Algorithms For Massive Dataset Exam Project
Tree Predictors for Binary Classification

Riccardo Raffini - 45999A

July 2024

Contents

1. Introduction

The purpose of this report is to illustrate and discuss the methodologies employed in the realization of a project, which main task is the implementation from scratch of a tree predictor model able to perform binary classification activities and evaluation of its performances on a specific given dataset.
In addition, the optional objective of implementing random forests starting from the previous result is achieved too.

The report is organized in different chapters and sections each one describing a significant aspect of the project and the steps that were performed in order to reach the objectives or the carried out experiments. In the next chapters, the considered dataset of examples, a possible implementation of a tree predictor and its usage for the specified task will be described. And at the end there is a chapter describing random forests.

All the structures and procedures described within the report are implemented using Python 3 programming language and collected in a public GitHub repository in turn organized in modules, available at this link. A copy of this report is also included in the repository.
Examples of use of the implemented code are available either in the form of simple scripts or in *Jupyther notebook* format, their content and results will be covered thought this report.

2. Dataset

2.1 Original dataset

The classification activities described in this report are carried out considering the **Secondary Mushroom** dataset, whose files and description are publicly available at this page. The considered version of the dataset is the one available at June 2024, however copies of the employed files are also included in the project repository under the directory `datasets/`.

The source dataset is made up of two main *CSV* files, namely *primary* and *secondary*, but only the latter is analyzed and will be employed. This file contains a total of 61.069 examples that were *synthesized* (artificially generated) from the real ones inside the primary file, additional information about the generative process are available in the original article [3].

Each example represents an hypothetical mushroom and it is characterized by **20 features**, belonging to **continuous** and **categorical** domains, plus a **categorical label** indicating whether the mushroom is *edible* or *poisonous*, with an approximate distribution of the two categories close to 55% and 45% respectively.
However either by reading the description on the linked page or by directly loading the CSV file containing the data, it is clear that this dataset is incomplete, meaning that there are **missing feature** values among the examples. More precisely, a summary of the percentage of missing features is shown in table 2.1.

For this reason it is necessary to carry out transformations and cleaning processes on the data before they can be used for the project purpose of classification using binary tree predictors.

Feature name	Type	Missing values (%)
class	categorical	0.00
cap-diameter	continuous	0.00
cap-shape	categorical	0.00
cap-surface	categorical	0.23
cap-color	categorical	0.00
does-bruise-or-bleed	categorical	0.00
gill-attachment	categorical	0.16
gill-spacing	categorical	0.41
gill-color	categorical	0.00
stem-height	continuous	0.00
stem-width	continuous	0.00
stem-root	categorical	0.84
stem-surface	categorical	0.62
stem-color	categorical	0.00
veil-type	categorical	0.95
veil-color	categorical	0.88
has-ring	categorical	0.00
ring-type	categorical	0.04
spore-print-color	categorical	0.90
habitat	categorical	0.00
season	categorical	0.00

Table 2.1: Features, types and percentages of missing values with respect to the total number of examples. Features presenting missing data are highlighted in yellow.

2.2 Data pre-processing

Further analysis of the dataset shows that the whole collection of examples is affected by at least one missing value, so the idea of simply discarding the affected examples is not applicable in this situation, as this would leave an empty dataset, thus more suitable processing techniques should be employed.

Missing data is a common challenge in machine learning applications, as it can affect the quality and the performance of models, some of the main approaches that can be used for handling missing values in a dataset are **deletion**, **imputation** and **learning**.
Although a learning strategy could be interesting, it may be too complex and inadequate for an unnatural generated data domain.

So a strategy that mixes the effects of both the deletion and imputation is chosen as alternative. Rather than removing affected examples, it might be better to remove features, but considering also the fact that some of the affected ones do not have very high missing percentages, deleting all them indiscriminately could be a too extreme decision. Thus features are deleted observing whether or not each of them has a *significant* quantity of missing values, where the importance can be formalized in term of a **threshold** value that implies the dropping of a feature if and only if its quantity is above this chosen value.
After this first reduction, it is possible that some examples still contain missing components, but if they have passed the previous phase, it means that the quantities related to such components are not so high to determine the deletion of the columns. Hence to complete the cleaning process, imputation is used to fill these remaining values with estimated or substituted ones. There are many ways of filling features, but the most simple and customizable strategy is the possibility of providing a generic value computed following any approach.

In this way, the advantage is that no example is removed from the dataset, preserving its size, but deleting and adding data have some **drawbacks** that can influence the generalization and prediction capabilities of models, because they introduce distortion in the original data distribution.

This cleaning process is translated into the following functions named `delete_dataset_features()` and `fill_dataset_samples()` defined inside the module `datasets.methods`:

```
def delete_dataset_features(dataset:pd.DataFrame, threshold:np.number) ->
↪ pd.DataFrame:
    samples_threshold = dataset.shape[0] * threshold
    null_columns = np.array([column_name for column_name in dataset.columns if
    ↪  dataset[column_name].isnull().sum() >= samples_threshold])

    new_dataset = dataset.drop(null_columns, axis=1)

    return new_dataset

def fill_dataset_samples(dataset:pd.DataFrame, filling_value:Any) -> pd.DataFrame:
    new_dataset = dataset.fillna(filling_value)

    return new_dataset
```

In the specific case of the considered dataset, the two function are applied with 0.40 as missing feature threshold and, since all remaining features are categorical (and nominal), a new attribute `'u'` (standing for *unknown*) is used as filling value.

To conclude the data pre-processing section, the examples originally loaded as a `pandas` *DataFrame* are turned into more easily manipulable `numpy` arrays and at the same time, the `class` feature (label) can be separated from the other example's features.
The following `extract_samples_labels()` function allows to do so and eventually apply a mapping to the labels, for instance this could be used to turn categorical labels into numerical ones.

```
def extract_samples_labels(dataset:pd.DataFrame, labels_map:Callable[[Any], Any] =
↪ lambda x: x) -> tuple[np.ndarray, np.ndarray]:
    # labels extraction
    labels = dataset['class'].to_numpy()
    labels = np.apply_along_axis(labels_map, 0, labels)

    # samples extraction
    samples = dataset.drop('class', axis=1).to_numpy()

    return samples, labels
```

Since the examples and labels will be used by predictors belonging to the decision tree class, there is no need of further processing the data, also because as written in the dataset description, all the remaining categorical feature are nominal (there is no ordinal feature), additional transformation would make no sense to the end of classification purpose, but this has to be taken in account for the predictor definition and implementation, that should then be able to work with both numerical and categorical features.

The result of these three transformation represent the final dataset of examples and labels that will be used as reference and for carrying out all the training, tests and experiments in the next chapters.

3. Tree predictors

3.1 Tree predictor model

Tree predictors (or decision trees) are a general learning method that can be used in many supervised learning problems, they can be applied to both classification and regression tasks according to the type of the labels. Due to their characteristics, they allow to represent predictive models that are very easy to understand but at the same time very effective.

Decision trees work as any other supervised learning methods, they are trained starting from a set of known examples and labels and since they are non-parametric algorithms, they rely on the seen data to predict outcomes of new data.

When the space of data X is not an uniform space like $\mathbb{R}^d$, usual classification or regression methods based on **hyperplane partition** can no longer be applied, because there is no guarantee that such space is composed of domains that are comparable or that can express a coordinate system, if they have not these characteristics, then it also not possible to apply very simple methods as KNN.
A generic data space $X = X_1 \times X_2 \times \cdots \times X_d$ with X_i as individual feature domains, can in fact include both **numerical** and **categorical** attributes that are not comparable against each other. So a different approach is used in order to still being able to create regions (partitions) of the space X.

As alternative, partitions could be determined by defining **thresholds** values in the feature domains, so that there is no longer the need of performing any comparison between different features, thresholds are defined independently on individual features.
Each threshold corresponds to a *question*, more formally to a **condition** related to the values of a feature i, whose *answers* determine one or more splits in the domain X_i.

There are different kind of conditions, the main grouping characteristic is the number of possible **outcomes** they have, that is because the number of outcomes determines the kind of tree predictors. Predictors using only **binary conditions** are thus called **binary tree predictors**, whereas **non-binary conditions** originate **non-binary tree predictors**, but they can also be distinguished according to to the number or kind of features they work on.
Some examples of commonly used types of binary conditions are *thresholding*, *equality*, *membership* or the more complex *oblique* conditions.

As it can be expected, a tree predictor that is based on non-binary conditions has higher discriminative power, but they are more likely to cause **overfit**, so binary tree predictor are generally preferred to them, although they can suffer of the same problem.

Predictors that work in this context are often referred to as *tree predictors* and represented by trees structures because of their computational behavior. In this optic a predictor is seen as an hierarchical organized structure assuming the shape of a tree origininting from a **root node**, where **internal nodes** express the decision based on a conditional statement, the **branches** coming out of them represent the possible outcomes, while **leaf nodes** do not have any condition in them, but instead they represent final labels for the classification/regression task.

Although a different method based on the thresholding is used, it has some similarities with the more complex hyperplane approach, in fact if data domains are limited to or transformed into numerical features, hyperplanes can be still employed inside the nodes instead of conditional statements.
Thus a tree predictor is as powerful as a hyperplane method, but additionally it can adapt to the the presence of categorical features by employing the thresholding of values.

3.2 Conditional statements

Conditional statements are decision criteria of the form $C_\theta : X_i \to \mathbb{N} \cup \{0\}$, defined on a specific feature domain X_i and depending on some parameters θ, usable inside predictor nodes for splitting the computation in branches. When used on this purpose the decision should act as a map between a feature value and a node's child index, so co-domain could be limited to a finite range $K = \{0, 1, \dots k\}$.
The conditional statements are implemented following an hierarchy of abstract and concrete classes, whose definition is inside the `commons.splitting_criteria` module.
A generic condition is represented by the abstract base class named `Condition`, defining the signature for a common method `test()`, that receives in input a value representing a feature $x_i \in X_i$ and that should be usable to test the feature on a specific **statement** parameterized by θ and returns a partition index $j \in K$ as result.

This abstract class allows to express generic conditional statements having an arbitrary number of outcomes, however for the final aim of the project, binary conditions are sufficient, so a more specific, still abstract, class `BinaryCondition` is defined to represent statements limited to only two possible outcomes $K = \{0, 1\}$, consequently simplified in the Boolean values `False` and `True`, but still usable as partition indices 0 and 1 thanks to the Python language characteristics.

Finally, from this class, some concrete classes implementing binary conditional statements can be derived. For instance, as it is required that statements involve just individual features, some of those conditions can be:

- Thresholding $T_\theta(x_i) = \begin{cases} 1 & \text{if } x_i \leq \theta \\ 0 & \text{otherwise} \end{cases}$

- Membership $M_\theta(x_i) = \begin{cases} 1 & \text{if } x_i \in \theta = \{s_i : s_i \in S_i \subseteq X_i\} \\ 0 & \text{otherwise} \end{cases}$

- Equality $E_\theta(x_i) = \begin{cases} 1 & \text{if } x_i = \theta \\ 0 & \text{otherwise} \end{cases}$

They are respectively implemented in the same module by the classes named `ThresholdCondition`, `MembershipCondition` and `EqualityCondition`.
Of course these classes just represent a small and partial implementation for the condition hierarchy, that it is sufficient for this project, although it could be easily extended. This partial hierarchy is expressed in a simplified representation in figure 3.1.

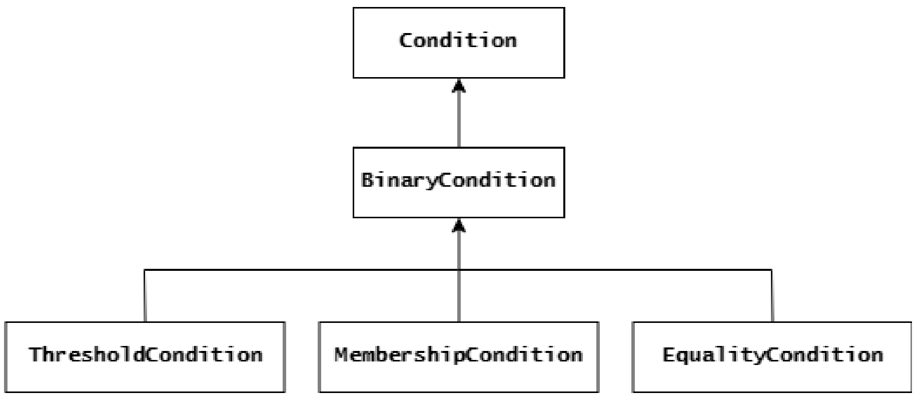


Figure 3.1: Partially extended hierarchy of the conditional statements.

3.3 Tree nodes

The base building blocks for a tree predictor are represented by tree nodes. As said before, a node can either represent an internal node or a external (leaf) node, the main difference between these two kinds of nodes is that the matter is characterized by the presence of a conditional statement (decision) inside, whereas the latter is characterized by a simple information holding the final label.

In term of implementation, this description is translated into a class named `TreePredictorNode` whose relative code is defined inside the module `predictors.treepredictors`. Each node has several attributes, namely an integer representing the **index** of the feature on which can operate, a **condition** on which its decision is based, a value representing the **label** and a list of **child nodes**.
But not all of those attributes are required to be initialized, the difference between the two kind of nodes is achieved according to which values were passed to the constructor: if the feature index and the decision attributes are initialized then it is an internal node, otherwise if only the label attribute was provided, it means that this is a leaf node. This difference can also be checked at any time by using the `is_leaf()` method, returning the answer by simply performing the same checks.
As the constructor of a node accepts any kind of conditional statement, the result of the initialization is a **generic node** whose splitting behavior (binary/non-binary and numerical/categorical) is depending on the given condition type.

Since the main purpose of a node is either returning the associated label or splitting data into partitions, the decision of an internal node can be applied to data examples by passing them to the `test()` method, that in turn calls the `test()` method of the associated condition passing the required feature value, returning its result, corresponding to the index of one of the node's children.

```
def test(self, sample:np.ndarray) -> None | int:
    if self.is_leaf():
        return None

    feature_value = sample[self.feature_index]
    condition_value = self.condition.test(feature_value)

    return condition_value
```

In case of leaf nodes, instead, the final label can be obtained by simply accessing the relative attribute.

3.4 Tree predictors

The implementation of tree predictors follows from the one of tree nodes, since nodes are generic, also the obtainable tree predictors are generic, but next section will explain how this generic definition could be exploited to achieve the required project objective of binary trees for binary classification.

Tree predictors are implemented by the class `TreePredictor` inside the `predictors.treepredictors` module of the project.
This class is characterized by the two methods commonly named `fit()` and `predict()`, whose behavior is almost intuitive and that will be described in the following subsections and that allow to obtain a fully functional tree predictor starting from a collection of examples.
The generic aspect of the implementation is mostly due to the constructor of the class that allows to customize part of the *computational behavior* of the tree, as well as to initialize other attributes used by the underlying tree structure. All the following things can be specified and customized:

- The **conditional statement** to employ in internal nodes when the decision consists in evaluating **continuous features**.
- The **conditional statement** to employ in internal nodes when the decision consists in evaluating **categorical features**.
- A **decision metric** for evaluating and scoring decisions on the features that represent possible splitting point during the construction of the tree.
- A list of **stopping criteria** that use the whole **tree** structure for determining an halt in the construction of the underlying tree. By default, if no criterion is provided, a criterion that assumes a maximum tree depth equal to 100 is used.

- A list of **stopping criteria** that use individual **nodes** for determining an halt in the construction of the underlying tree. By default, a criterion that stops the expansion of a node when the impurity (entropy) of its samples is equal to zero is always added.

In a more abstract view, those parameters of the class represent hyperparameters of the predictive model, that require to be specified by users or tuned in order to achieve the best performances. This aspect will be further explored in next chapter.

3.4.1 Stopping criteria

Stopping criteria are also implemented through a hierarchy of class similar to the conditional statements one, with a base abstract class `StopCondition` specializing in the two `TreeStopCondition` and `NodeStopCondition` (still abstract), with the purpose of providing common interfaces for the two kinds of condition involving either the whole tree or a node.

Many different stopping conditions can be defined extending this hierarchy, some of which were already provided inside the `commons.stopping_criteria` module, and then used to customize the tree predictor's computational behavior.
More specifically, three stopping conditions regarding the tree are the **maximum depth**, the **maximum number of leaf nodes** and the **maximum number of nodes**, whereas two stopping conditions for nodes are the **minimum number of samples** in the node and the **impurity level of samples** in the node. They are all implemented by a distinct class.

A general idea of this hierarchy is provided in the figure 3.2.

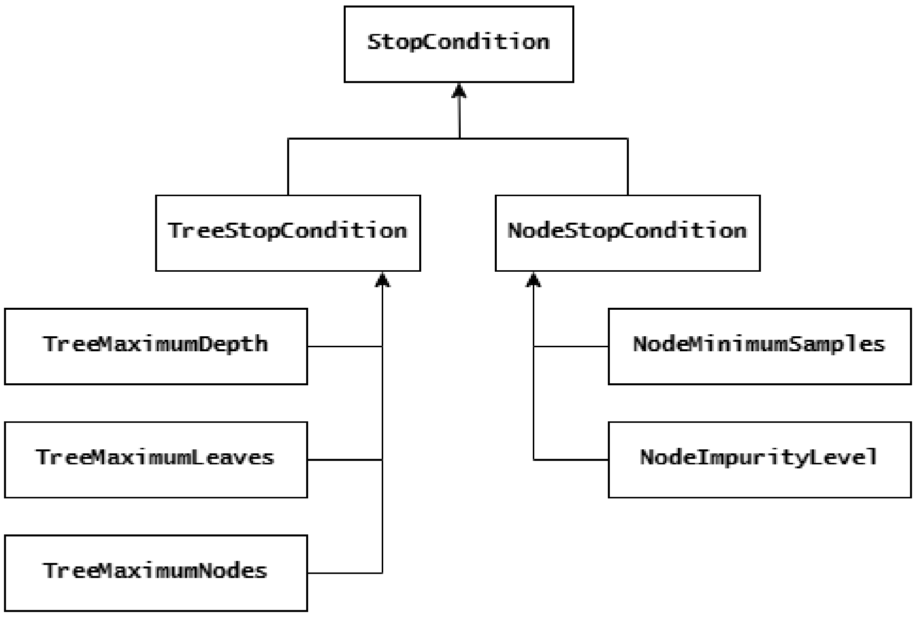


Figure 3.2: Stopping criteria hierarchy, expanded to include some stopping condition regarding both individual nodes and entire tree.

3.4.2 Training of the model

Training a tree predictor is considered a *NP-hard* problem, this implies that it is almost impossible to obtain a perfectly trained model with a test error, or any other evaluation metric, exactly equal to zero (**optimal predictor**), but heuristics as **greedy** approaches allow to still obtain good approximations with low error.

The provided implementation of training procedure is mainly inspired by the **ID3** algorithm [1] and by some aspects of the **C4.5** algorithm [2].
Following a greedy approach, the tree is built gradually, making local best choices that guarantee the best expansion of the structure at each step of construction. The following snippets of code represent the training procedure used by the `TreePredictor` class.

```python
def fit(self, samples:np.ndarray, labels:np.ndarray, verbose:bool = False) -> None:
    ## Information about sample
    _, samples_dimensionality = samples.shape
    self._features_number = samples_dimensionality

    ## New tree initialization
    common_label = self._common_label(labels)
    self._root = TreePredictorNode(label=common_label)
    self._depth = 1
    self._nodes_count = 1
    self._leaves_count = 1

    ## Tree building
    self._grow_tree(samples, labels, verbose)
```

The training process starts by calling the `fit()` method, which receives as argument a **training set** of examples $S = \{(x_1, y_1), \dots, (x_m, y_m)\}$ (already split in samples and labels). The first construction step consists in cleaning any previous information and underlying tree structure and initialize a new tree by adding a single root node r, that is a leaf node having the most common label $\tilde{y}$ within the entire training set S.

The next step consists in gradually expand this newly created root into a tree, by calling `_grow_tree()`, another method inside the class. Of course at the beginning the node r is the only one available for expansion, but more in general, in successive steps, there will be multiple leaf nodes $\mathcal{L} = \{l_1, l_2, \dots\}$ available, thus it is necessary to select which leaf to expand. This choice could be taken in many ways, but in the provided implementation there is no real selecting criterion, or more precisely, every leaf node $l_i \in \mathcal{L}$ becomes a **candidate** node. However nodes can only be expanded one at time, hence at this end a **queue** of leaves $Q_{\mathcal{L}}$ is introduced; it holds not only the leaves themselves but also some other information, specifically the **associated examples** (their indices) and the **depth levels**, that will be useful later.
The overall effect of this choice is then that the tree grows level by level from left to right.

Although **all** nodes are selected for expansion, there is no guarantee that the growing tree will be *complete*, in fact the first action performed on each candidate leaf l_c inside the main loop is checking all stopping criteria (previously provided in the constructor), both the conditions regarding the node and the overall tree status. If any of the stopping condition is verified, then the expansion of this particular node (and thus its branch) is interrupted.

If instead l_c passes all the stopping criteria, it means that is suitable for the expansion. The expansion of a leaf node consists in turning it in a internal node having a decision condition rather than a label and creating its new child nodes.
The best decision condition for a node is found by considering the set of examples $S_c = \{(x_j, y_j)\} \subseteq S$ associated to it and exploring all the features $i = 1, 2, \dots, d$ they have.
Using the provided scoring metric for the decisions, every feature i is considered individually, testing several parameters $\theta \in X_i$ for the conditional statement C chosen according to the feature type, returning the feature i^* and the parameter θ^* for a condition C that assure that maximization of the scoring metric.

This part of the tree construction is the one having the most significant impact on the **time complexity** of the training procedure, because it is necessary to find a condition for each node and in the worst case (each node split data in two branches, one with a single example and all the rest in the other) the depth of the tree is equal to the number of training examples m. Finding a condition requires to analyze all values for the d features representing possible parameters, categorical features usually have a constant number of values whereas continuous can have many, up to a distinct value for each example, but usually with a simple trick (finding midpoint of unique values), their number can be reduced and so the scoring metric is applied to a subset of values.
Part of the conditions exploration is reported in the following snippet of code:

```
def _find_best_condition(self, samples:np.ndarray, labels:np.ndarray,
↪  available_feature_indices:list[int], verbose:bool = False) -> tuple[int, Any,
↪  bool]:
    # ...

    for feature_index in available_feature_indices:
        feature_values = samples[:, feature_index]
        possible_parameters = np.unique(feature_values)
        is_continuous = np.issubdtype(type(feature_values[0]), np.number)

        if is_continuous:
            # find midpoints ...

        for possible_parameter in possible_parameters:
            if is_continuous:
                condition = self._continuous_condition(possible_parameter)
            else:
                condition = self._categorical_condition(possible_parameter)

            parameter_score = self._decision_metric(feature_values, labels,
            ↪  condition.test)

            if parameter_score > best_condition_score:
                # save best parameter ...

    return best_feature_index, best_parameter, is_best_feature_continuous
```

Once the best decision for the parent node l_c is found, it becomes trivial to convert it in an internal node and to build the new child leaf nodes. The decision is simply applied to the associated samples producing different **partitions** $S_i \subseteq S_c$ and for each of them a new leaf node, whose label is the majority class $\tilde{y}_i$ in that partition S_i, is linked to the parent, extending so the tree structure and then it is also added to the leaves queue $Q_{\mathcal{L}}$ in order to become a next candidate of the expansion.

The training procedure is completed when the queue becomes empty, meaning that the tree cannot further be expanded, either because all nodes are fully expanded or stopping criteria halted the construction.

3.4.3 Predicting using the model

Assuming that the training of the tree predictor model is completed, its instance will now hold an underlying decision tree that can be used by calling the method `predict()` for predicting labels. This method accepts collection of data points $\{x_1, \ldots, x_n\}$ and applies the following procedure to each of them, therefore it returns a collection of predicted labels $\hat{y}_t \in Y \ \forall t = 1, \ldots, n$.
Once a new data point represented by a vector $x_t = (x_1, \ldots, x_d)$ is provided to the predictor, its class label is predicted visiting the tree.

The procedure consists in traversing the underlying tree starting from its root node r, applying the conditional statement of each internal node visited on one of the features x_i, providing the considered data point x_t as argument to node's `test()` method and using its returned value as branch to follow in order to move to the next child node to visit.
This mechanism is repeated until the currently considered node is not a leaf, as soon as one leaf is reached, then visit ends by returning the label $\tilde{y}$ that is associated to this leaf node, $\hat{y}_t = \tilde{y}$.

This procedure can be implemented in many ways, but an iterative approach is probably preferred to a recursive approach because of the unknown depth reachable by the tree, that as suggest before can reach a depth equal to the number of training examples m, but it can be assumed that each node test require just a constant time. The following lines of code show a possible implementation of the `predict()` method based on an iterative traversing of the underlying tree:

```
def predict(self, samples:np.ndarray) -> np.ndarray:
    samples_predictions = np.array([self._traverse_tree(sample) for sample in
    ↪  samples])

    return samples_predictions

def _traverse_tree(self, sample:np.ndarray) -> Any:
    current_node = self._root

    while not current_node.is_leaf():
        next_node_index = current_node.test(sample)
        current_node = current_node.children[next_node_index]

    final_label = current_node.label

    return final_label
```

This way of predicting class labels make really easy to **explain** or interpreter decision trees behavior, a label assignment is due to the **path** followed during the visit of the tree, even when the tree grow very large. However this strength also represent a point of **instability**, a small variation in the choice of training samples may lead to an entirely different underlying decision tree and thus to possible different classifications and performances.

3.5 Binary tree predictors for binary classification

As said in previous sections, the provided implementation describes a **generic** tree predictors, that is the more general version of predictors able to work with any condition, in terms of their outcomes number, their types and also feature types.

The specific predictor required by the asked project task of classification can be achieved by forcing the following **restrictions** on the class instances: using only binary conditions operating on single features and by considering only data assigned to binary classes.
First, using binary conditions assures that the tree underlying the obtained predictor is a binary tree, a tree whose nodes can have exactly either zero or two children nodes, because a node is either a leaf with no child or it is an internal node whose decision can produces exactly two branches and thus two children.
The second restriction is almost intuitive and acts on the results of classification, as the predictor is trained on data having only two classes, as consequence the obtained tree can only produce leaf nodes predicting such two classes, thus the predictions will be binary.

Since the provided implementation of conditional statements has only binary outcomes statements and since the considered dataset has only to class labels, every tree predictor trained using them will be a binary tree predictor producing binary classifications.
A simple proof of this is given by a necessary condition on nodes, every tree predictor trained by the previously described procedure returns trees with a number of total nodes T and leaf nodes L in the relation $L = \frac{T+1}{2}$, typical condition of binary trees.

4. Training models

4.1 Decision metrics

The expansion step of a nodes in the training procedure of tree predictor requires a decision metric for **ranking** the possible decision statements and the partitions they create. A metric is an evaluation criterion that in turn is based on a so called **index** that assigns a numerical value to each partition so that then they can easily be compared or used in metric computation. As commonly used indices tend to be a measurement for a *negative aspect* of the partitions, usually the behavior consists in minimizing the metric values of the resulting partitions, but as done in the project, there are many equivalent computations, one of which is the maximization of the **gain** of the split.

Given an index function $\mathcal{I} : S \to \mathbf{R}$, defined on sets of data (partitions) S, the decision metric returning the gain $\mathcal{G}$ of a split can be computed as the difference between the index computed on the original partition S and the weighted sum of the indices computed on the new partitions $\{S_0, S_1, \dots S_k\}$ resulting from the split, where the weight w_i is usually given by their relative cardinality:

$$\mathcal{G} = \mathcal{I}(S) - \sum_{i=0}^{k} w_i \mathcal{I}(S_i) \qquad \text{with} \quad w_i = \frac{|S_i|}{\sum_{j=0}^{k} |S_j|} = \frac{|S_i|}{|S|}$$

In the project, this gain is implemented by the function `measure_gain()` in the module named `commons.splitting_criteria` and moreover, the following index functions are available and the code relative to their implementation is available in the same module:

- Minimum $\quad M(S) = \min_{i=1,\dots,n} p_i$
- Entropy $\quad H(S) = -\sum_{i=1}^{n} p_i \cdot \log_2 p_i$
- Gini impurity $\quad I_G(S) = 1 - \sum_{i=1}^{n} p_i^2$

Here n represents the number of distinct class labels characterizing the problem while p_i is the frequency of a class i inside the considered partition S.

Any index can then be applied to implement some specific decision metrics for the training of tree predictors, since the `TreePredictor` class follows the equivalent definition of gain maximization, the metrics have to be defined so that the split giving the highest gain is selected.
Three commonly used gain metric are defined after the three previously described indices, they are thus named respectively **misclassification gain**, **information gain** and **Gini impurity gain**.

An alternative decision metric is the **Chi-Square** value, whose computation is slightly different from the base gain definition, but still suitable for the maximization behavior of the implementation. The Chi-Square $\mathcal{X}_C(S)$ of a decision C is computed as the sum of Chi-Square $\mathcal{X}_S(S_j)$ of resulting partitions S_j, in turn computed as the sum of Chi-Square $\mathcal{X}_S(S_j, i)$ of each class i:

$$\mathcal{X}_C(S) = \sum_{j=0}^{k} \mathcal{X}_S(S_j) = \sum_{j=0}^{k} \sum_{i=1}^{n} \mathcal{X}_S(S_j, i) \qquad \text{with} \quad \mathcal{X}_S(S_j, i) = \sqrt{\frac{(p_i^{(S_j)} - p_i^{(S)})^2}{p_i^{(S)}}}$$

These metrics are all available inside the module `commons.splitting_criteria` by homonyms functions, anyway one can still use the generic function named `measurement_gain()` for implementing additional decision metrics by simply providing new indices.

4.2 Exhaustive training

As first practical application of the implemented classes, four different tree predictors are trained on the whole pre-processed mushroom dataset using the four decision metrics presented in the previous section. The train of these trees is also characterized by the fact that no additional stopping criteria is employed (except for maximum purity in the nodes).
Consequently, the results are some *exhaustively expanded* (but not complete) decision trees, distinguished by their decision metrics. Those trees can then be seen as *reference models* for the successive applications, because they are fully expanded, they highlight the maximum depth and number of nodes reachable using the examples in the dataset; for instance, hyperparameters tuning can exploit those prior observation to conduct *cross validation* on more suitable ranges of values.

More specifically a **configuration** of hyperparameters contains all the parameters characterizing the construction of a `TreePredictor`'s instance and the common values that are set for training the four models are

- `continuous_condition` is set to `ThresholdCondition`
- `categorical_condition` is set to `MembershipCondition`
- `node_stopping_criteria` is set to `[]` (empty list)
- `tree_stopping_criteria` is set to `[]` (empty list)

whereas the `decision_metric` hyperparameter was set individually for each tree, to any of the four possible metric functions `misclassification_gain()`, `information_gain()`, `gini_impurity_gain()` and `chi_square()`.

All the training steps performed and the code relative to this experiment are available in form of a Jupyter notebook or a simple script named `training.ipynb` and `training.py` in the main directory of the project repository.
The characteristics of the underlying trees are reported at the end of each cell and in table 4.1.

Decision metric	Tree depth	Total nodes	Leaf nodes	Train error	Test error
misclassification gain	15	2027	1014	0.017	0.036
information gain	28	773	387	0.000	0.004
impurity gain	28	837	419	0.000	0.005
chi-square	27	1119	560	0.000	0.007

Table 4.1: Decision metric, depth, number of total nodes and number of leaf nodes characterizing the decision tree underlying the exhaustively trained models and their train/test errors.

4.3 Evaluation and hyperparameters tuning

Although a simple computation of train/test error and confusion matrices in figures 4.1 and 4.2 show that previous trees work well, their exhaustive training requires some time and also there is no guarantee that the achieved results are the best.
There are many different ways of getting a better evaluation of the performances of a tree and find the hyperparameters that provides the best results, in this section the two commonly used **cross validation**, **nested cross validation** and other variants will be explored.

All the function implementing the methods that will be described are available in the project's module `commons.losses` and examples of their uses are provided in a Jupyter notebook or a script under the names of `evaluation.ipynb` and `evaluation.py`.
Since some of those methods are computationally expensive, a small subset of the mushroom dataset, consisting of just 10.000 examples was randomly extracted, so that the training phase of the models can be completed in few minutes.

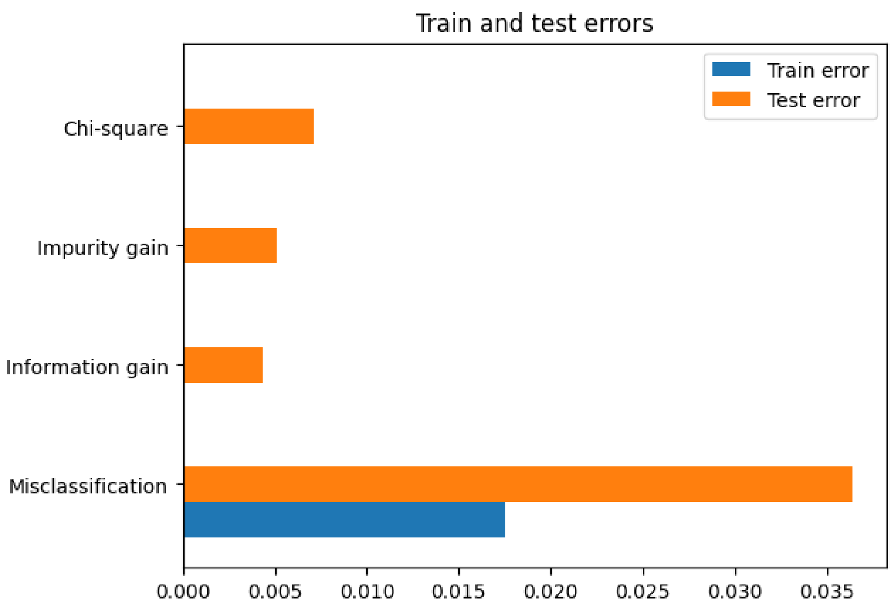


Figure 4.1: Graph representing train and test of the four exhaustively trained models.

4.3.1 Train and test split

Having fixed the set hyperparameters θ, the simplest way of **evaluating** a model h_θ consists in determining how good (or bad) it is at predicting labels $\hat{y}$ for a set of examples $S = \{(x_1, y_1), \dots, (x_n, y_n)\}$ that it has never seen before, where the goodness is usually measured as the number of the correct prediction over the total (**accuracy**).
Or it can be evaluated by taking in account a loss function $l(y, \hat{y})$ comparing a predicted label $\hat{y} = h_\theta(x)$ and a real label y of an example (x, y), by computing the average of the losses on all the examples as $l_S(h_\theta) = \frac{1}{n}\sum_{t=1}^{n} l(y_t, h_\theta(x_t))$, a simple evaluation metric is obtained. For this project the considered loss function is the **zero-one loss** $l_{0/1}(y, \hat{y}) = I\{y = 1\}I\{\hat{y} = -1\} + I\{y = -1\}I\{\hat{y} = 1\}$, a suitable loss for the classification task solved.

Thus the dataset of examples is simply partitioned in two splits taking the name of **train set** and **test set** (commonly with a ratio 80/20), with the matter used for training the model and the latter used for evaluating it. Average of the losses can be computed on both partitions and resulting values are respectively named **train error** and **test error**, however depending on the model type, these two values are not always equivalently meaningful.

Under the independent and identically distributed (*i.i.d.*) assumption of test set sampling over the (unknown) data distribution $\mathcal{D}$, the test error $l_S(h_\theta)$ represents an estimator for the true **statistical risk** $l_\mathcal{D}(h_\theta)$ of the model, so controlling the mean test error implies controlling the statistical risk.

The previously described exhaustive models and the model trained in the first section of the new notebook were all evaluated using this method and their train/test errors are also reported in table 4.1. However the train/test split method is sensible to the selection of the split, first in term of partition sizes, but mostly in term of partition samples, different ordering and cut points may originate completely different predictors and consequently different performances.
Results in the notebook show that even when considering a very small subset of examples, a tree predictor with unconstrained configuration of hyperparameters is able to well generalize the decisions that lead to correct predictions. As it can be seen train error is exactly equal to 0.0 while test error is very low at 0.017 (of course training error is not really meaningful in case of decision trees).

4.3.2 Cross validation

The previous method allows to compute the statistical risk $l_\mathcal{D}(h_\theta)$ for a sample of data, but ideally the objective is to estimate the **expected statistical risk** $E[l_\mathcal{D}(h_\theta)]$, where the expectation is with respect to any random drawn of training set S usable by an algorithm A for training the predictor $h_\theta = A_\theta(S)$.

But computing the expected statistical risk is complex, so also this quantity is estimated by a method named **cross validation**, usually though a technique named **K-fold cross validation**.

With still a fixed set of hyperparameters θ, the problem of estimating $E[l_{\mathcal{D}}(A_\theta(S)]$ can be solved by dividing the full set S containing m examples into K subsets $\{S_1, \dots S_K\}$ named **folds**, each of which now contains $\frac{m}{K}$ examples.
After this, it is possible to arrange the folds in different ways in order to obtain K different train/test splits of the original set: for each each $j = 1, \dots, K$, the fold S_j is considered as test part while the remaining folds $S_{-j} = S \setminus S_j$ become the train part.
Each train part S_{-j} is then used for training a predictor $h_{(\theta,j)} = A_\theta(S_{-j})$ while each test part S_j is used for computing the test errors $l_{S_j}(h_{(\theta,j)})$.
The final cross-validation estimate is computed as the average of all those folds' errors:

$$l_S^{CV}(A_\theta) = \frac{1}{K}\sum_{j=1}^{K} l_{S_j}(h_{(\theta,j)}) \qquad \text{with} \quad l_{S_j}(h_{(\theta,j)}) = \frac{K}{m}\sum_{(x,y)\in S_j} l(y, h_{(\theta,j)}(x))$$

An example of use of the cross validation estimate method is provided in the notebook. The hyperparameters configuration considered is the same one used for the previous method and the number of folds is $K = 5$. A function named `cross_validation()` receives the model class and the folds composing the dataset and executes the estimation process on them, returning an estimated value of 0.019, that it is really close to the 0.017 value estimated on a fixed train/test split using the previous method.

4.3.3 Find best hyperparameters

All the previously described methods share a strong assumption, that is the fact that the set of hyperparametrs $\theta \in \Theta$ is already fixed and thus their main purpose is that of just evaluating how the models perform. However in practice it also necessary to find or estimate θ, with the aim of finding the hyperparameters set θ^* that guarantees the minimum statistical risk on the considerd set of examples, $l_{\mathcal{D}}(A_{\theta^*}(S)) = \min_{\theta\in\Theta} l_{\mathcal{D}}(A_\theta(S))$.

In general it is difficult to exactly determine the best hyperparameters configuration, but the **nested cross validation** technique is the best variant of the estimator methods that allow to do so. As the name may suggest, the dataset is initially split in K folds, a loop allows to iterate on them and build the train/test parts S_j and S_{-j}. Then the best configuration θ_j^* is optimized locally for each fold j, running an internal (nested) cross validation procedure on the training part S_{-j}. After the optimization a predictor $h_{(\theta_j^*,j)}$ is retrained on the set S_{-j} and evaluated on S_j computing its average error $\varepsilon_j = l_{S_j}(h_{(\theta_j^*,j)})$. The final nested cross validation estimate is computed as the average of these errors $(\varepsilon_1 + \cdots + \varepsilon_K)/K$, while the best configuration is one of the locally optimized during the procedure.
This complete variant of nested cross validation is computationally intensive, so lightweight variants such as the use of a **train/validation split** (also named holdout cross validation) replacing the nested procedure or rather structural variants, like **flat cross validation** [4], are preferred because less computational demanding, but still valid ways of estimating the expected statistical risk.

Independently from the variant employed, the reason of the estimation complexity is due to the hyperparamters space Θ, that can be multidimensional and/or large (even infinite), so generally the minimization is performed over a smaller suitable subset $\Theta_0 \subset \Theta$. This is exactly the case of the hyperparameter space of the `TreePredictor` class, where all possible combinations of construction arguments (configurations) create an infinite *grid space* on 5 dimensions.
However an hyperparameters space (or sub-space) can be explored in different ways: *exhaustively* inspecting all the configurations, *randomly* sampling some configurations or *optimally* by focusing on specific configurations in the space.

In the proposed function `nested_cross_validation()` and its variants, this evaluation method was slightly modified, so that also the best hyperparameters configuration is returned along with the risk estimation. More over as described before, the plain procedure of the nested cross validation is expensive, therefore although its implementation is provided, it is not used for performing the process of hyperparameter tuning. Instead, a function with the name `holdout_cross_validation()` implmenting the train/validation split variant was used. Also the flat cross validation variant is implemented in the module by a function named `flat_cross_validation()`, but as for nested cross validation, its expensiveness does not allow to tune parameters efficiently, however all methods can easily be interchanged as they have similar signatures.

Hence, the function `holdout_cross_validation()` is used for performing an hyperparameter tuning process on the `TreePredictor` class. The structure of possible configurations is fixed and designed to contain all construction parameters, including all stopping criteria. Of curse, since the grid space Θ is infinite, it is necessary to limit the search space: some of the values in a configuration are naturally limited, either because are fixed for the considered problem, as the decision metric, or because only a single value is available for the provided implementation, as the choices of continuous and categorical conditions for splitting, whereas the remaining parameters are not naturally limited because dependant on continuous values, so they have to be limited manually.
Table 4.2 sums up the individual ranges of values considered for the tuning process, that are then combined to create a limited sub-space Θ_0 of configurations for the search composed of 560 configuration to test.

And a further simplification of the process is in the number of considered sample, as said at the beginning of the section, that is limited to 10.000 to speed up the training of the predictors.

Name	Range start	Range stop	Range step
Continuous condition	`ThresholdCondition`	-	-
Categorical condition	`ThresholdCondition`	-	-
Decision metric	`information_gain`	-	-
Maximum depth	15	25	10
Maximum leaves	50	250	50
Maximum nodes	150	450	100
Minimum samples	0*	150	25
Impurity level	0.00	0.25	0.25

Table 4.2: Individual hyperparameters ranges used for defining possible configurations. * By default maximum purity stopping criteria is employed, so minimum samples range act like if it starts from 1.

After the exploration of this limited space, the resulting best hyperparameter set is the one composed of the possible values that allows to obtain an underlying tree that is almost equal to the one evaluated using the previous methods, however the estimated expected risk found by holdout cross validation is 0.028, higher then the base cross validation value, but as said before, the trees producing such results are not identical and the computation of the two methods is different.

To complete the tuning procedure, the best hyperprameter set found was used for initializing a `TreePredictor` instance and training it. The train/test errors and the confusion matrix relative to its evaluation are shown in figure 4.3. They show that the achieved accuracy is also in this case very high.

Of course the obtained results in 4.3 are not meant to be fully trusted, because they are computed on a strongly limited space of parameters and on a limited dataset of examples. If the same best hyperparamters were used for the training a model results may be slightly different, but since they are already very similar to the one presented in 4.1 and 4.2, one can expect that such difference is not much significative.

4.4 Overfit and underfit

Two important phenomena that should be considered when training and evaluating a model are **overfit** and **underfit**. These two problems influence the predictive performances of a model because it could not be able to explain the data and approximate the unknown distribution.
It is possible to observe the overfit problem when there are a small train error but a large test error, whereas the underfit one occurs when there is a large train error that in turn implies a large test error.

Looking at the results obtained in previous sections, both in the cases of the exhaustive models and evaluation models, there are no signals that may indicate the presence of overfit or underfit. The provided outputs in the notebook, as well as the plots derived from them, show that both train and test errors are really small, excluding so the presence of underfit, and although train error almost tends to zero, also the test error does so, therefore there is no overfit.

This implies that typical further processing, generally applied on overfitting tree predictors, as **pruning** branches, are not required in this case. This consideration is also supported by the fact that the statistical risk and the expected risk are also small and close to zero.

A better and more precise analysis regarding the two phenomena could be achieved considering the bias-variance decomposition of the statistical risk, $l_{\mathcal{D}}(h) = [l_{\mathcal{D}}(h) - l_{\mathcal{D}}(h^*)] + [l_{\mathcal{D}}(h^*) - l_{\mathcal{D}}(f^*)] + [l_{\mathcal{D}}(f^*)]$, but without knowing neither the optimal predictor h^* nor the Bayes optimal predictor f^*, it cannot be performed (also if any of h^* or f^* were already known, there would be no reason for training a different predictor).

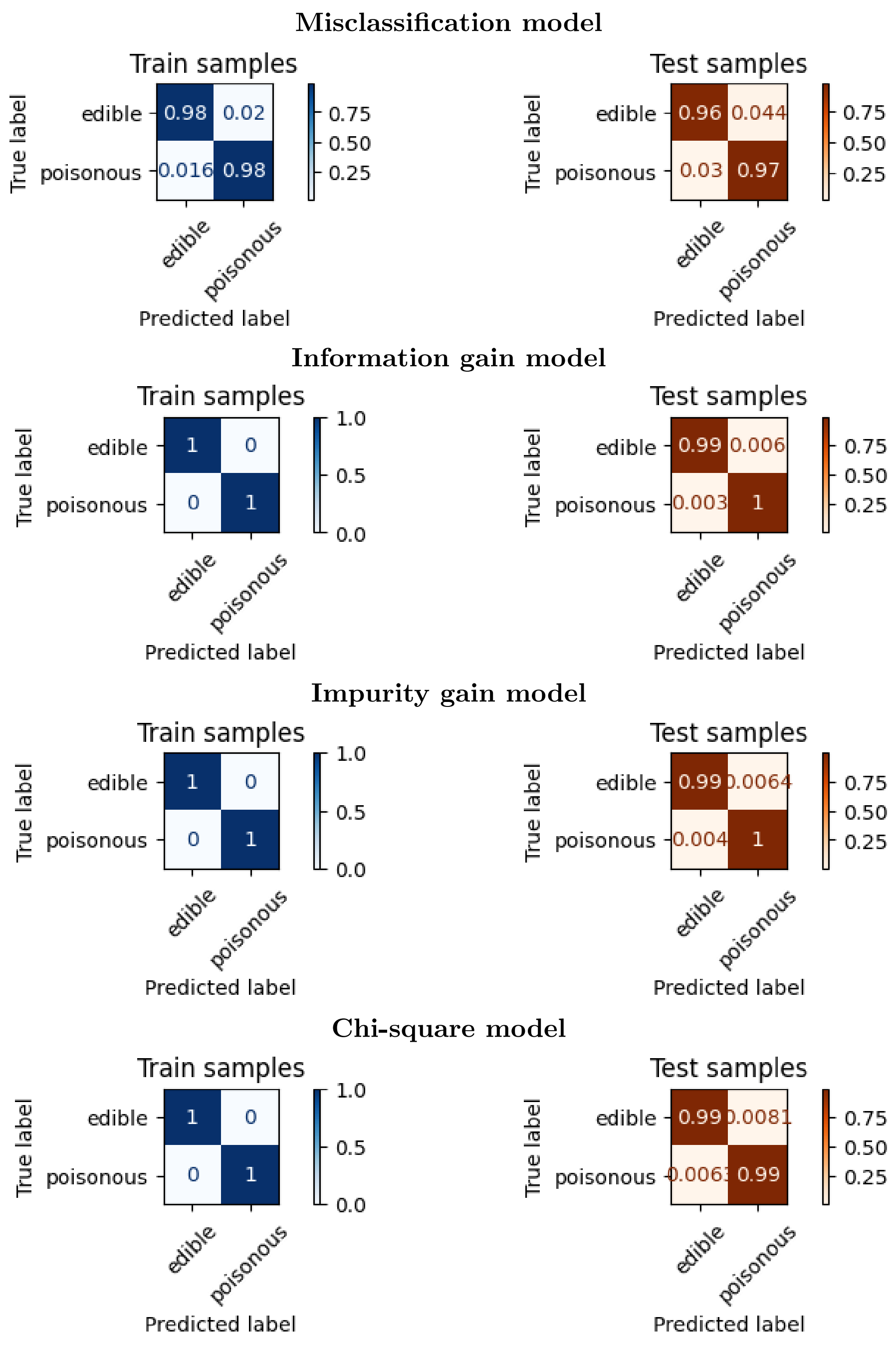
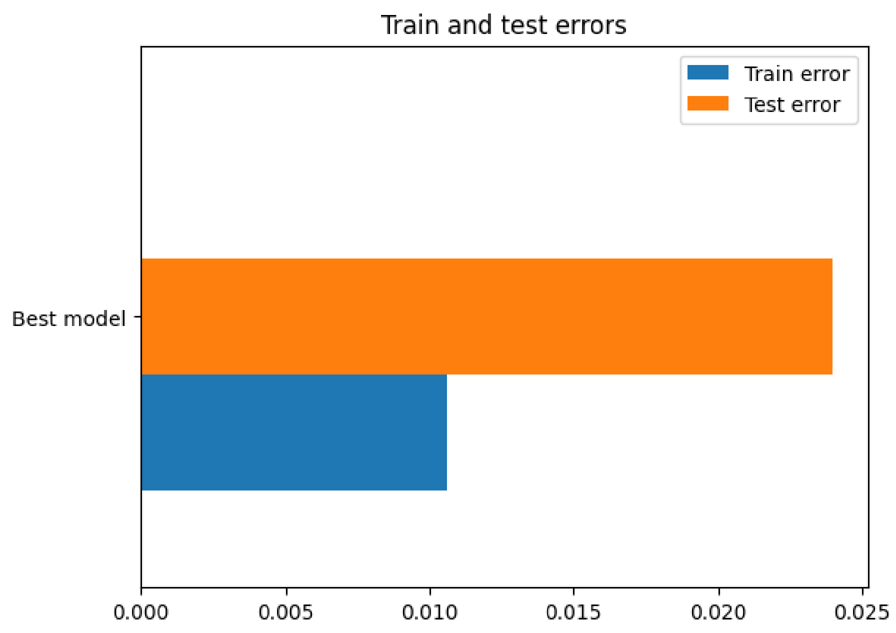


Figure 4.2: Confusion matrices (normalized by class) of the four exhaustively trained models.



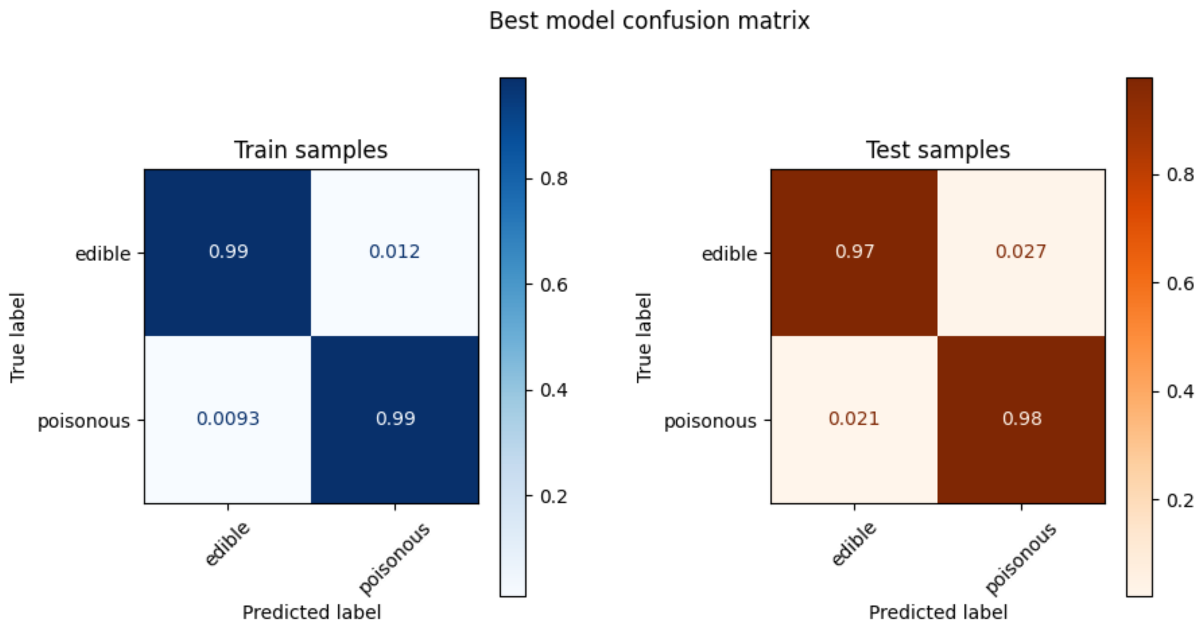


Figure 4.3: Train and test errors and confusion matrix of the best model trained using the hyperparameter configuration found by holdout cross validation.

5. Random forest

5.1 Random forest model

A **tree forest** (or decision forest) is an ensemble of general tree predictors that aim at solving the same task. The individual tree predictors in the forest can be obtained in any manner, however they are usually obtained using the same learning algorithm with different hyperparameters and they all contribute to perform predictions by combining their own predictions, generally in a manner depending on the task; in the case of classification tasks, combined predictions are commonly achieved by a simple **majority vote**. The advantage behind the ensemble methods is that they allow to control the error of prediction, achieving a better **bias-variance tred-off**.

However, basic ensemble methods work under strong assumptions regarding the independence of individual predictors and their errors, but there are different heuristics that allow to achieve the same advantages.
One of those is the **random forest**, a heuristic that extend the **bagging** one: given a dataset S of m examples, it is uniformly randomly sampled with replacement multiple times, to obtain some new datasets $S_1, \dots, S_T$ (bags), one for each base predictor in the ensemble. These sets are characterized by the fact of still having m examples, but due to the randomness of the sampling, it can be proven that around 2/3 of them are unique while the remaining 1/3 are duplicated.
Each bag is then used for training a predictor, but the addition provided by random forest is the introduction of noise in the examples by also sampling the available features at each step of the training.

The overall effect of the random forest is that bias and variance are reduced by the first sampling lowering the prediction error and the addition of noise in the second sampling reintroduces some variance in the model, allowing to prevent **overfit** problem and thus obtain a good statistical risk for the final ensemble.

The implementation of random forest requires a small modification to the previous definition of tree predictor that consequently influences its implementation.
The new modification consists in the adding of an additional mechanism regarding the selection of the features to rank when looking for the best decision of nodes during the training of the tree predictors.

To this end, inside the `commons.splitting_criteria` module of the project, a simple hierarchy of class is added, it is based on the abstract class `FeatureSelector`, that provide a common interface for concrete **feature selectors**. Specifically, two selectors named `AllFeaturesSelector` and `RandomFeaturesSelector` are implemented and, as it can easily be understood from their name, they allow to respectively select all features and a subset of randomly extracted features.

This new additions are added to the previous implementations in such a way that all scripts and notebook still work without any edit. Hence in this new optic, all previously described and used tree make use of `AllFeaturesSelector` selector, because their searches of the best features is not limited, whereas in the tree employed inside a random forest would use the `RandomFeaturesSelector` selector.

The random forest model is implemented within the `predictors.forests` module in a class named `TreePredictorRandomForest`. Since this forest is based on the previously described implementation of tree predictor, the parameters required for the construction of the forest are the same as the `TreePredictor` class, with some obvious additional parameters representing the **number of trees** and eventually a **random seed** required for initializing other random aspects of the ensemble.
The class follows the same simple interface, with `fit()` and `predict()` methods respectively used for training the ensemble and use it for prediction. These two actions are of course adapted to the new structure: the training procedure consist in training each individual tree predictor on a different bag of

examples and using a feature selector randomly initialized, while prediction similarly consists in applying majority voting to the collection of predictions obtained from individual trees.

The following snippet of code shows the training procedure applied in the `fit()` method of the class:

```
def fit(self, samples:np.ndarray, labels:np.ndarray, verbose:bool = False) -> None:
        ## ...

        ## New forest initialization
        self._tree_predictors = []
        if self._random_seed:
            np.random.seed(self._random_seed)

        ## Trees random seeds
        tree_bagging_seeds = np.random.randint(1, 2**16,
        ↪   self._tree_predictors_number)
        tree_selector_seeds = np.random.randint(1, 2**16,
        ↪   self._tree_predictors_number)

        ## Individual tree training
        for i in range(self._tree_predictors_number):
            ## Samples boosting aggregation
            samples_bag, labels_bag = self._samples_bagging(samples, labels,
            ↪   tree_bagging_seeds[i])

            ## Features selector
            features_selector =
            ↪   RandomFeaturesSelector(self._features_selection_number,
            ↪   tree_selector_seeds[i])

            ## Tree predictor initialization
            tree_predictor = TreePredictor(
                self._tree_continuous_condition,
                self._tree_categorical_condition,
                self._tree_decision_metric,
                self._tree_tree_stopping_criteria,
                self._tree_node_stopping_criteria,
                features_selector
            )

            ## Training tree predictor
            tree_predictor.fit(samples_bag, labels_bag, verbose)

            ## Adding tree to forest
            self._tree_predictors.append(tree_predictor)
```

5.2 Training and evaluation

The described random forest model is also tested on a binary classification task with the mushroom dataset, processed in the usual way. The code relative to this experiment is available in the notebook `forest.ipynb` and in the script `forest.py` under the main directory of the repository.

A random forest is initialized with a number of underlying tree equal to $T = 5$, while the rest of the hyperparameter configuration is exactly the same unbounded one used for the training of exhaustive models, therefore also the obtained forest is fully expanded, but the underlying tree predictor may differ due to the introduction of randomness in the training process.

The evaluation of a random forest would require a procedure named **out-of-bag** (OOB) evaluation, that evaluate the model according to the bagging procedure: using the training set, the label of each example should be compared only with the one predicted by a tree that has never seen such example. But such procedure is complex to implement, hence for simplicity, the train/test split method is still used as alternative. This method provide the examples in the test partition to every tree in the forest, but still guarantees that they have never been seen because the model is trained on the train partition. And furthermore this is the same method used before, so the results of individual tree and random forests can be compared.

Results of the evaluation trough the train/test errors and the relative confusion matrices are shown in figure 5.1. As it can be seen, both train and test errors are almost nonexistent and also accuracy (expressed in terms of TP and TN) practically equal to 1.
Although non-significant, the presence of train error is due to the randomness that is introduced in the selection of the features.

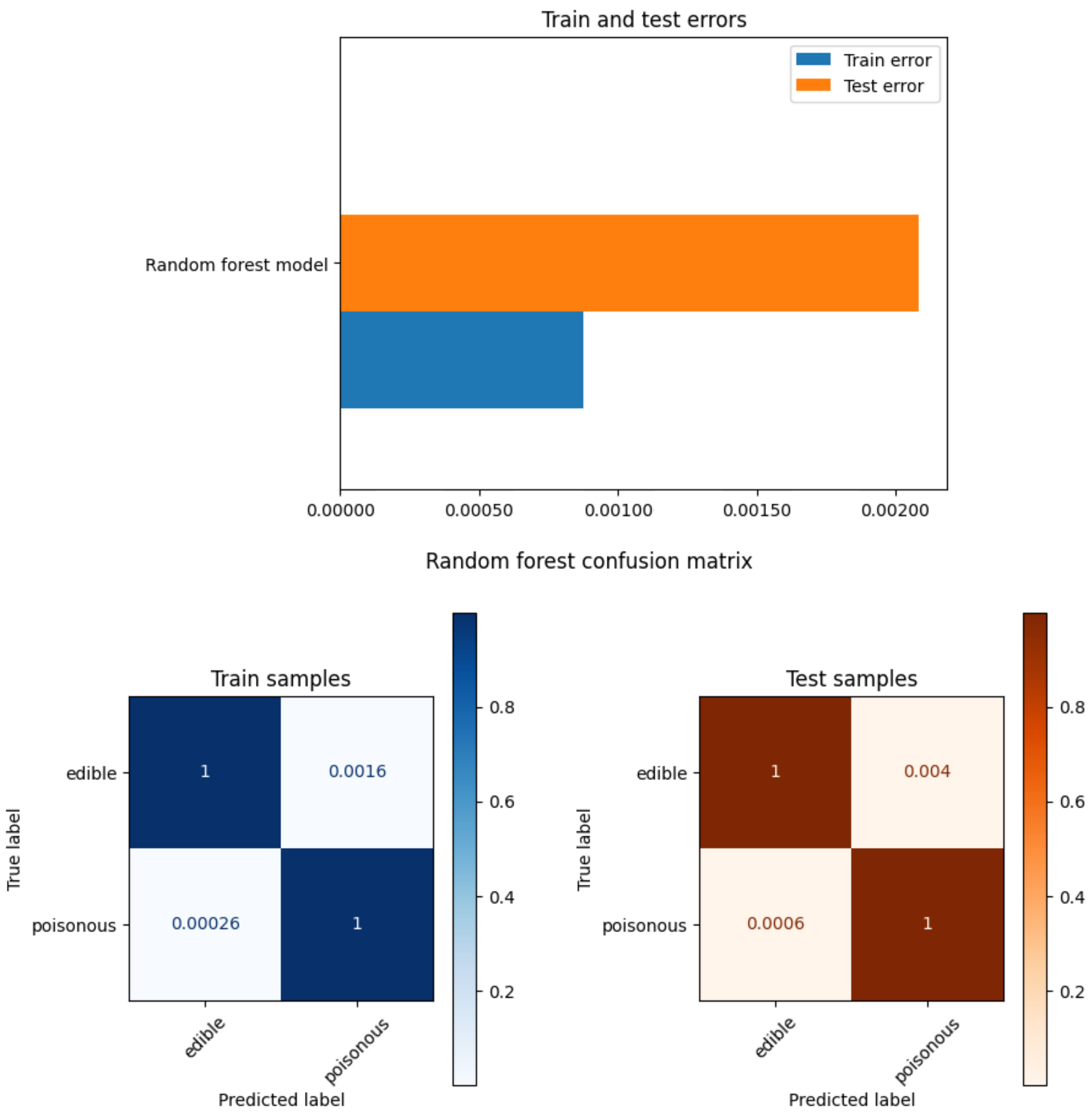


Figure 5.1: Train and test errors and confusion matrix of the random forest trained using the hyperparameter configuration of exhaustive predictors.

6. Conclusions

The realization of this project and its related report shown how it is possible to implement an algorithm for training tree predictors and use them to perform prediction over a given set of annotated data.

Specifically the proposed algorithm was employed in a binary classification task, a pre-processed dataset containing examples regarding the description and binary classification of mushrooms was considered for training predictors and testing their effectiveness. Experiments and evaluation procedures performed on this dataset shown that high accuracy, close to 98%, can be achieved both when building exhaustive models that rely on fully expanded trees and with smaller models trained on few data.
Simple observations on the results also shown that these models are not affected by the typical overfitting problem, commonly observed in decision trees.

With small additions to the main implementation, it was also possible to implement an algorithm for training random forests built on tree predictors.
As it could be expected, when considering the same dataset, random forest achieved an extremely high accuracy, almost equal to 100%, demonstrating that ensemble methods allow to achieve better performances than individual predictors and due to their construction, random forests prevent the overfitting problem.

Bibliography

[1] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1:81–106, 1986.

[2] J Ross Quinlan. *C4. 5: programs for machine learning.* Elsevier, 2014.

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

[4] Jacques Wainer and Gavin Cawley. Nested cross-validation when selecting classifiers is overzealous for most practical applications. *Expert Systems with Applications*, 182:115222, 05 2021.