Decision Trees and Hyper-parameter Tuning

Antonio De Patto - 46430A

Machine Learning Module - Data Science for Economics

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.

1 Introduction and dataset description

Figure 1: Missing values

Features	NA
cap-diameter	0
cap-shape	0
cap-surface	14120
cap-color	0
does-bruise-or-bleed	0
gill-attachment	9884
gill-spacing	25063
gill-color	0
stem-height	0
stem-width	0
stem-root	51538
stem-surface	38124
stem-color	0
veil-type	57892
veil-color	53656
has-ring	0
ring-type	2471
spore-print-color	54715
habitat	0
season	0
label	0

In this project we will study the behavior of a decision tree on a dataset related to the classification of poisonous and non-poisonous mushrooms. The tree was created from scratch in python and will be using scaled entropy as splitting criteria. Subsequently this tree will be compared with the trees obtained through other splitting criteria to see any differences. The performance of each decision trees will be defined through the Accuracy and the 0-1 loss, that will give a numeric measure of the prediction ability of each tree. The results of these performances will then be evaluated through K-fold cross validation. We will conclude our work with the hyper-parameter tuning procedure in order to improve the decision-making power of our model and avoid under-fitting or over-fitting problems.

The dataset The dataset contains 61.069 hypothetical mushrooms, each modeled after one of 173 species with 353 mushrooms per species, where each mushroom is classified as either definitely edible or definitely poisonous. As we can see from the table in Figure 1, many variables have a high number of missing values, as in the case of 'gill-spacing', 'stem-root', 'stem-surface', 'veil-type', 'veil-color' and 'spore-print-color'. We then proceed to remove these variables because they would produce distorted classifications due to the lack of information they would give. We then remove the missing values of the variables 'cap-surface', 'gill-attachment' and 'ring-type'. Since they present a smaller number of missing values than the previous variables, it is preferable to keep them in our study. We therefore have a reduced dataset of 37,065 observations but it is clean and ready to be used.

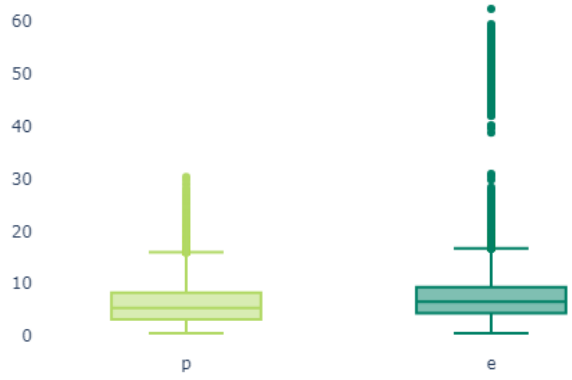
Response variable Let's now move on to the analysis of the response variable. This variable, which is called 'label' in our dataset is divided into two categories, 'e' that stands for edible mushroom and 'p' that stands for poisonous mushroom. Our analysis shows that both classes are fairly distributed and are both balanced around 50%, even if slightly more balanced towards the 'poisoned' category. More precisely, the number of observations indicated as 'poisoned' are 20.121, so 54,3% of the total, and those indicated as 'edible' are 16.944, so 45,7% of the total. The difference between the two classes is therefore not marked, otherwise we would have had a biased model that we could not generalize to

external data. We can therefore conclude that we do not have a distribution skewed towards a single category and we expect our model to be balanced.

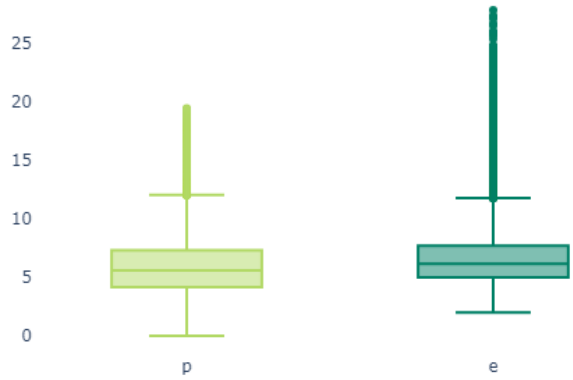
Explanatory variables The remaining 14 variables will be used by the decision tree to carry out the classification. It is important to underline that even if the variables have outliers and are slightly correlated with each other, the decision tree will not be influenced by these factors. This is one of the strengths of the trees, namely the robustness to outliers and the lack of explicit a priori assumptions. The descriptive variables are listed below with their relative subdivision into continuous and categorical ones.

1. cap-diameter (continuous): float number in cm
2. cap-shape (categorical): bell, conical, convex, flat, sunken, spherical, others
3. cap-surface (categorical): fibrous, grooves, scaly, smooth, shiny, leathery, silky, sticky, wrinkled, fleshy, dry
4. cap-color (categorical): brown, buff, gray, green, pink, purple, red, white, yellow, blue, orange, black
5. does-bruise-bleed (categorical): bruises-or-bleeding, no
6. gill-attachment (categorical): adnate, adnexed, decurrent, free, sinuate, pores, none
7. gill-color (categorical): see cap-color, none
8. stem-height (continuous): float number in cm
9. stem-width (continuous): float number in mm
10. stem-color (categorical): see cap-color, none
11. has-ring (categorical) ring, none
12. ring-type (categorical): cobwebby, evanescent, flaring, grooved, large, pendant, sheathing, zone, scaly, movable, none
13. habitat (categorical): grasses, leaves, meadows, paths, heaths, urban, waste, woods
14. season (categorical): spring, summer, autumn, winter

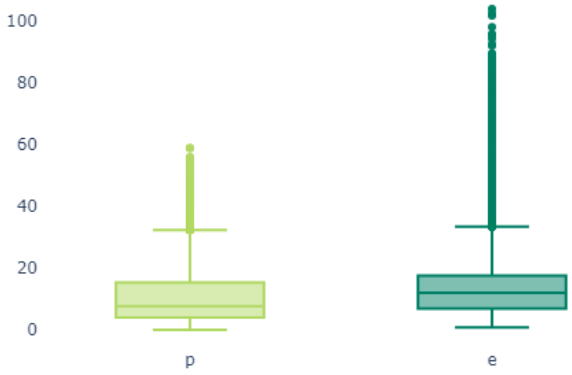
We now present a brief descriptive analysis of these variables just listed, using boxpots for continuous variables and barplots for categorical variables.



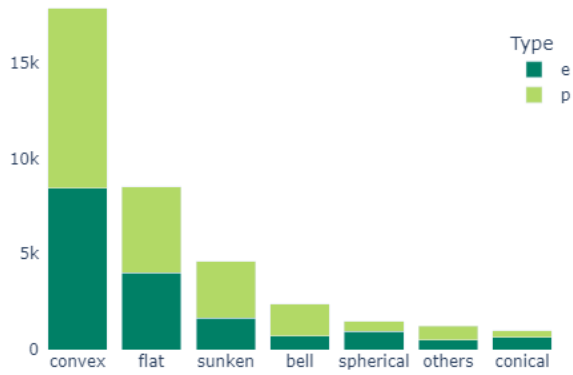
(a) Boxplot of Cap-diameter(cm)



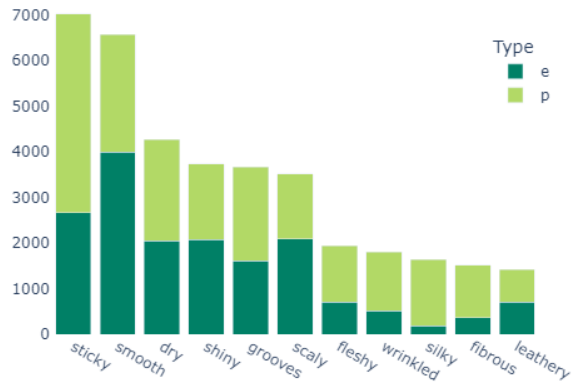
(b) Boxplot of Stem-height(cm)



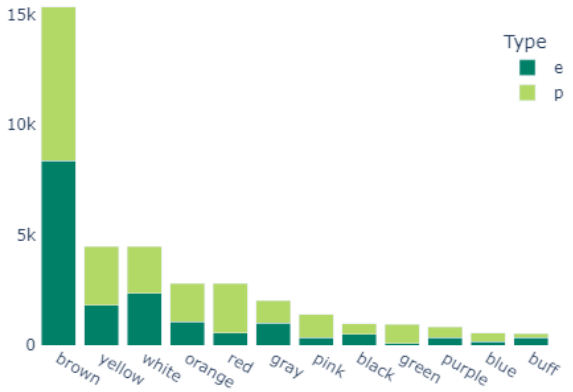
(c) Boxplot of Stem-width(cm)



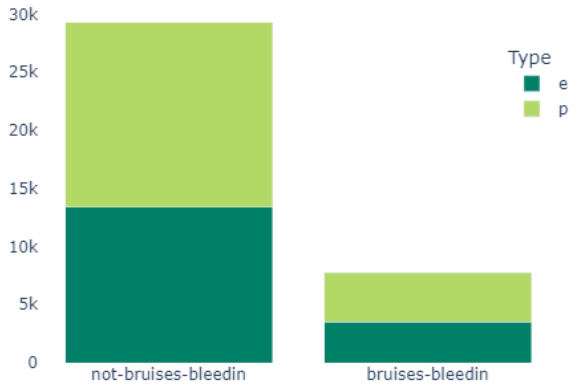
(d) Barplot of Cap-shape



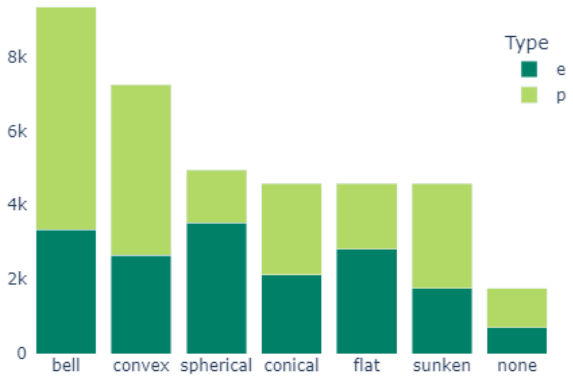
(e) Barplot of Cap-surface



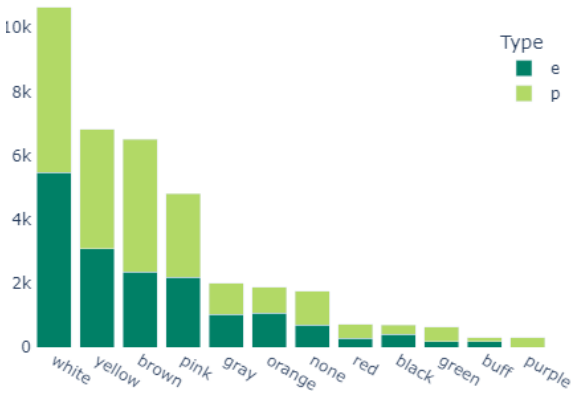
(f) Barplot of Cap-color



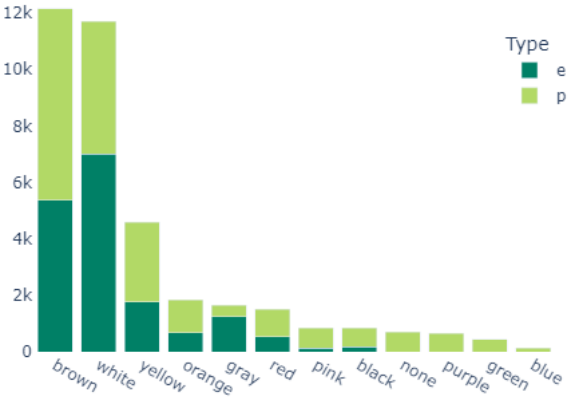
(g) Barplot of 'Does bruise or bleed'



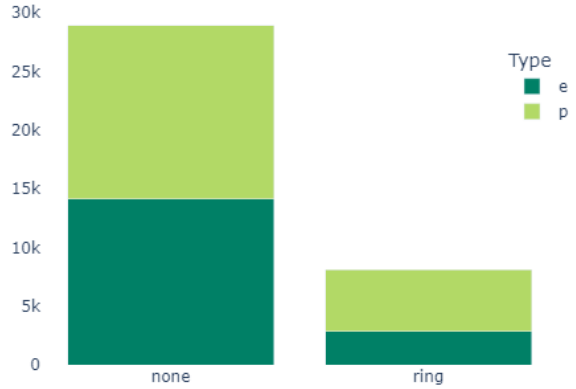
(h) Barplot of Gill attachment



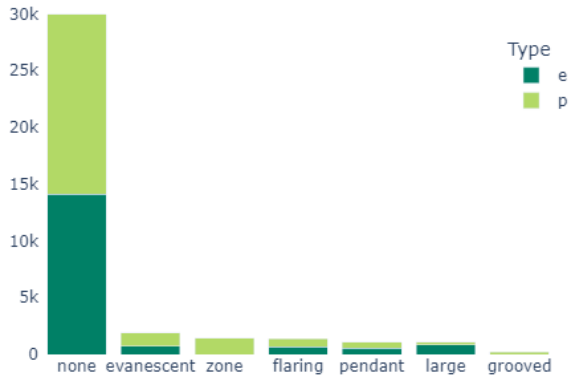
(i) Barplot of Gill color



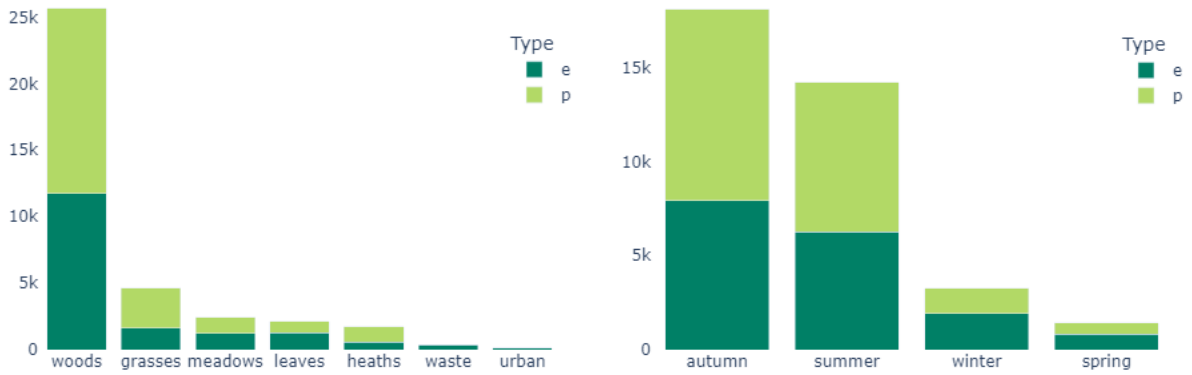
(l) Barplot of Stem color



(m) Barplot of 'Has ring'



(n) Barplot of Ring type



(o) Barplot of Habitat

(p) Barplot of Season

Figure 2: Boxplots and Barplots of the Explanatory variables

Looking at the box-plots in Figure 2, we can see the presence of outliers in both categories, but as previously stated, the decision trees are not affected by these problems, so we can continue. Regarding the bar-plots, that is, the graphs starting from (d), we can almost always see a division of edible and poisoned mushrooms for each category of each variable, to underline once again how the dataset is not biased and oriented towards a single response category.

2 The theory behind a decision tree classifier

Visually, a decision tree is structured as a set of interconnected elements known as nodes. At the top of the tree is the root node, which serves as the starting point from which all other nodes branch out. Each node that generates subsequent nodes is referred to as a parent node, while the nodes that descend from it are called child nodes. The thresholds that determine how the data is divided within each node are known as splits. At the end of each branch, the nodes that do not further split are called leaf nodes. The hierarchical segmentation obtained through a decision tree can be defined as a step-wise procedure, through which the set of n statistical units is progressively divided, according to an optimization/splitting criterion, into a series of disjoint subgroups which present within them a greater degree of homogeneity than the initial set. The algorithm uses a splitting criterion, such as Scaled Entropy or Gini impurity for classification tasks, or mean squared error reduction for regression tasks, to evaluate the effectiveness of each split. These splitting criteria for a decision tree are:

- Scaled entropy: $\psi(p) = -\frac{p}{2}\log_2(p) - \frac{1-p}{2}\log_2(1-p)$
- Gini impurity: $\psi(p) = 2p(1-p)$
- Bhattacharyya coefficient: $\psi(p) = \sqrt{p(1-p)}$

At each step of the process the heterogeneity in the groups is reduced compared to the previous step. At the end, the leaves of the tree, used to graphically describe the procedure, present such a degree of homogeneity that they can be attributed to one of the starting classes.

To address this challenge, decision tree algorithms employ a top-down, greedy approach known as recursive binary splitting. This approach starts at the root node, representing the entire feature space, and recursively splits the predictor space into two child nodes at each step. Each split divides the feature space into two subsets based on the value of a chosen predictor variable. At each step of the tree-building process, the algorithm selects the best split among all possible splits available at that particular step. The splitting process continues recursively until a stopping criterion is met, such as reaching a maximum tree depth, minimum number of samples in a node, or when further splits no longer provide significant improvement in homogeneity or purity. By employing recursive binary splitting, decision tree algorithms efficiently construct a hierarchical tree structure by making locally optimal decisions at each step. While this greedy approach may not always lead to the globally optimal tree, it often results in a satisfactory tree that effectively captures the underlying patterns in the data.

0-1 loss and Accuracy We conclude by citing the formulas relating to the evaluation of the model, which will be widely used in this study to evaluate its performance. First of all we have the 0-1 loss that is calculated according to the following formula:

$$l(y, \hat{y}) = \begin{cases} 0 & \text{if } y \neq \hat{y} \\ 1 & \text{if } y = \hat{y} \end{cases}$$

where l is the loss function, y is the true label and $\hat{y}$ is the predicted label.

Secondly we will use the accuracy, who measures the percentage of exact predictions out of the total instances and ranges from 0(worst) to 1(best):

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN}$$

where:

1. True Positives(TP): the number of examples that were predicted as positive(label 'p') and are actually positive(true label 'p')
2. True Negatives(TN): the number of examples that were predicted as negative(label 'e') and are actually negative(true label 'e')
3. False Positives(FP): the number of examples predicted as positive(label 'p') but actually negative(true label 'e')
4. False Negatives(FN): the number of examples predicted as negative(label 'e') but actually positive(true label 'p').

3 How does the algorithm works ?

Train and Test set We begin the coding of the decision tree by initializing a function to divide our dataset into train and test sets. This function will allow us to choose the size of our test set, and consequently that of the training set, assigning the observations randomly. This can be done through the 'test_size' parameter. If 'test_size' is provided as a float, it represents the proportion of the original dataset that should go into the test set. The function calculates the corresponding number of rows for the test set based on this proportion. If 'test_size' is an integer, it directly specifies the number of rows that should be in the test set. The function begins by creating a list of all row indices contained in the mushroom dataset. It then uses the 'random.sample' function to randomly select a subset of these indices for the test set while the training set is formed by removing these rows from the original dataset. In our case, 20% of the observations, or 7.413 units, will be included in the test set while the remaining 80%, or 29.652 observations will be included in the training set.

Type of feature We now describe a function that allows us to distinguish between categorical variables and continuous variables. In fact, these will be subject to different classification methods as will be explained later. In short, this function allows us to classify variables based on two requirements. It initializes an empty list called 'feature_types' to store the type of each column and sets a threshold of 15 unique values to help in this classification. For each column in the dataset, excluding our response variable column, the function checks the unique values. If the first value is a string or if the column has 15 or fewer unique values, the function considers the column 'categorical'. Otherwise, it classifies the column as 'continuous'. Finally, these classifications will be appended to the 'feature_types' list which will be given as output.

Node purity We proceed by describing a function that will be subsequently called by the main algorithm during the tree encoding phase. This function called in our code 'evaluate_purity' checks if a given partition of data contains only one unique class, which would indicate that the partition is 'pure'. The function works by extracting the last column of the data, which contains the labels, and then determining how many unique classes are present using Numpy's unique function. If there is only one unique class in a specific subset of data, the function returns 'True', indicating that the data is pure. Otherwise, it returns 'False', meaning the partition contains multiple classes and is not pure. In our case, we will classify as pure a node that only has mushrooms classified as edible or that only has mushrooms classified as poisonous. This purity check is essential for our decision trees, where pure nodes can be directly classified without further splitting.

Classification Let's now define the function related to the classification of mushrooms as poisonous or not. This classification will be performed based on the class most present in a given node, that is, whether 'p' or 'e'. It works by first extracting the labels from the last column of the data. It then uses Numpy's unique function to identify all unique classes and count how many times each class appears. The function finds the index of the class that appears most frequently using the 'argmax' method on the counts. The class corresponding to this index is considered the majority class, and the function returns this class as the classification result. For example, considering how the algorithm would classify the data of the training set that have a mushroom cap diameter of less than 10 cm, we would have that the class that appears most frequently is 'p', therefore the majority of these mushrooms would be classified as poisonous.

Potential split The 'find_potential_splits' function is used to determine possible split points for each feature in a dataset. It starts by creating an empty dictionary to hold potential splits for each feature and then retrieves the number of columns in the data, ignoring the response variable. For each of the remaining columns, it extracts the unique values and stores these values in the dictionary as potential split points for that feature. At the end the dictionary is returned and each key is a column index and each value is a list of potential split values for that column. For example, considering a categorical variable, the values will be the classes, taken individually, that make up this variable. Relative to continuous variables instead the number of values will be equal to the number of unique values that appear in this variable.

Splitting data Next we find the function that actually takes care of splitting the data into two different nodes. Potentially the data will be divided into 'data below' and 'data above' based on a certain split value and would respectively represent the left child node and the right child node. This function takes three inputs: the training set, the index of the column to split on and the value to use for the split and returns two parts of the dataset: one for the data that meets the split criteria and one for the data that does not. However, it is necessary to remember that the split is performed differently depending on the type of variable that is presented to the algorithm.

For example, in a continuous variable we split the data based on a numerical threshold value and the data is divided depending on whether it is below or above this threshold value. If the feature is categorical, the function splits the dataset into subsets based on whether the values are equal to or not equal to the split value. For example, if the algorithm decides to split the data based on the variable 'cap-color' and identifies the color red as the split value, then all mushrooms that have a red cap will be distinguished from mushrooms that do not have this color into two different set. Remember that the split is performed based on the chosen splitting criteria, which we will explain in detail shortly.

Splitting criteria We now introduce the first splitting criteria, namely Scaled Entropy, whose formula is:

$$\psi(p) = -\tfrac{p}{2}\log_2(p) - \tfrac{1-p}{2}\log_2(1-p)$$

In this case the probability 'p' for each class is calculated starting from the number of units classified as 'e' and classified as 'p' and divided by the total number of units present in the variable 'label'. Initially, we will obtain as probabilities 0.457 and 0.543, or the initial subdivision presented in paragraph 2.2. The entropy will therefore be the sum of these probabilities according to the formula above. In the revised function, a small constant (1e-10) is added to the probabilities before taking the logarithm to prevent mathematical errors that occur when dealing with zero probabilities. Calculating the entropy of the initial digits we obtain a value of 0.99, therefore close to 1 to indicate how the data of the training set are very confusing and random. Indeed this formula gives a measure of the unpredictability of the label distribution, with higher entropy indicating more unpredictability. Similarly, the same rules that apply to Scaled Entropy also apply to the Gini impurity and the Bhattacharyya coefficient split method.

Remember that through this function our algorithm will be able to determine which are the best splits to make. More precisely, we will be able to check all the potential splits calculated previously and determine the split that involves the lowest overall entropy. For each split, the data is divided into two groups, that are 'data_below' and 'data_above' as we have seen before. We then calculate the entropy for each of these groups and compute the weighted average entropy of the split. The goal is to

find the split that results in the lowest overall entropy, as this indicates a better separation of the data according to the target labels. The best split is the one that minimizes the total entropy across the resulting groups. The resulting value is the overall entropy of the split, which represents the average level of uncertainty across the two subsets. This value helps in deciding which split is the best by comparing it with other potential splits.

Best splits Let's now move on to the function that identifies the most effective way to divide a dataset depending on the chosen splitting criteria. The goal of this function is to set an arbitrarily high value of the entropy (or otherwise of the Gini impurity or of the Bhattacharyya coefficient) and determine which split present in the dictionary containing the potential splits involves a lower possible entropy. For each potential split, the function recall the split function and divides the dataset into two parts based on the current column and split value. Then calculates the overall entropy of this split using the 'calculate_overall_entropy' function. If this entropy is lower than or equal to the previously recorded entropy, the function updates its record of the best split to include the current column and value. After evaluating all potential splits, the function returns the column and value that resulted in the lowest overall entropy. In this way the tree will be able to create its own nodes and leaves and give a final classification to the observations. Remember that this function will perform a loop both on the variables of our dataset and on the potential splits so as to be able to return these two values to the main algorithm. It is therefore a recursive function as it will have to analyze all the columns of our dataset and all the potential splits obtained previously in order to obtain the best result.

Decision tree algorithm We now come to the core of our code, that is, the function that will actually take care of giving the actual tree as output. This function takes as parameters our dataset, a counter set to zero that will allow the algorithm to perform its loop, a parameter that allows us to choose the split method between the ones mentioned before and two parameters related to the stopping criteria that do not allow the tree to grow unbounded. The first of these two parameters is the minimum number of units that each node must have before it can be classified. When set to zero, the tree continues to enlarge nodes till all leaves are pure or until each leaf consists of fewer samples than the specified 'min_samples' parameter. The second parameter is related to the maximum depth reached by our tree in the growth phase. It is important to remember that these two parameters will be used again later to carry out the hyper-parameter tuning methodology, an important phase in which the algorithm will be able to determine which are the best values for these two parameters according to 0-1 loss and accuracy.

Going more deeply into the code, the 'decision_tree' algorithm constructs a decision tree by recursively splitting a dataset based on certain criteria. When the function is called for the first time, it initializes some global variables, such as 'COLUMN_HEADERS' and 'FEATURE_TYPES', which contain the names of the columns and the types of features in the dataset. It also converts the dataset from a dataset to a 2D NumPy array for easier manipulation. If the function is called recursively, it continues to work with the data in array format. The function checks several base cases that determine when to stop splitting the data. If the data is already "pure" (meaning all samples belong to the same class), if the number of samples in a node is less than a specified minimum ('min_samples' threshold), or if the tree has reached a maximum depth ('max_depth'), the function classifies the data at that node. So if in a particular node the number of samples becomes less than the minimum samples then we will not split that node any further and it will be a leaf node. Likewise, if the depth of the tree reach the maximum depth we will not split the nodes further. The classification is made using the 'determine_majority_class' function, which assigns the most common label in that node.

If none of the stopping conditions are met, the function continues to split the data. It first increments the counter, which tracks the current depth of the tree. It then identifies all possible splits using the 'find_potential_splits' function and determines the best split by minimizing entropy with the 'determine_best_split' function. The dataset is divided into two subsets: 'data_below' and 'data_above', based on the selected split column and value. Before proceeding with further splits, the function checks if either subset is empty. If so, it classifies the data at that node since no meaningful split can be made. Otherwise, the function prepares a question that represents the split, using the feature name and the split value. This question is the basis for creating a sub-tree. The function then recursively applies itself to both subsets ('data_below' and 'data_above'), which represent the 'yes' and 'no' branches of the current node. If both branches of the sub-tree lead to the same classification, the function sim-

plifies the tree by collapsing it into a single classification. If the branches differ, the function appends the results to the corresponding branches of the sub-tree. The completed sub-tree is then returned, representing one part of the larger decision tree.

Examples classification We conclude our explanations with the last functions related to the performance of the decision tree. Let's define the function that allows us to compare the values predicted by our model with the real ones of our dataset. This function is called 'predict_example'. It begins by extracting the first question from the tree, which includes the feature name, comparison operator, and value. If the feature is continuous, it checks if the feature's value in the example is less than or equal to the specified value. If true, it follows the 'yes' branch; otherwise, it follows the 'no' branch. For categorical features, it checks if the feature's value matches the specified value and follows the appropriate branch based on the result. If the branch leads to a final classification, the function returns this classification. If the branch leads to another sub-tree (a dictionary), the function recursively applies itself to this sub-tree until a final classification is reached. The process continues until the data point is fully classified.

0-1 loss and Accuracy Let's move on to the function that will allow us to calculate the accuracy of our tree. The 'compute_accuracy' function evaluates the accuracy of a decision tree by comparing its predictions to the actual labels in a dataset. It first adds a new column to the dataset called 'classification', which is generated by applying the 'predict_example' function to each row, using the decision tree to predict the label. Then, it creates another column called 'classification_correct' that checks whether the predicted label matches the actual label. The function calculates accuracy as the mean of the 'classification_correct' column, which represents the proportion of correct predictions, and returns this value as the accuracy of the model.

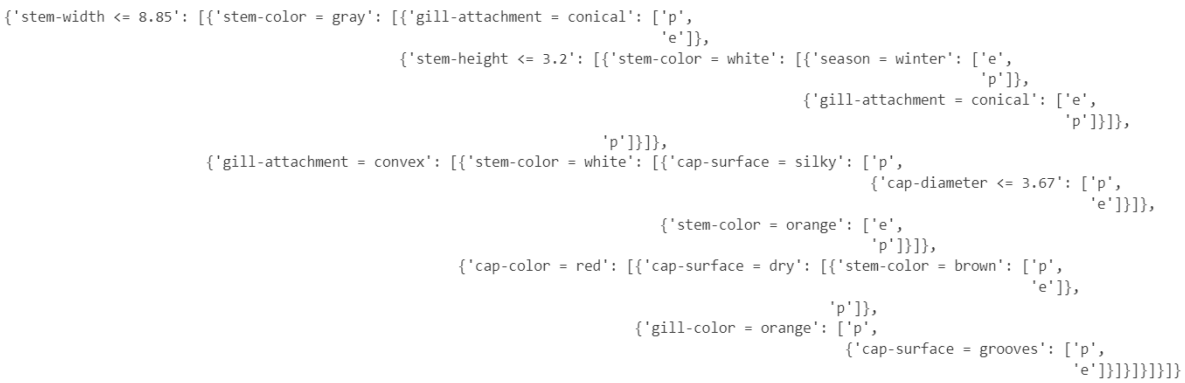
Similar to the operation of accuracy we find the function that defines the 0-1 loss. The 'zero_one_loss' function measures the proportion of incorrect predictions in a dataset. It starts by converting the true labels ('y_true') and predicted labels ('y_pred') into NumPy arrays. The function then counts the number of mismatches between these arrays, which represent the incorrect predictions. Finally, it calculates the loss by dividing the number of incorrect predictions by the total number of labels, returning this value as the loss. This loss value indicates the fraction of predictions that are incorrect, with a higher value representing worse performance.

4 Description and evaluation of the model

Scaled Entropy model The decision tree in Figure 3 was obtained starting from a training set that includes 80% of the initial observations while the remaining 20% are in the test set. The tree has a maximum depth of 5 and a minimum number of observations that each node must have in order to be classified equal to 200. At first the tree seems to present 3 main branches. The first one branches out from the first choice regarding the stem width while the division between the second and third branch is made when evaluating the gill attachment following the negative answer to the first decision, that is considering mushrooms that present a stem width greater than 8.85 centimeters.

The tree begins by evaluating whether the stem width is less than or equal to 8.85 centimeters. If this condition is true, the tree then considers the stem color. For mushrooms with a gray stem, the tree further checks if the gill attachment is conical, leading to a final decision of poisonous mushroom is the condition is verified. Alternatively, if the stem color is not gray we consider the stem height and we evaluate if is less than or equal to 3.2 centimeters. If so, it checks whether the stem color is white. If the stem is white and the season in which the mushroom grows is winter then the mushroom is classified as edible. If instead the stem is not white but the gill attachment is conical the mushroom is classified as edible in the same way.

We can note that mushrooms with a stem width less than or equal to 8.85 centimeters, whose stem color is not gray and that do not have a stem height less than or equal to 3.2 centimeters are classified as poisonous. Going back to the beginning of our tree we can consider mushrooms that have a stem width greater than 8.85 centimeters. In this case the variable related to the gill attachment is considered. If this is convex and the stem color is white and the cap surface is silky then we find ourselves with a mushroom classified as poisonous. If instead the cap surface is not silky but the cap diameter is less than or equal to 3.67 centimeters then the mushroom can still be considered poisonous.



Fihure 3: Tree obtained with Scaled Entropy, max_depth = 5, min_sample = 200

Going back briefly but always remaining on this second branch we can affirm that if the color of the stem was not white but orange we would have a solution that leads to classifying the mushrooms as edible. We finish by describing the third and final branch of our tree, which begins when the gill attachment is not convex and therefore the cap color is considered. If the cap is red and the cap surface is dry and the stem color is brown then we will classify the mushroom as poisonous. We will also classify the mushroom as poisonous if the cap surface is not dry. Going back briefly along the tree we note that if the cap color is not red and the gill color is orange then the mushroom is classified as poisonous. Similarly, if the gill color is not orange but the cap surface is grooves then the mushroom is still classified as poisonous.

What happens if we instead increase the minimum number of examples that each node must have in order to be classified? For example, by increasing the parameter from 200 to 500? In this case, we prune the tree since we only consider the most supporting branches.

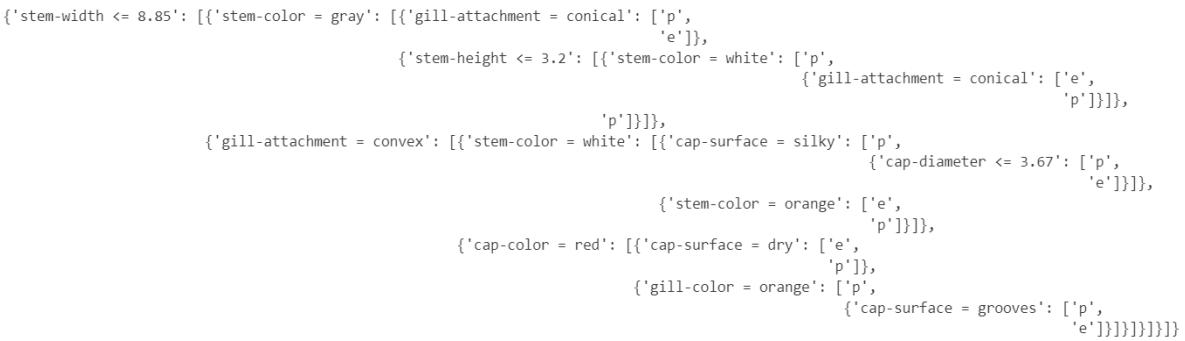


Figure 4: Tree obtained with Scaled Entropy, max_depth = 5, min_sample = 500

For example, in the first branch, the part relating to the season in which the mushroom develops is eliminated. In fact, looking at Figure 3, we can see that if the mushroom had a stem width less than 8.85 centimeters, the stem color was not gray, the stem height was less than 3.2 centimeters, the stem color was white and the season in which the mushroom appears was winter, then it would have been classified as edible. While if this last part relating to the season is removed, as in the case of the tree with a minimum of 500 observations per node, the mushroom is classified as poisoned as can be seen in Figure 4. Therefore, there is less precision in the classifications but greater ease of reading the model. We will now evaluate in more depth the difference between these two trees using evaluation methods such as accuracy and 0-1 loss.

Model evaluation Now that we have finished the description of our tree we can proceed with the evaluation of the model, studying the accuracy, 0-1 loss and confusion matrix. This part will be of fundamental importance to understand the performance of the model and if there are problems of over-fitting or under-fitting. In these cases it will be necessary to proceed with pruning procedures. Let's consider the tree obtained using Scaled Entropy and which has a maximum depth of 5 and a

minimum number of examples for the classification of the node equal to 500. First of all, let's start by presenting two classification cases performed by the model just to get an idea of what the tree should do for every row of the dataset. This classification reported in Figure 5 show the difference between a correctly classified example and an incorrectly classified example.

cap-diameter	8.05
cap-shape	convex
cap-surface	fibrous
cap-color	brown
does-bruise-or-bleed	not-bruises-bleedin
gill-attachment	bell
gill-color	brown
stem-height	9.44
stem-width	14.27
stem-color	brown
has-ring	none
ring-type	none
habitat	woods
season	autumn
label	p
classification	e
classification_correct	False

(a) Example incorrectly classified

cap-diameter	7.88
cap-shape	convex
cap-surface	dry
cap-color	red
does-bruise-or-bleed	bruises-bleedin
gill-attachment	spherical
gill-color	green
stem-height	6.37
stem-width	13.21
stem-color	yellow
has-ring	none
ring-type	none
habitat	woods
season	summer
label	e
classification	e
classification_correct	True

(b) Example classified correctly

Figure 5: Classification of examples

Let's now move on to the study of the accuracy of the model, which corresponds to 0.762 for the test set, meaning 76.2% of the predictions were correct. In this case the Accuracy will be the average of the correctly classified samples, as in case (b) in Figure 5. Similarly, looking at the 0-1 loss, 23.8% of the predictions were incorrect. Instead, for the training set, we have an accuracy of 77.1% and an error of 22.9%, therefore results very similar to those obtained in the test set.

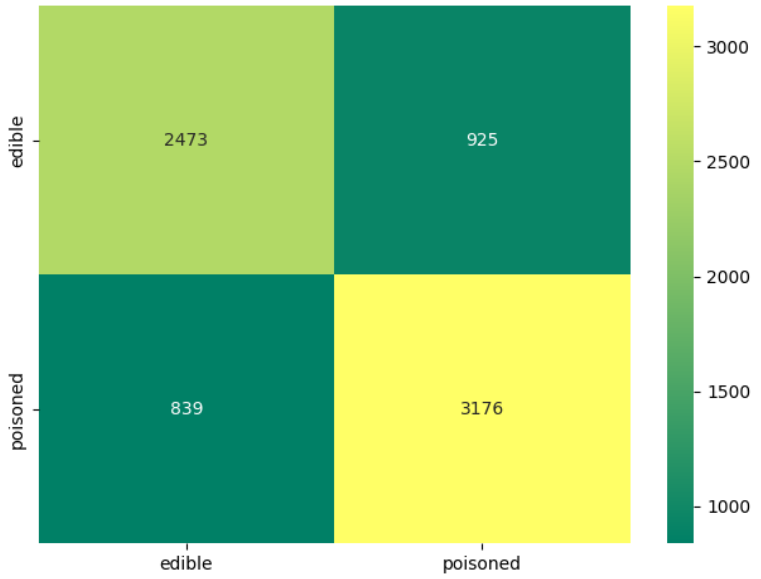


Figure 6: Confusion Matrix

To get a clearer idea of the model's classifications we can look at the Confusion Matrix shown in Figure 6. For example, 2473 observations (33.4% of the total observations) were correctly classified as edible and 3176 (42.8% of the total observations) were correctly classified as poisonous for a total of 76.2% correct classifications. This percentage corresponds exactly to the accuracy of the test set.

Therefore we can say that the model mostly makes correct predictions and does not seem to suffer from over-fitting or under-fitting problems. Mainly an under-fitting problem occurs when the test

error and the training error are similar to each other and are both large. In this case the errors are very similar, therefore we can safely say that the model does not suffer from this problem. Over-fitting instead occurs when the training error is low, but the test error is high. As before we can say that this problem does not affect our decision tree as both errors are low and pretty the same.

5 Hyper-parameter tuning

We now proceed with a fundamental part of our study, that is the hyper-parameter tuning related to the stopping criteria used by the decision tree, that is the minimum number of examples that prevent the tree from growing without boundaries and the maximum depth reached by the tree. We recall that in the previous paragraph, in Figure 4 the tree was built considering a minimum of 500 examples before stopping the growth and a maximum depth of 5. In this chapter instead these two parameters will vary in order to evaluate which combination returns a better result. More precisely, the tree will be evaluated by studying the accuracy of the training and test set and the minimum number of examples will vary from 500 to 2500 by 500 while the depth will vary from a minimum value of 3 to a maximum of 6.

	max_depth	min_samples	accuracy_train	accuracy_test
15	6	500	0.776845	0.764738
10	5	500	0.771213	0.762040
16	6	1000	0.772191	0.759477
11	5	1000	0.766559	0.756779
5	4	500	0.758937	0.748550
17	6	1500	0.756981	0.745312
18	6	2000	0.756644	0.744638
19	6	2500	0.756644	0.744638
6	4	1000	0.754620	0.743559
12	5	1500	0.751349	0.742614
13	5	2000	0.751045	0.742345
14	5	2500	0.751045	0.742345
7	4	1500	0.739410	0.729394
8	4	2000	0.739174	0.728990
9	4	2500	0.739174	0.728990
0	3	500	0.731283	0.719142
1	3	1000	0.726966	0.714151
4	3	2500	0.726966	0.714151
3	3	2000	0.726966	0.714151
2	3	1500	0.726966	0.714151

Figure 7: Train and test accuracy when considering different depth and minimum samples

As we can see in Figure 7, the results are sorted in decreasing order based on the accuracy of the model on the training and test sets. As could be imagined, at a greater depth and with a lower threshold of examples we have much higher accuracy values. This is because the leaves will be purer compared to shallower trees and with fewer minimum observations to perform a classification. Obviously, the more the depth of the tree increases, the more the explanatory capabilities of the model increase, but if the accuracy is too high this does not mean we have a perfect model. In fact, we could suffer from over-fitting, whereby the model is too dependent on the training model and has a low predictive power. In this case, however, the test set also presents values very similar to the training set in terms of accuracy

and precision of the classifications, therefore we can affirm that the model does not suffer from these problems. However, it must be considered that a model with high depth is also more complex to read and interpret, therefore since the accuracy is already high with a maximum depth of 5 and 500 as threshold for the minimum number of samples we can accept the model obtained previously.

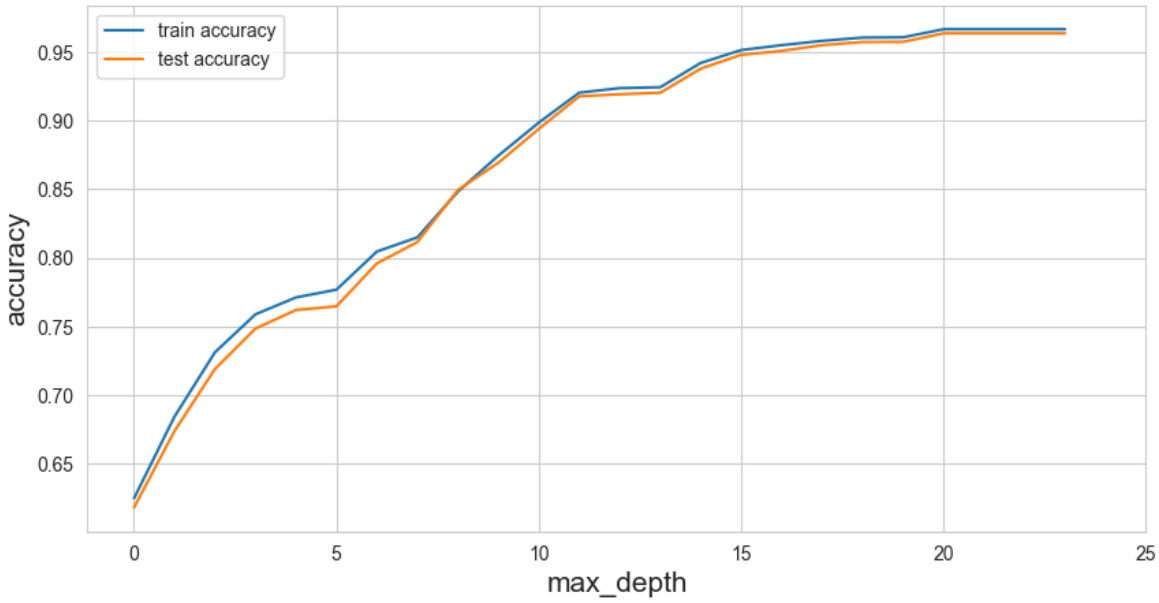


Figure 8: Relation between accuracy and max depth reach by the tree with 'min_sample' = 500

Let's now look at Figure 8, where the accuracy of the train and test sets are compared with the maximum depth reached by the tree. In this case the parameter relating to the minimum number of examples to classify a node remained constant at 500 and only the parameter relating to the depth of the tree changed. From the graph we can see that the two accuracies are almost identical and show the same trend. We can also see that by increasing the depth of the tree the accuracy of the two datasets increases significantly. This is because the tree is able to capture more complex patterns among the data up to creating a leaf per observation in the case in which no limits are imposed on the reachable depth. However, we would expect a situation in which the test error would increase due to the excessive dependence of the tree on the training set, but not in this case. This indicates that the algorithm does not over-fit and there is no dependence on the training set as the accuracy of the classifications performed on the training and test sets is almost the same at each depth.

6 K-fold cross validation

Previously we divided our dataset into two parts, namely the training set and the test set, obtaining a fairly significant model with a good percentage of accuracy. However, when employing the validation approach for model evaluation, it's crucial to recognize that the estimation of test error can exhibit high variability. This variability arises from the specific composition of the training set, meaning that slight variations in which observations are included in each set can lead to different performance estimates. This is particularly significant because, in the validation approach, only a subset of the data, namely those in the training set, is used to fit the model. Consequently, the model may not fully capture the complexities present in the entire dataset. As a result, the test error, which is based on this limited subset, may tend to overestimate the true test error for the model fitted on the entire dataset. This phenomenon occurs because the model might generalize better or worse when exposed to unseen data that differs from the training set. Therefore, it's essential to interpret validation set error estimates with caution, understanding that they may not accurately reflect the model's performance on completely unseen data. To mitigate this issue, techniques like cross-validation, which repeatedly partition the data into training and validation sets, can provide more robust estimates of model performance by averaging out the variability inherent in single validation set approaches. A widely used approach for

estimating test error is k-fold cross-validation. In our case we will split our dataset into 5 folds, of which 4 will be used as training in a recursive way so that training and test sets change each time. The concept behind k-fold cross-validation is to randomly divide the dataset into K equal-sized parts or folds. Then, iteratively, one of the K parts is left out as the validation set while the model is trained on the remaining K-1 parts combined. Predictions are then made for the left-out k-th part. This process repeats for each part k = 1, 2, ..., K, ensuring that each part serves as the validation set exactly once. Finally, the results from each fold are combined by averaging, to obtain an overall estimate of model performance. This approach provides a robust estimate of the model's generalization ability by leveraging multiple train-test splits of the data, thus offering a more reliable assessment compared to a single train-test split. By utilizing k-fold cross-validation, practitioners can make informed decisions about model selection and have a better understanding of the expected performance of the chosen model on unseen data.

```
Fold:1, Train set: 29652, Test set:7413
[   0    1    2 ... 7410 7411 7412]
Fold:2, Train set: 29652, Test set:7413
[ 7413  7414  7415 ... 14823 14824 14825]
Fold:3, Train set: 29652, Test set:7413
[14826 14827 14828 ... 22236 22237 22238]
Fold:4, Train set: 29652, Test set:7413
[22239 22240 22241 ... 29649 29650 29651]
Fold:5, Train set: 29652, Test set:7413
[29652 29653 29654 ... 37062 37063 37064]
```

Figure 9: Cross validation with 5 folds

As previously mentioned, the dataset has been divided into 5 folds after being randomized, and each fold is composed of 29.652 observations for the training set and 7.413 observations for the test set, as can be seen in Figure 9. Cross validation will also be performed by considering the same tree used in Figure 4, that is, a model with a maximum depth of 5 and a minimum number of examples to be able to classify a node equal to 500.

```
K-Fold Cross-Validation Scores: [0.7624443545123432, 0.7679751787400513, 0.7722919196007014, 0.7736409011196547, 0.7678402805881559
Mean Accuracy: 0.7688385269121814
```

Figure 10: Cross validation output

Looking at the Accuracies obtained in Figure 10 we can notice an average accuracy similar to the one obtained in Chapter 4. Therefore, even changing training and test sets at each iteration of the cross validation the model obtains excellent results.

7 Comparison between trees

Gini Impurity model Now that we have identified the optimal parameters using Scaled Entropy as the split method, we can move forward to compare this decision tree with those obtained through two other methodologies. First, we will construct a tree using the Gini impurity, which is presented below:

$$\text{Gini impurity: } \psi(p) = 2p(1-p)$$

Let's now visualize the tree obtained in Figure 11, which at first seems very similar to the tree obtained previously using Scaled Entropy. The tree changes only along the first branch, which is the one we find considering mushrooms with a stem width less than 8.85 centimeters and a stem color other than gray. In this case, if the stem width is less than 3.27 centimeters and the stem is not white then we are going to consider the stem width again. In this case if the mushroom has a stem width less or equal to 7.06 centimeters than it will be poisonous. Let's now go back a little and consider the case in which the stem height is greater than 3.27 centimeters. In this case, the tree becomes slightly more complex than the tree obtained with Scaled Entropy. In fact, considering the tree in Figure 4,

we can see that the mushroom was directly classified as poisoned if the stem height was greater than 3.2 centimeters. In the tree obtained through the Gini impurity, instead, we can find a more complex articulation that takes into account the color of the cap. If this is gray then the mushroom will be classified as edible. If instead the color of the cap were not gray but the gill-attachment was convex the mushroom would be classified in the same way as edible. The rest of the tree instead is completely equal to the one obtained through the Scaled Entropy method.

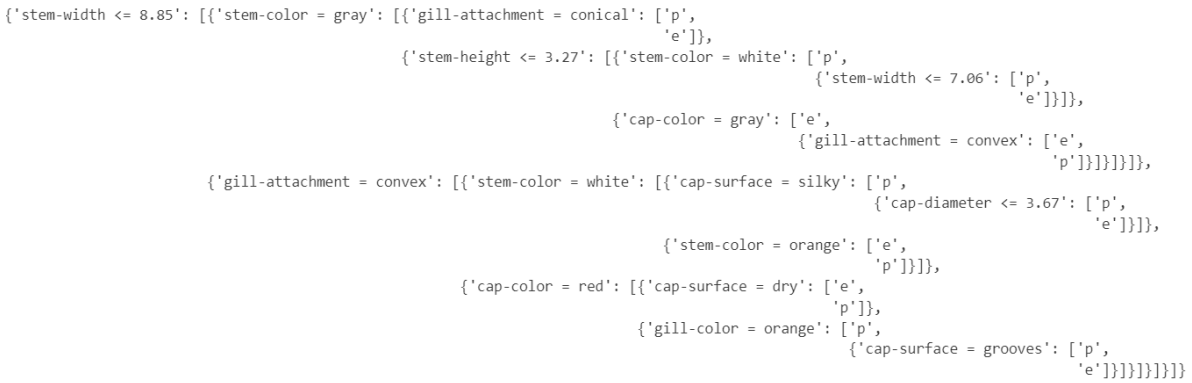


Figure 11: Tree obtained with Gini Impurity, max depth = 5, min sample = 500

Bhattacharyya coefficient model Let's now move on to the analysis of the tree obtained using the Bhattacharyya coefficient split method, of which we report the formula:

$$\text{Bhattacharyya coefficient: } \psi(p) = \sqrt{p(1-p)}$$

As we can see from Figure 12, the tree obtained is the one that differs most from the other trees. Starting from the root, which is the same as the previous models, the algorithm evaluates whether the stem width is less than 8.85 centimeters. If so, it proceeds to calculate the stem height and if this is less or equal than 3.2 centimeters then it proceeds to consider the stem color. If it is white then the mushroom is classified as edible, otherwise if the stem color is different from white but the stem height is less than 2.75 centimeters then the mushroom is poisonous. If instead the stem height is not less than or equal to 2.75 centimeters but the stem width is less than 7.06 centimeters then we will classify the mushroom as poisonous again.

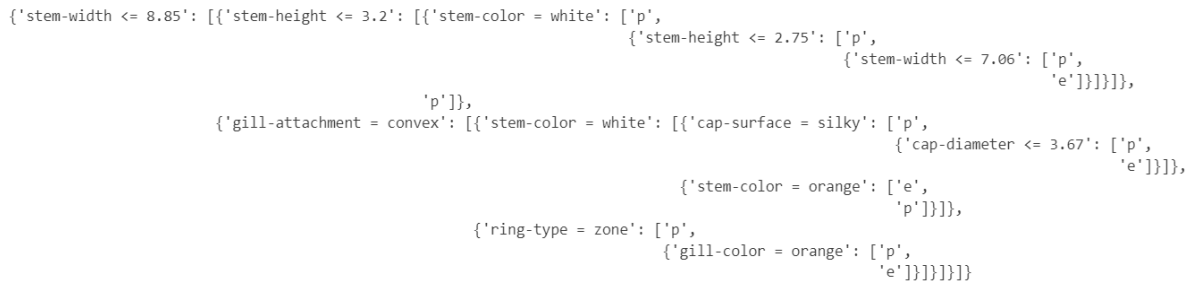


Figure 12: Tree obtained with Bhattacharyya coefficient, max depth = 5, min sample = 500

In case the stem width was less than or equal to 8.85 centimeters but the stem height was greater than 3.2 centimeters then we would have mushrooms classified as poisonous. As for the second branch, that is the one that derives from considering the stem width greater than 8.85 centimeters at the root, the same classification obtained previously by the two models is obtained. What changes is the third branch, which derives when the gill attachment is considered different from convex. In this case the ring type must be taken into consideration, if it is zone then the mushroom will be classified as poisonous, while if it is not of the zone type the gill color must be considered to determine the nature of the mushroom. If the gill color is orange the mushroom will be poisonous, otherwise it will be classified as edible.

Comparison We now report a short table that compares the models based on accuracy and 0-1 loss to see if there are substantial differences based on the split criterion chosen. Obviously the trees are created using the same parameters, that is, a maximum depth of 5 and a minimum of 500 samples to be able to classify a node.

	Scaled Entropy	**Gini Impurity**	**Bhattacharyya coefficient**
Accuracy Train	0.771	0.788	0.720
Accuracy Test	0.762	0.780	0.712
0-1 loss Train	0.228	0.211	0.279
0-1 loss Test	0.238	0.219	0.287

Table 1: Comparison between models with same parameters

Looking at Table 1, it can be noted that among the three models, the one that is best suited for classification is the Gini impurity method. The difference is not so marked with the other two methods, also because it differs little from the Scaled Entropy tree. The Bhattacharyya coefficient method, which is the most original, is also the poorest, even if the result is highly acceptable. However, its train and test accuracy is 6.5 percentage points lower than the results obtained by the tree obtained through Gini impurity and 5 percentage points lower than the accuracy of the tree obtained through Scaled Entropy. Furthermore, the model obtained with the Gini Impurity method is the fastest in terms of computation. In fact, the model obtained with Scaled Entropy takes about 8 to 10 minutes to be output, as well as the method obtained with Bhattacharyya coefficient. The tree that is created using the Gini Impurity as a split method is instead faster by 1 minute on average.

8 Conclusion

Having completed our work, we can now draw the guidelines. Starting from the description of the dataset we noticed how it was evenly divided between edible and poisonous mushrooms, so as to be more confident in not having an altered model. After a theoretical description of how a decision tree works and a brief explanation of the functions used by the algorithm, we moved on to the description and evaluation of the model. The first tree was created starting from a maximum depth of 5 and a minimum number of examples, or a minimum number of observations that a node must have before it can be classified equal to 200. The second model instead consisted of a more concise version of the first as it took into consideration 500 minimum examples. Both trees were created using the Scaled Entropy method and gave very positive levels of accuracy. More precisely, the tree that considers a minimum of 500 examples obtained a test accuracy of 76.2%. The confusion matrix confirms that the model generally performs well without over-fitting or under-fitting. Further refinement is achieved through hyper-parameter tuning, adjusting the tree's depth and minimum samples per node to optimize performance. The hyper-parameter tuning was then carried out considering a variation of the minimum number of samples from 500 to 2500 with an increase of 500 at each iteration and a depth that varied from 3 to 6. Through this technique it was possible to verify the lack of under-fitting and over-fitting and the model showed excellent levels of adaptation also in the case of the test set. To conclude, the trees created using two other split methods were compared, namely the Gini impurity and the Bhattacharyya coefficient. The model obtained through the Gini impurity proved to be very similar to that of the Scaled Entropy and improved the classification performance with equal parameters. The last model instead proved to be the most original but also the poorest, resulting in a higher training and test error and a lower accuracy. In conclusion, the decision tree classifier demonstrates robust performance in classifying mushrooms, with Gini impurity being the most reliable splitting criterion.