Algorithms For Massive Dataset Exam Project
Tree Predictors for Binary Classification

Riccardo Raffini - 45999A

June 2024

I/We declare that this material, which I/We now submit for assessment, is entirely my/our 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/We 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/us or any other person for assessment on this or any other course of study.

Contents

1. **Introduction**

2. Dataset

2.1 Original dataset

The project described in this report is carried out considering the **Secondary Mushroom** dataset, whose files and description are 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.

The dataset is made up of two main *CSV* files, namely *primary* and *secondary*, but only the latter is analyzed; this file contains a total of 61069 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.

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.

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.

2.2 Data pre-processing

Further analysis of the dataset shows that the whole list 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 by a threshold value that implies the dropping of the feature if and only if its quantity is above this chosen value.
After this first action, 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 custom 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, 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 translates into the following functions named `delete_dataset_features()` and `fill_dataset_samples()` defined inside the module `datasets.methods`:

[code implementation]

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()` 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.

[code implementation]

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.

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.

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 build hyperplanes.
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 split 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 using non-binary conditions have 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 named *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 origining 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 partitions, when used on this purpose the decision should act as a map within a value and a node's child, so co-domain could be limited to a finite range $K = \{0, 1, \dots k-1\}$. They are implemented following an hierarchy of abstract and concrete classes, whose code is inside the `commons.splitting_criteria` module.
A generic condition is represented by an 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 to return 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`, `ThresholdCondition` 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 represented in figure [???].

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 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 have 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, according to which values were passed to the constructor, the difference between the two kind of nodes is achieved: 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.

Since the main purpose of a node is either returning the associated label or use it to split data into partitions, the decision criterion of an internal node can be applied to a 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.

[code implementation]

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 employed 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 samples collection.

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 metric for evaluating and scoring conditional decisions on the features that represent possible splitting point during the construction of the tree.
- A list of stopping conditions 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 conditions 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.

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 [???].

3.4.2 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 an error 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 was 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 locally best choices at each step of construction. The following snippets of code represent the training procedure used by the `TreePredictor` class.

[code implementation]

The training process starts by calling the fit() methods, 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 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. 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 $\{l_1, l_2, \dots\}$ 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 becomes a *candidate* node. However nodes can only be expanded one at time, hence at this end a queue of nodes is introduced; it holds not only the nodes themselves but also some other information, specifically the associated samples (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, there is no guarantee that the growing tree will be complete, in fact the first action performed on each candidate node 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 a node 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 samples $S_c = \{(x_j, y_j)\} \subseteq S$ associated to it and 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 θ for a condition C chosen according to the feature type, returning the feature i^* and the parameter θ^* for 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 of 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 a simple trick whose main cost is due to a sorting, their number can be reduced and the scoring metric applied to them requiring $O(m)$. A bound for the final cost of training is thus $O(m \cdot d(m \log m + m^2))$, successive operations have lower complexity.

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 node whose label is the majority class $\tilde{y}_i$ in that partition is linked to the parent, extend so the tree structure and it is also added to the leaves queue 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.

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 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 $\hat{y}_i \in Y$.
Once a new data point represented by a vector $x_i = (x_1, \ldots, x_d)$ 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, applying the conditional statement of each internal node visited on one of the features x_i, providing the considered data point x 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 $\hat{y}$ that is associated to this leaf node.

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 how it is possible to implement the `predict()` method with a $O(nm)$ time complexity, where m is the maximum depth of the underlying tree and n is the number of data points to which predict labels.

[code implementation]

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.

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 required predictor required by the asked task of classification can be achieved by forcing the following restrictions on the tree: 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 statement conditions have only binary outcomes and that the considered dataset has only to class labels, every trained tree predictor will be a binary tree predictor producing binary classifications.
A simple proof of this is given by the subdivision of 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 $L = (T + 1)/2$, typical property 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 decisions, that is an evaluation criterion that in turn is based on a so called **index** that assigns a numerical value to a partition. Usually the criterion consists in minimizing the values of any index when it is computed on the samples assigned to the resulting partition, but as done in the project, there is an equivalent computation consisting in maximizing a **gain** between parent and children nodes of a split.

In this project, the following indices are considered and the code relative to their implementation is in the module `commons.splitting_criteria`:

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

Here n represents the number of distinct class labels characterizing the problem and p_i the frequency of a class i inside the considered partition p.

Those indices can then be applied to implement some decision metrics for the training of tree predictors, as the `TreePredictor` class follows the equivalent definition of *gain*, its metrics have to be defined as difference between the index value computed on the original partition and the weighted values computed on partitions obtained as consequence of a split decision, 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**.

They are all available inside the module `commons.splitting_criteria` by homonyms functions, however the generic function named `measurement_gain()` used for implementing them also allows to easily define additional decision metrics by simply providing new indices.

4.2 Exhaustive training

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

More formally the common hyperparameters for the `TreePredictor` instances that were trained 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)

while `decision_metric` parameter was independently set for each tree to any of `misclassification_gain`, `information_gain` and `gini_impurity_gain`.

Although no control on the growth of the trees was applied, the results of prediction over the test set are definitely *good*, they seem to not show any presence of **overfitting** as shown by the simple graph and the confusion matrices in figures [???] and [???].

All the code relative to this experiment is available in the Jupyther notebook named `notebook_1.ipynb` in the main directory of the repository.

4.3 Hyperparameters tuning

Of course the previous tree work well, but their exhaustive training requires some time, ...

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.