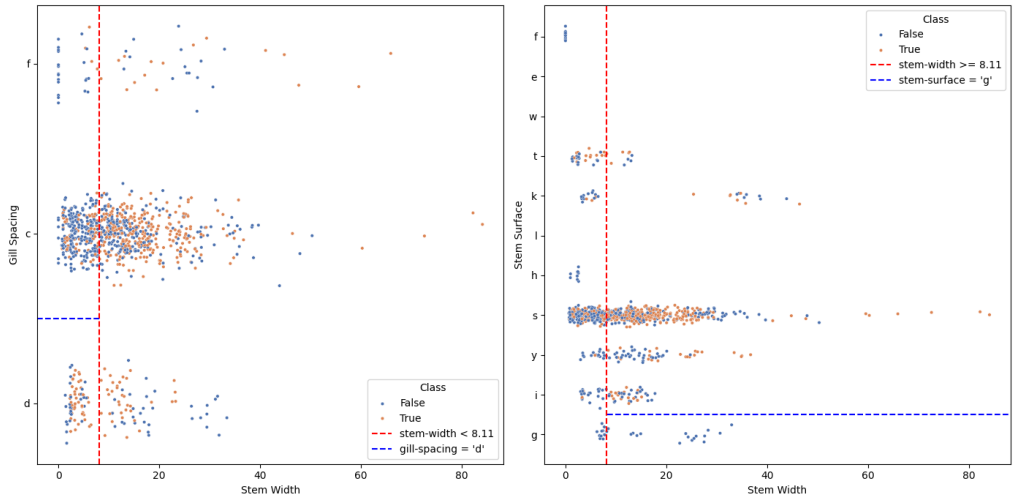


Figure 4: Decision boundaries of the first layer of the decision tree.

6.2 Evaluation of Decision Trees

Several trees were built to evaluate the presence of overfitting or underfitting and to be able to compare the performances with the results of the trained tree using the optimized hyperparameters. Each tree was built using the samples from the training set and the accuracy, precision, recall and f1 score metrics were calculated on the test set data. Tab.2 reports the performance of various decision trees, including one with optimized hyperparameters. To evaluate splits for numerical attributes, 50 thresholds were considered.

Table 2

Config	Train acc.	Test acc.	Test precision	Test recall	Test F1 score
28,1, scaled_entropy	**100%**	**99.864%**	**99.79%**	**99.901%**	**99.845%**
28,1,gini	100%	99.853%	99.789%	99.876%	99.833%
20,1,scaled_entropy	99.925%	99.831%	99.777%	99.839%	99.808%
28,1,sqrt_impurity	99.998%	99.831%	99.826%	99.789%	99.808%
28,3,scaled_entropy	99.934%	99.82%	99.703%	99.888%	99.796%
28,20,scaled_entropy	99.085%	98.903%	98.7%	98.81%	98.755%
10,1,scaled_entropy	85.288%	85.236%	83.142%	83.379%	83.26%

The results confirm that a very specialized tree on the training set samples is able to generalize to the test set samples as well. In addition, the choice of hyperparameters strongly influences the accuracy on the test set, since trees with a low maximum depth or a high number of minimal examples per leaf are prone to underfitting. All evaluated decision trees perform very well, particularly when scaled entropy is used as a splitting criterion. In most of the tests performed, recall is higher than precision, this is an indication that the tree predictor can classify edible mushrooms more easily.

6.3 Evaluation of Random Forest

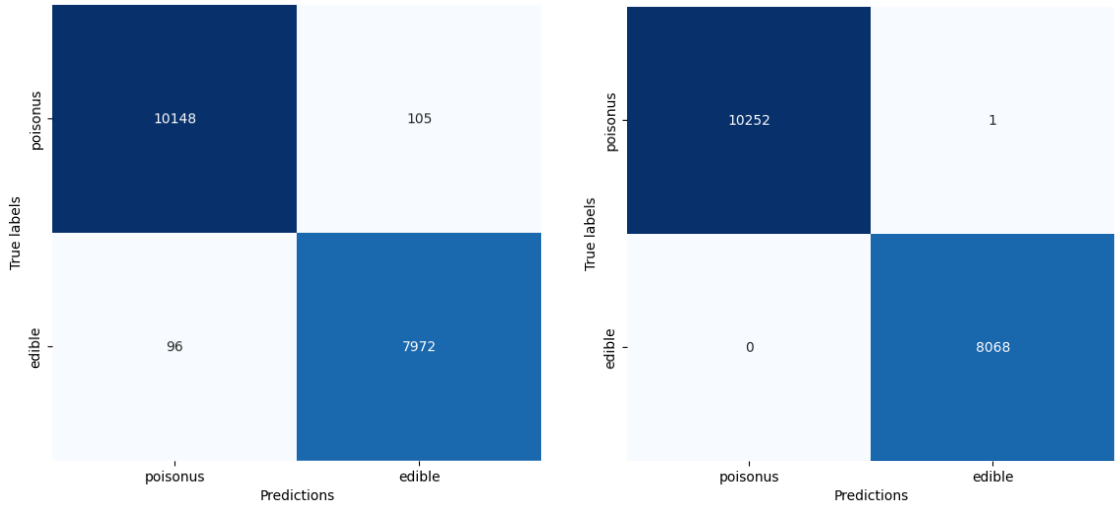
To evaluate the performance of the ensemble method, RFs with 30 trees and 5 candidate attributes extracted for each node were considered. The decision trees that make up the implemented RFs have no maximum depth and the minimum number of samples per leaf is 1; the splitting criterion used is scaled entropy. The performance obtained[2] is as follows:

Table 3

Numero di alberi	Train acc.	Test acc.	Test precision	Test recall	Test F1 score
30	**99.707%**	**99.995%**	**99.988%**	**100%**	**99.994%**
40	99.678%	99.995%	99.988%	100%	99.994%
15	99.744%	99.989%	99.975%	100.0%	99.988%
5	99.709%	99.967%	99.975%	99.95%	99.963%

The accuracy on the training set is intended as the average training accuracy of the trees that make up the RF. Does not reach 100% due to the small number of attributes evaluated for the split. In fact, if none of the 5 extracted attributes is able to separate the

[2]In the project, a random state of 1 was used for generating the random forest.



(a) CM for tree predictor.

(b) CM for random forest.

Figure 5: Comparison of the confusion matrices between the optimized decision tree and a random forst with 30 predictors.

samples routed on that node, the subtree expansion ends. The average accuracy on the test set of the individual trees that make up the RF is 99.58%, lower than that obtained by the RF. This confirms that the use of ensemble techniques can lead to improvements in performance. The use of an RF allows to obtain better results than the single decision tree on all the metrics evaluated.

7 Conclusion

The tree predictors used to classify a mushroom dataset proved to be effective and highly accurate. The constructed decision trees do not suffer from overfitting despite their high depth, so the pruning techniques applied were not essential to limit overfitting. Although the performance achieved with a decision tree was already very good, the accuracy of the model could be further improved by using an ensemble method such as random forest. In this context, the high accuracy on the training set is justified by the simplicity of the problem and the presence of well-defined patterns in mushroom classification.

The tree predictors used for the classification of the mushroom dataset have proven to be effective, obtaining high accuracy values. Despite the depth of the constructed decision trees, they are not subject to overfitting, given that the high accuracy obtained can be justified by the simplicity of the mushroom classification problem, which presents well-defined patterns. Despite the already excellent performance obtained from a single tree, it was possible to further increase the accuracy of the model using an ensemble method such as random forest.

References

[1] Wikipedia contributors. Mutual Information. Wikipedia, The Free Encyclopedia, 2024. https://en.wikipedia.org/wiki/Mutual_information.

[2] Wikipedia contributors. Out-of-bag error. Wikipedia, The Free Encyclopedia, 2024. https://https://en.wikipedia.org/wiki/Out-of-bag_error.

[3] Google Developers. Decision forests, n.d. Accessed: 2024-08-26.

[4] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

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

[6] Dennis Wagner, D. Heider, and Georges Hattab. Secondary Mushroom. UCI Machine Learning Repository, 2023. DOI: https://doi.org/10.24432/C5FP5Q.