Statistical Methods for Machine Learning: Tree Predictors for Binary Classification

Matteo Onger

October 2024

1 Introduction

The goal of the project is to implement binary decision trees from scratch and use them also to implement random forests. These models will then be trained to obtain predictors that will hopefully be able to classify mushrooms as poisonous or edible, so to solve a binary classification problem.

In this document, the symbol Y will be used to denote the set of all the possible labels, which, in the specific case of the problem addressed, is $Y = \{e, p\}$. While S will be used to denote the dataset, $m = |S|$ will be its cardinality and d will be the number of features.

The report is structured as follows: the following section briefly describes the dataset used, while the third one lists the most relevant implemented classes with their characteristics and analyzes some implementation choices. Finally, before reaching the conclusion, the fourth section reports the experimental results of the tests performed in order to verify the performance of the models created.

2 Dataset Description

The *Secondary Mushroom* dataset [2] includes 61069 hypothetical mushrooms based on 173 species (353 mushrooms per species) and each of them is labeled as edible (e) or poisonous (p). Under this point of view, the dataset is not perfectly balanced, as 55.49% of the mushrooms are classified as poisonous, while the remaining ones (44.51%) are classified as edible.

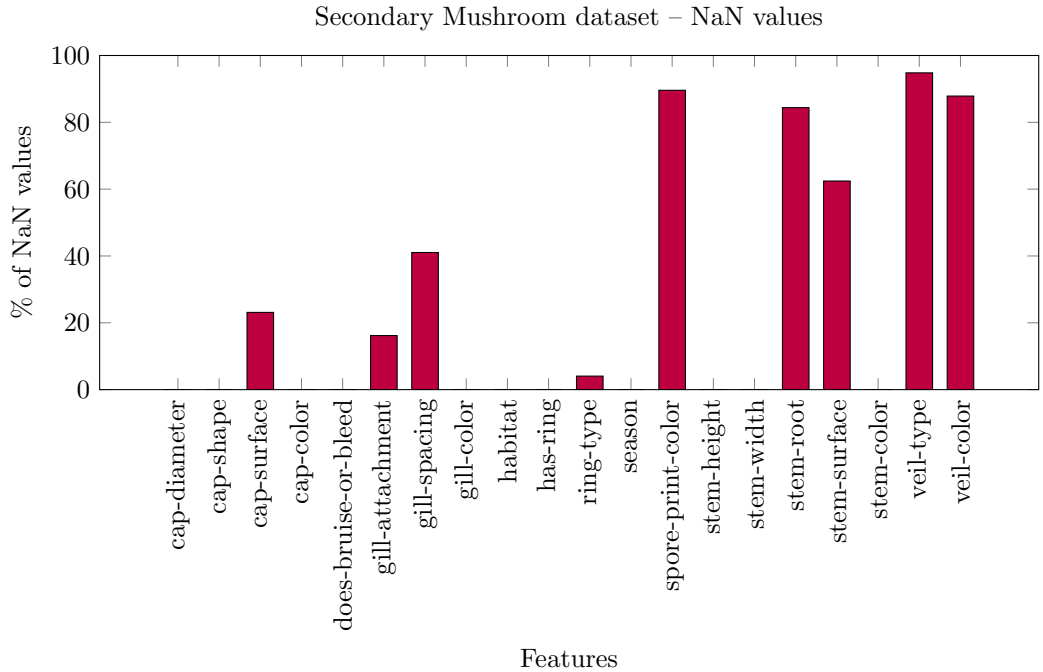

Figure 1: Percentages of NaN values per feature.

The features considered are 20: 17 of them are categorical features, while the remaining three (*cap-diameter*, *stem-height*, *stem-width*) are continuous features. As clearly shown by the plot 1, some features have a large number of NaN values, such as *veil-type*, which takes this value in over 94% of cases. So either these values will need to be removed during the preprocessing, or the predictor will need to know how to handle them. In any case, classification risks being more complex with many missing values, especially if their presence is not justified and it is therefore the result of a mistake.

Focusing on categorical features, and leaving aside the continuous ones, for which the following reasoning does not make much sense, they have, on average, 7.3 possible values, with some features (like *stem-color*) having up to 13 unique possible values and others (like *veil-type*) being effectively binary features.
As for the three continuous features, instead, the table 1 below shows the minimum value, the maximum value, the mean and the standard deviation of each of them.

	cap-diameter	*stem-height*	*stem-width*
Min	0.38	0.00	0.00
Mean	6.73	6.58	12.14
Std	5.26	3.37	10.03
Max	62.34	33.92	103.91

Table 1: Brief description of the continuous features in the dataset.

3 Implementation details

In the following section, the main Python modules developed are described and some implementation choices adopted are discussed.

3.1 Module: *data.py*

The main class in this module is the class `DataSet`: it basically wraps a Pandas `DataFrame` and provides some specific methods to simplify its use as a training set or as a test set. Not by chance, the only input required by the methods `fit` and `predict` of the implemented predictors is in fact an object of this type. The methods implemented in the class `DataSet` allow to get, if known, the expected label of each data point in the dataset, for example, or to retrieve all the values of a feature given its name.
Each data point can be seen as a row, while each feature as a column of the dataset; so its shape will be $m \times d$, where m is the number of data points and d is the number of features. If the labels are known, they can be added within the same dataset, with the corresponding features, in a special column, whose name will have to be specified during the initialization of the object. In this case, its shape will be $m \times (d + 1)$.

	class	*cap-diamete*	...	*habitat*	*season*
0	p	15.26	...	d	w
1	p	16.60	...	d	u
...	...	...	...	...	...
61067	p	1.24	...	d	u
61068	p	1.17	...	d	u

Table 2: Tabular representation of the *Secondary Mushroom* dataset as an example.

The inner class `Schema`, on the other hand, allows to obtain more information about the "structure" of the dataset: it is possible, for instance, to get the type of a feature (categorical or numerical), or its domain, here understood as the list of all values that appear in the dataset for a certain feature but without duplicates, hence as a list of unique values. All this information is crucial during predictor's training phase, particularly when looking for the best split.

3.2 Module: *bintreepredictor.py*

The implementation of binary decision trees is deeply based on the class `BinNode`. In fact, every inner node and every leaf of a tree is represented by an object of this type.

A dataset can be assigned to a `BinNode` object with no children (i.e. a leaf node). But these data points will be used during the training phase to build the decision tree, so this set will be used as a training set and, therefore, it must contain the expected labels.
In addition, a node can be assigned a test. A test, that is an instance of the inner data class `TestCondition`, has two fields: the name of the feature under consideration and a threshold. Using the method `check_test`, it is possible to verify the impact of the test that has been set on a dataset: a list of boolean values will be returned and, if the feature under consideration is categorical, the i-th entry will be `True` if and only if the value of the i-th data point, always for the feature under analysis, is equal to the threshold that has been set. Whereas, if the feature is numerical, the i-th element will be `True` if and only if the feature value of the i-th data point is less than or equal to the threshold.
The `split_node` method, on the other hand, allows to actually split a node based on the test assigned to it: two child nodes will be created and the data previously assigned to the parent will be split between them. The data points for which the test is `True` will be assigned to the left child, while the remaining ones will be assigned to the right child.

Regarding missing values, several strategies can be adopted to manage them, one possible solution are the surrogate splits: they act as backup choices when the primary test cannot be applied, that is, when the data point to be processed has value NaN for the feature considered by the primary test.
The solution implemented here is more straightforward and simply imposes to always route data points having value NaN for the feature used by the test to the right child. Despite its simplicity, this approach can still lead to good results, especially when even NaN values provide useful information: for example, the feature *veil-type* often takes value NaN simply because many mushrooms do not have a veil, so they actually have something in common and routing them towards the same child makes absolute sense. In all such cases, missing values should not be considered a problem of the dataset. Obviously, it would be completely different if there were many missing values but without a valid justification, especially if they were not missing at random. In that case, it is necessary to act on the problem during the preprocessing phase.

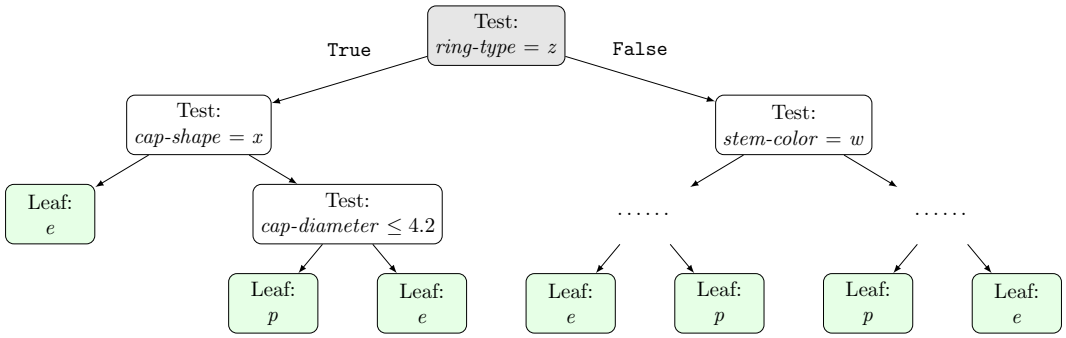

Figure 2: Example of binary decision tree.

The other class defined in this module is the `BinTreePredictor` class which, as the name indicates, implements the actual decision tree using the classes seen so far. During the initialization, all the hyper-parameters of the model, plus some additional function parameters, must be specified. The most important are:

- loss function: the loss function specified is used to compute the training error and the test error. Currently, the only implemented loss function is the zero-one loss.

$$\ell_{0-1}(y, \hat{y}) = \begin{cases} 1 & \text{if } y \neq \hat{y} \\ 0 & \text{otherwise} \end{cases} \qquad \text{with } y, \hat{y} \in Y \tag{1}$$

- prediction criterion: it is the criterion used by leaf nodes to determine which is their label based on the data assigned to them. Currently, the only implemented criterion is the mode. Hence, the label assigned to a leaf is the most common label among the data points assigned to the considered node.

- split criterion: it is the criterion used to determine which split (uniquely defined by a test and a leaf) is the best. Currently, three criteria are implemented: the entropy (equation 2),

the Gini impurity index (equation 3) and the misclassification rate, which is the percentage of misclassified data points. All these indices are normalized, so that their value is always between $[0, 1]$.

$$H(leaf) = \frac{-\sum_{y \in Y} \mathbb{P}(y) \cdot \log_2 \mathbb{P}(y)}{\log_2 |Y|} \tag{2}$$

$$G(leaf) = \frac{|Y| \cdot (1 - \sum_{y \in Y} \mathbb{P}(y)^2)}{|Y| - 1} \tag{3}$$

- stop criterion: it is used to limit the growth of the tree, so as to avoid overfitting. Currently, to do this, it is possible to limit the number of leaves or the height of the tree.

During the training phase, to find the best split, for each leaf, every possible test should be tried. This may not be feasible due to the excessively large number of combinations, so it is possible to specify the following hyper-parameters in order to try only a subset of all the possible tests for each leaf:

- maximum number of features: in each leaf, at most k randomly chosen features among the d available will be taken into account. Tests may be based only on them.
- maximum number of thresholds: at most t thresholds will be taken into account for each feature and leaf. This constraint is applied only to numerical features, since the domain of categorical features usually has a low cardinality.

All the implemented criteria work well if the decision tree is used in a classification problem, but they are not suitable for regression problems. In any case, the criteria are implemented as dictionaries having a structure like {`name:function, ...`}, so by simply extending the latter, it would be easily possible to also solve regression tasks or even to define new criteria for classification tasks.

Algorithm 1 `Fit` method of the class `BinTreePredictor`

Require: *data* ▷ *training set of size m*
  root ← *BinNode*(...)
  root ← assign *data*
  leaves ← {*root*} ▷ *init the tree*
  while *stop* is *False* **do**
    best_score ← 0
    best_leaf ← *None*
    best_feat ← *None*
    best_threshold ← *None* ▷ *best split found and its score*
    for all *leaf* ∈ *leaves* **do**
      for all *feat* ∈ *features* **do**
        for all *threshold* ∈ *thresholds* **do**
          score ← compute the score by using the split criterion
          if *score* > *best_score* **then**
            best_score ← *score*
            best_leaf ← *leaf*
            best_feat ← *feat*
            best_threshold ← *threshold*
    if *best_score* > 0 **then**
      best_leaf ← split this leaf using the test found
      leaves ← *leaves* \ {*best_leaf*}
      leaves ← *leaves* ∪ {*best_leaf.sx*, *best_leaf.dx*} ▷ *add the children of the split leaf*
    else
      break
  training_error ← $\frac{1}{m}\sum_{i=1}^{m} \ell(y_i, \hat{y}_i)$ ▷ *compute training error using the selected loss func*
  output *training_error*

As previously mentioned and as shown by the pseudo-code 1, a score is computed for each split: it estimates its quality and, at each outer iteration, only the split with the highest score is actually applied. The score is derived from the split criterion f in the following way:

$$score_{s,l} = \frac{|l|}{m} \cdot IG(s, l) = \frac{|l|}{m} \cdot \left(f(l) - \left(\frac{|l'|}{|l|} \cdot f(l') + \frac{|l''|}{|l|} \cdot f(l'') \right) \right) \tag{4}$$

where m is the cardinality of the training set, s is the split, l is the leaf to split, l' and l'' are the children of l after the split s and $|l|$ is the number of data points assigned to the leaf l. So, in other words, the goal is to maximize the information gain IG scaled by the percentage of data points involved in the split. This is done as an additional measure to control the overfitting: in fact, given the same IG, it is preferable to divide, at least first, the node with the greatest number of data points assigned, because the greater the number of samples involved, the greater the probability that the rule can be generalized to other datasets.

To get an idea of the predictor's performance, as shown in the pseudo-code above, the training error, computed by using the selected loss function, is returned by the `fit` method at the end of the training phase, while the test/validation error is returned by the `predict` method only if the dataset provided as input also contains the expected labels, only in this way, in fact, it can be used as a test/validation set.

3.3 Module: *binrandomforest.py*

The class `BinRandomForest` contained in this module, as the name suggests, implements the random forests. A random forest is an ensemble learning method that works by creating a collection of T decision trees during the training phase. Obviously, here, each tree is a binary decision tree represented by an object of the class `BinTreePredictor`.
The methods made available by this class are the same as those of the `BinTreePredictor` class, as are essentially the same the function parameters/hyper-parameters to be provided as input to the class constructor, with only two differences:

- a new hyper-parameter (`num_trees`) must be specified to indicate the number of trees that will make up the forest;
- and now it is possible to set the hyper-parameter `max_features` to *sqrt*. By doing this, during the training phase, once the number d of features available in the training set has been established, `max_features` will be set to $\sqrt{d}$. This is in fact a commonly chosen value.

Moreover, the criterion used to compute the final prediction, given the T labels predicted by the decision trees, is the same used to determine which label should be assigned to the leaves of the trees that make up the forest. So, currently, only the mode is implemented, but in regression tasks, for example, the mean could be used by simply adding the new rule to the dictionary `BinTreePredictor.PREDICTION_CRITERION`.

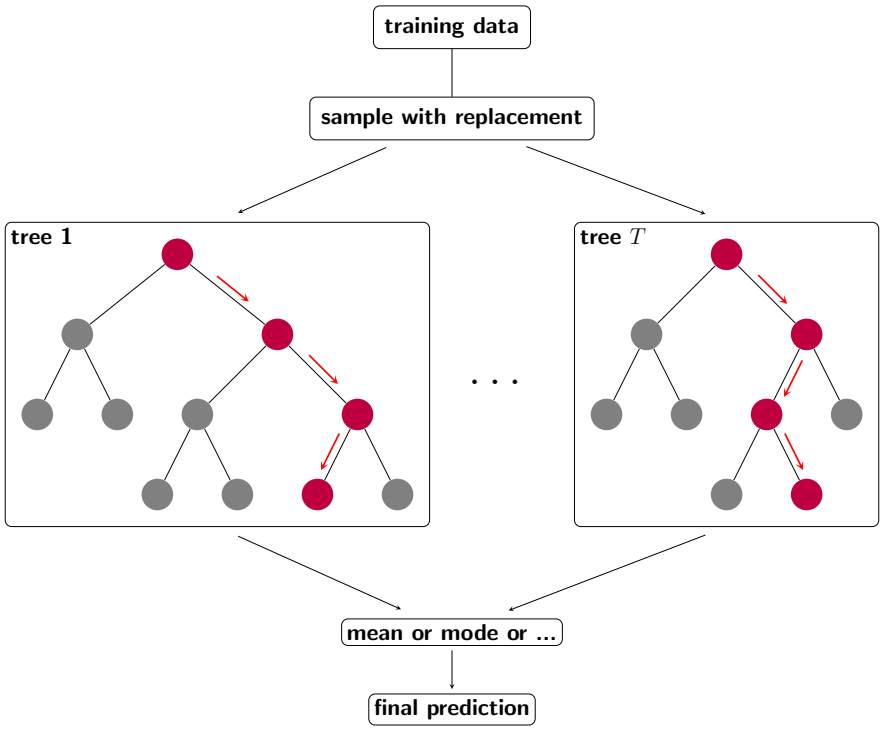

Figure 3: Schema of a random forest based on T binary decision trees.

4 Experimental Results

The following section shows the outcomes obtained as a result of some tests performed in order to verify the performance of the implemented algorithms.

4.1 About tree predictors

The first test, the results of which are reported in the two tables and in the two charts below, was performed with the aim of studying how the expected value of the risk (ℓ_D) of the learning algorithm (A) varies as a function of the stop threshold, both when the height of the tree is limited and when the number of nodes is constrained. All the remaining hyper-parameters were instead kept constant, in particular, the mode was used to assign a label to each leaf, the number of thresholds tested per node was limited to five and the zero-one loss was used as the loss function, while the entropy as the split criterion. Obviously, to estimate $\mathbb{E}[\ell_D(A_\theta(S))]$, where S is the dataset, for all the 12 combinations of the hyper-parameters θ considered, the k-folds cross validation was applied with $k = 5$.

Max height	MTRE (%)	MTSE (%)
5	31.361	31.387
10	19.873	19.945
15	8.242	8.342
20	3.627	3.686
25	0.231	0.262
30	0.000	0.041

Table 3: Data shown by the plot 4, where MTRE is the mean training error and MTSE is the mean test error.

Max nodes	MTRE (%)	MTSE (%)
16	29.916	30.035
32	24.467	24.539
64	16.284	16.409
128	5.858	6.011
256	0.068	0.009
512	0.000	0.004

Table 4: Data shown by the plot 5, where MTRE is the mean training error and MTSE is the mean test error.

As the plots 4 and 5 clearly show, the more the tree can grow, the more the mean test error (MTSE), which is an estimate of the expected risk, decreases, until trees that correctly classify virtually all the data are produced. An extremely significant aspect is that, most of the time, excellent performance are obtained using trees that always have about 173 leaves, regardless of the fact that, in many cases, the maximum size set has not yet been reached. An example of this is given here:

```
---- TREE ----
tree -> {
    'id':0,
    'prediction_criterion':'mode',
    'split_criterion': 'gini',
    'stop_criterion':'(max_height, 25)',
    'num_nodes': 343,
    'height': 23,
    'max_features': None,
    'max_thresholds': 5,
    'root': 0,
    'num_leaves': 172
}

[...]
---- ---- ----
Training error:0.0
Test error:0.00051
```

This is very important because 173 is the number of mushroom species in the dataset. Therefore, during the training phase, the algorithm basically learns to recognize the different species and then classifies the mushrooms of each species as edible or poisonous. This also justifies the fact that the model does not suffer from underfitting and not even overfitting (as shown by the tables 3 and 4), and proves that the techniques put in place to limit it actually work.

The first expedient used to control the overfitting is definitely to limit the tree growth, while the second is the term $\frac{|l|}{m}$ added in the formula 4 used to compute the goodness of a split, which means, as mentioned in section 3.2, that, for the same information gain, it is preferable to apply the split that involves the greatest number of data. All this ensures that, although the numerical features have not been discretized to avoid introducing a new variable, there is no risk of applying a series of splits based uniquely on these features to route each data point to a different leaf, because this would involve an enormous number of nodes, especially in a binary tree, and a huge number of splits involving less and less data.

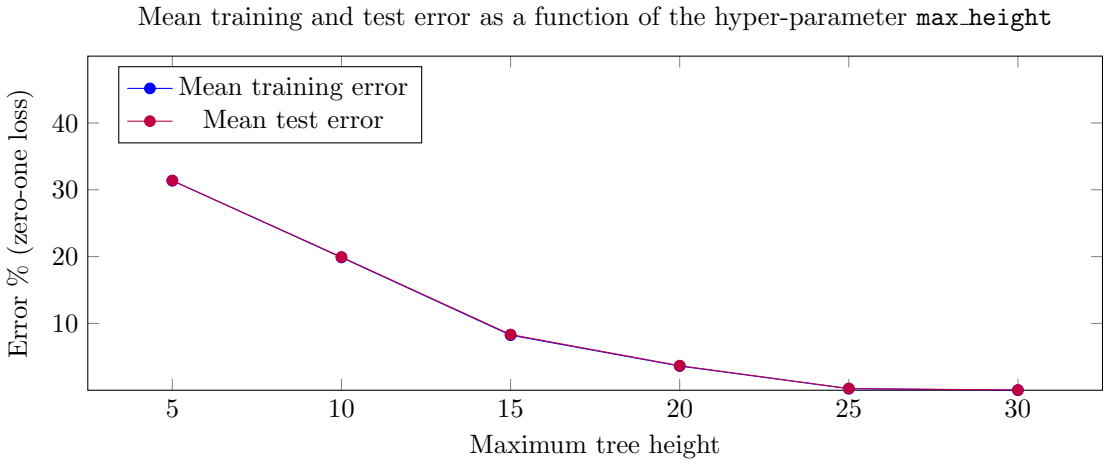

Figure 4: The plot shows the impact of the hyper-parameter `max_height` on the average training error and the average test error computed by applying the 5-folds cross validation. The loss function used is the zero-one loss, while the split criterion is the entropy.

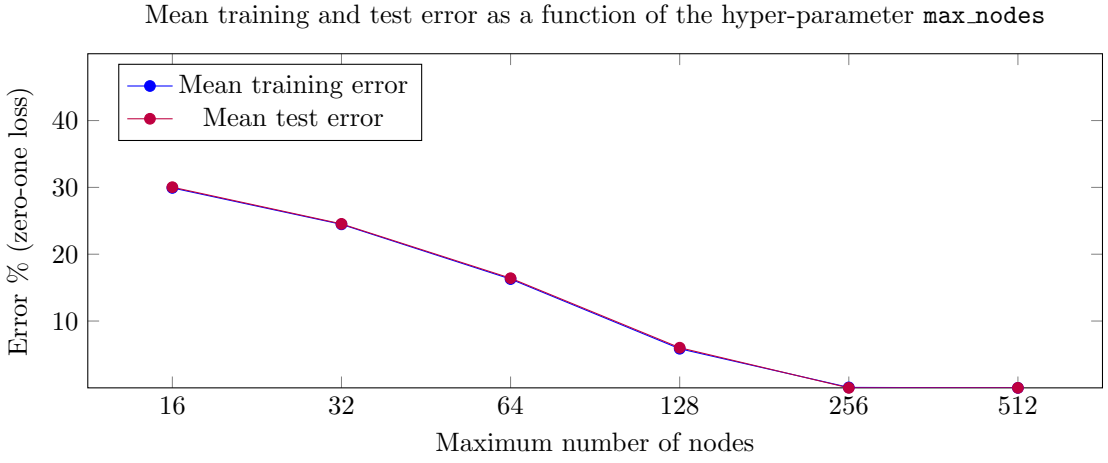

Figure 5: The plot shows the impact of the hyper-parameter `max_nodes` on the average training error and the average test error computed by applying the 5-folds cross validation. The loss function used is the zero-one loss, while the split criterion is the entropy.

The second test, on the other hand, compares the expected risk of decision trees that are using different split criteria while the maximum allowed size changes. The results obtained are reported in the table 5 and displayed in the plot 6.

Max nodes	Entropy		Gini Index		Misclass	
	MTRE (%)	MTSE (%)	MTRE (%)	MTSE (%)	MTRE (%)	MTSE (%)
96	9.137	9.196	9.095	9.206	11.641	11.685
128	5.858	6.011	4.809	4.912	9.936	10.000
160	2.508	2.558	2.525	2.617	9.010	9.157

Table 5: Mean training error (MTRE) and mean test error (MTSE) of the three split criteria computed by applying the 5-folds cross validation. The loss function used is the zero-one loss.

The data collected show that entropy and Gini impurity index lead to similar results, with the Gini index being slightly better in some circumstances. On the contrary, the misclassification index, always has significantly worse performance than the other criteria and, above all, the average error decreases much more slowly as the maximum number of allowed nodes increases.

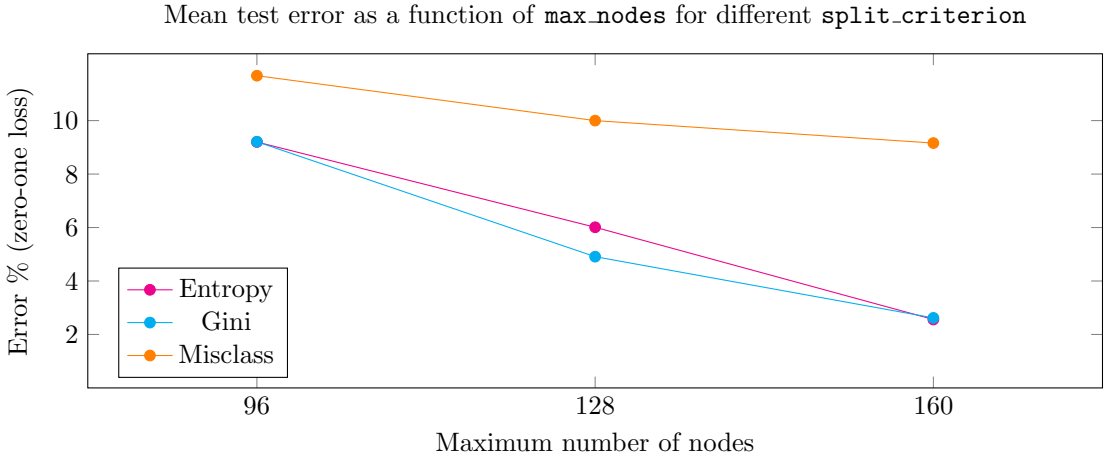

Figure 6: The plot shows the average test error, computed by applying the 5-folds cross validation, of the three split criteria as the maximum number of nodes allowed changes.

This behavior should not be surprising, however, because it is in line with a well-known characteristic of this index: the inability to evaluate some splits as favorable just because they do not lead to a reduction in the number of misclassified data, without rewarding the fact that they can lead to pure leaves, for example. This explains why even increasing the stop threshold, the improvement is small: because regardless of the threshold chosen, the training phase ends before reaching the maximum size because no split is seen as convenient. This is also demonstrated by the logs reported here:

```
2024-10-04 15:21:05,434 - WARNING - bintreepredictor -
    BinTreePredictor_id:8 - no split found

[...]

2024-10-04 15:29:05,545 - WARNING - bintreepredictor -
    BinTreePredictor_id:9 - no split found
```

4.2 About random forests

Regarding random forests, it is known that one of their most important properties is the ability to reduce overfitting. As the tests reported in the section 4.1 have shown, the implemented decision trees are able to solve the binary classification task under consideration without overfitting problems; in fact, the difference between the test error and the training error is always small and entirely reasonable. However, this, on the other hand, makes it difficult to prove the effectiveness of the implemented random forests.

Forest size	Training error (%)	Test error (%)
1	0.000	22.447
8	0.052	22.384
16	0.012	17.695
32	0.011	16.241
64	0.000	15.837

Table 6: Training and test error as the number of trees that compose the forest changes. Note that the first row reports the performance of a binary decision tree, not a random forest.

In order to show their effectiveness, it was hence chosen to force the learning process and consider an extreme case: the training set was reduced to only 250 samples (in previous tests it was 80%

of the data) and the maximum height of the trees has been set at 15, this allows the trees to have at most $2^{15} - 1$ nodes of which 2^{14} leaves: an enormity given the small size of the training set.
With these values, and using the Gini impurity index as the split criterion, four forests, each composed of a different number of trees, and one binary tree were trained. The training error and testing error of these predictors are shown in the table 6 and in the plot 7.

The training error always turns out to be zero, or almost zero, while in all the predictors the test error is significantly higher. But, even in the chart, it is easy to see that as the size of the random forest increases, the test error decreases, which implies that overfitting is also decreasing, always obviously within the limits of what is possible given the extreme case considered. This, therefore, proves the effectiveness of the implemented random forests in controlling the overfitting.

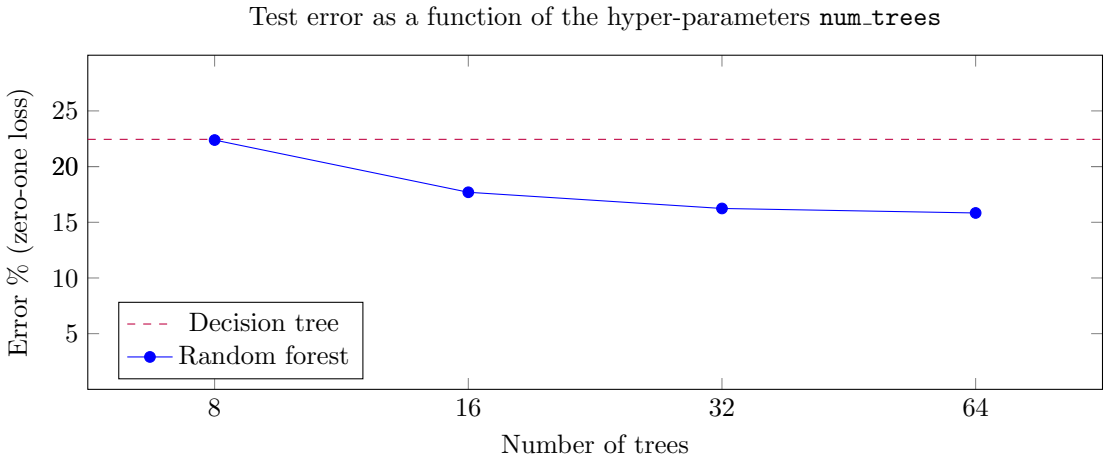

Figure 7: The plot shows the test error as the number of trees changes: the blue line represents the performance of random forests, while the dashed red line used as a benchmark represents the performance of binary trees.

5 Conclusions

In conclusion, given the results obtained, we can consider the binary classification problem solved: both binary decision trees and random forests have been shown to be able to correctly label mushrooms as edible or poisonous with excellent accuracy.
Obviously, several improvements remain possible: from extending the available criteria, so that it is possible to solve other types of problems, to implementing more sophisticated algorithms for the training of the models, or the code could be modified in order to return also an estimate of the confidence of each prediction. Another key aspect overlooked here is time complexity: performance, in terms of execution time, especially of the training phase, can be greatly improved by exploiting parallelism. In fact all the most time-consuming functions could in theory be executed in parallel: see, for example, the search for the best split, or the training of T decision trees performed during the training phase of the random forests. So, if the training were performed on GPU, for instance, the computation time would benefit greatly.

References

[1] Scikit-learn. Decision trees. URL: https://scikit-learn.org/stable/modules/tree.html. [Accessed 12-Oct-2024].

[2] D. Wagner, D. Heider, and G. Hattab. Secondary Mushroom. UCI Machine Learning Repository, 2021. DOI: https://doi.org/10.24432/C5FP5Q.

Declaration

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.