Binary Classification with Decision Trees: Identifying Poisonous Mushrooms

Machine Learning and Statistical Learning (Machine Learning module)

Daniele Piazza, 45740A*
*Data Science for Economics, Università degli Studi di Milano
daniele.piazza1@studenti.unimi.it

1 Introduction

The aim of this project is to build a binary decision tree from scratch and then test it on a mushroom dataset. The goal is to predict whether a mushroom is poisonous or edible based on numerous attributes.

The first step was to analysed the data, divide it into training and test sets, before proceeding with a data cleaning phase.

The central point of the project was the construction of the decision tree for binary classification. Starting by creating a class for the nodes and then by developing the tree from the fit and prediction functions.

Three methods for the splitting criteria were used: Gini impurity, scaled entropy and squared error. As for the stopping criteria, three were implemented: maximum depth, minimum sample splitting and minimum impurity decrease. These criteria are essential to avoid underfitting or overfitting of the model.

To find the best criteria, the hyperparameters were adjusted, with and without cross-validation, using 0-1 loss as the evaluation metric. This step was essential to optimise the performance of the decision tree.

Finally, a random forest was constructed using the structures previously presented, in order to further study the predictive capabilities of the model and the importance of the features.

2 Data Analysis and Preprocessing

The mushroom dataset includes 61069 hypothetical mushrooms with caps based on 173 species (353 mushrooms per species).[1] Each mushroom is identified as edible or poisonous, and each one is described by the following attributes:

- **Classification:** Edible or Poisonous
- **Cap Diameter (Numerical):** Float number in cm
- **Cap Shape (Categorical):** Bell, Conical, Convex, Flat, Sunken, Spherical, Others
- **Cap Surface (Categorical):** Fibrous, Grooves, Scaly, Smooth, Shiny, Leathery, Silky, Sticky, Wrinkled, Fleshy, d
- **Cap Color (Categorical):** Brown, Buff, Gray, Green, Pink, Purple, Red, White, Yellow, Blue, Orange, Black
- **Does Bruise or Bleed (Categorical):** True, False
- **Gill Attachment (Categorical):** Adnate, Adnexed, Decurrent, Free, Sinuate, Pores, None

- **Gill Spacing (Categorical):** Close, Distant, None
- **Gill Color (Categorical):** Brown, Buff, Gray, Green, Pink, Purple, Red, White, Yellow, Blue, Orange, Black, None
- **Stem Height (Numerical):** Float number in cm
- **Stem Width (Numerical):** Float number in mm
- **Stem Root (Categorical):** Bulbous, Swollen, Club, Cup, Equal, Rhizomorphs, Rooted
- **Stem Surface (Categorical):** Fibrous, Grooves, Scaly, Smooth, Shiny, Leathery, Silky, Sticky, Wrinkled, Fleshy, None
- **Stem Color (Categorical):** Brown, Buff, Gray, Green, Pink, Purple, Red, White, Yellow, Blue, Orange, Black, None
- **Veil Type (Categorical):** Partial, Universal
- **Veil Color (Categorical):** Brown, Buff, Gray, Green, Pink, Purple, Red, White, Yellow, Blue, Orange, Black, None
- **Has Ring (Categorical):** True, False
- **Ring Type (Categorical):** Cobwebby, Evanescent, Flaring, Grooved, Large, Pendant, Sheathing, Zone, Scaly, Movable, None
- **Spore Print Color (Categorical):** Brown, Buff, Gray, Green, Pink, Purple, Red, White, Yellow, Blue, Orange, Black
- **Habitat (Categorical):** Grasses, Leaves, Meadows, Paths, Heaths, Urban, Waste, Woods
- **Season (Categorical):** Spring, Summer, Autumn, Winter

The data was split into training and test sets (80% training, 20% test). To prevent data leakage, only the training set was utilized to clean the dataset.

- Columns with more than 80% missing values were removed:
  - `stem-root`
  - `veil-type`
  - `veil-color`
  - `spore-print-color`
- Missing values, which were only present for categorical attributes, were replaced using the training set mode. No null values were found for the numerical attributes. However, for `stem height` and `stem width` several mushrooms had a value of "0.0". As it was not clear whether these were errors, they were retained. If null values had been present, the median would have been used as a replacement to avoid the influence of outliers.
- Duplicate rows, although few, were removed to reduce the risk of overfitting.
- The classes in the dataset were balanced, so there is no need to apply oversampling or undersampling technique.

No scaling or outlier removal was performed, as decision trees do not require these preprocessing steps.

3 Decision Tree

A `decision tree` is a machine learning model used for classification and regression tasks. It operates by recursively partitioning the input space into increasingly homogeneous regions, ultimately assigning a predicted class or value to each region. The tree structure consists of interconnected elements called `nodes`, each representing a decision point or a prediction outcome. [2]

3.1 Structure of a decision tree

At the top of the tree is the `root node`, which represents the entire data set and is the starting point of the process. The tree expands downwards through a hierarchy of nodes, with `parent node` branching into `child nodes` based on specific `splitting criteria`. This hierarchical structure facilitates a step-by-step division of the data, guided by thresholds or tests at each node, the splitting continues until the final decision or stopping criteria is reached. As this is a binary classification, at each split the decision tree divides the data into two subsets based on the predictor variables. The leaf nodes at the end of each branch represent the final result of the tree.

3.2 How a decision tree works

Decision trees work by recursively partitioning the input space based on selected features, aiming to create subsets of data that are increasingly homogeneous with respect to the target variable. This process is guided by specific splitting criteria, for binary classification tasks the most common are:

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

At each step, the goal is to reduce the heterogeneity of the groups, meaning that each split should create more homogeneous subgroups compared to the previous level. At the end, the tree's leaf nodes represent highly homogeneous groups that can be assigned to a particular class or output value.

To build the tree, a decision tree algorithm uses recursion. This process begins at the root node, which represents the entire feature space, and recursively divides the data into two child nodes at each step based on the value of a chosen predictor variable. The algorithm evaluates all possible splits at each node and selects the one that maximizes homogeneity or minimizes impurity.

The splitting continues until a `stopping criterion` is reached, such as:

- Maximum tree depth,
- Minimum number of samples in a node,
- Minimum impurity decrease

3.3 Model Evaluation

The evaluation of decision trees for binary classification involves various performance metrics, which measure the effectiveness of the model in distinguishing between the two classes. These include 0-1 loss, accuracy, precision, recall, F1 score, and the confusion matrix.

Before presenting the formulas for these metrics, some essential terms used in binary classification are:

- `True Positives (TP)`: The number of times the model correctly predicted the positive class.
- `False Positives (FP)`: The number of times the model incorrectly predicted the positive class.
- `True Negatives (TN)`: The number of times the model correctly predicted the negative class.
- `False Negatives (FN)`: The number of times the model incorrectly predicted the negative class.

Now, using these definitions, the most commonly used evaluation metrics can be defined:

- `0-1 Loss` is a simple metric that calculates the proportion of incorrect predictions out of the total number of predictions. It is defined as:
$$L_{0-1} = \frac{1}{n}\sum_{i=1}^{n}\ell_i$$
where:

  - n is the total number of predictions,
  - $\ell_i = \mathbb{I}(y_i \neq \hat{y}_i)$ is the loss for the i-th sample,
  - y_i is the true label for the i-th sample,
  - $\hat{y}_i$ is the predicted label for the i-th sample,
  - $\mathbb{I}(\cdot)$ is the indicator function, which returns 1 if $y_i \neq \hat{y}_i$, and 0 otherwise.

- `Accuracy` measures the proportion of correct predictions out of the total number of predictions. It can be written as:

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

  Alternatively, accuracy can also be written as:

$$\text{Accuracy} = \frac{1}{n} \sum_{i=1}^{n} \mathbb{I}(y_i = \hat{y}_i)$$

  where:

  - n is the total number of predictions,
  - y_i is the true label for the i-th sample,
  - $\hat{y}_i$ is the predicted label for the i-th sample,
  - $\mathbb{I}(\cdot)$ is the indicator function, which returns 1 if $y_i = \hat{y}_i$, and 0 otherwise.

- `Precision` is the ratio of correctly predicted positive instances to the total predicted positives. It is defined as:

$$\text{Precision} = \frac{TP}{TP + FP}$$

- `Recall` is the ratio of correctly predicted positive instances to the total actual positives. It is defined as:

$$\text{Recall} = \frac{TP}{TP + FN}$$

- `F1 Score` is the harmonic mean of precision and recall, providing a balance between the two. It is defined as:

$$F1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

- `Confusion Matrix` is a 2x2 table that summarizes the performance of a classification model by showing the counts of TP, FP, TN, and FN:

/	Predicted Positive	Predicted Negative
Actual Positive	TP	FN
Actual Negative	FP	TN

  The confusion matrix allows for a detailed analysis of the classification results, providing more insights into model performance beyond accuracy alone.

4 Implementation

This chapter illustrates how the decision tree has been implemented.

4.1 Node Class

The `Node` class is a central component of the decision tree structure.

4.1.1 Attributes

Each node can have the following attributes:

- `feature`: The index of the feature used to split the data.
- `thr`: The threshold value used for the split.
- `left`: The left child node.
- `right`: The right child node.
- `value`: The value assigned to the leaf node.
- `categorical`: A boolean indicating whether the feature is categorical or numerical.

4.1.2 Methods

The `Node` class includes the method `is_leaf()`, which returns `True` if the node is a leaf node and `False` otherwise.

4.2 DecisionTree Class

The `DecisionTree` class is responsible for building and managing the decision tree.

4.2.1 Attributes

The main attributes are:

- `max_depth`: The maximum depth the tree can reach.
- `min_samples_split`: The minimum number of samples required to split a node.
- `min_impurity_decrease`: The minimum decrease in impurity required for a split to occur.
- `criterion`: The criterion used to evaluate splits (e.g., `'gini'`, `'scaled_entropy'`, or `'squared'`).

4.2.2 Methods

The main methods are:

- **Fit**

  The decision tree is trained using the `fit` method, which recursively splits the data. The algorithm begins at the root and continues down the tree by recursively finding the best split at each node.

  Pseudocode for the fitting method:

Algorithm 1 Recursive Fit

Input: Samples X, target values y, depth *depth*
num_sample ← dimension of X
if *depth* > *max_depth* **or** *num_samples* < *min_samples_split* **or** all samples have the same class **then**
  return leaf node with the most common class
end if
split, *gain* ← Find Best Split(X, y)
if *gain* < *min_impurity_decrease* **then**
  return leaf node with the most common class
end if
$X_{left}, y_{left}, X_{right}, y_{right}$ ← Split Samples(X, y, *split*)
left_child ← RecursiveFit(X_{left}, y_{left}, *depth* + 1)
right_child ← RecursiveFit(X_{right}, y_{right}, *depth* + 1)
return node with the split information, left_child, and right_child

- **Best Split**

  The `best split` is determined by evaluating the `information gain` for each feature and threshold combination. In the implementation of the decision tree for the project, the algorithm first checks whether the feature is categorical or numerical. For numerical features, thresholds on a single feature are set from the 10th to the 90th percentiles for computational efficiency. For categorical features, membership tests using each unique value are applied as a possible split point.

Algorithm 2 Best Split

Input: Samples X, target values y
for each feature f in X **do**
  if feature f is numerical **then**
    split ← percentiles from 10 to 90
  else if feature f is categorical **then**
    split ← all unique values of f
  end if
  Calculate information gain for each possible *split*
end for
return the split and gain with the highest information gain

- **Information Gain Calculation**

  Information gain measures the reduction in impurity achieved by splitting a node. The impurity is calculated based on the criterion chosen during the initialization of the tree (`gini`, `scaled_entropy`, or `squared`).

  The general formula for information gain is [2]:

$$IG = \psi(p) - \left(\frac{N_l}{N} \cdot \psi(l) + \frac{N_r}{N} \cdot \psi(r) \right)$$

  where:

  - IG is the Information Gain,
  - $\psi(p)$ is the impurity of the parent node,
  - $\psi(l)$ is the impurity of the left child node,
  - $\psi(r)$ is the impurity of the right child node,
  - N_l is the number of samples in the left child node,
  - N_r is the number of samples in the right child node,
  - N is the total number of samples in the parent node.

  So, Information Gain is the reduction in impurity achieved by splitting the parent node into child nodes.

- **Feature Importance**

  The importance of a feature is measured by the total information gain it contributes across all nodes where it is used as a splitting criterion. The feature importance is calculated by summing the information gain for each feature and normalizing the values so that the total importance sums to 1.

  The feature importance is given by:

$$FI_j = \frac{\sum_{n \in N_j} IG_n}{\sum_{m \in N} IG_m}$$

  where:

  - FI_j is the importance of feature j,
  - $\sum_{n \in N_j} IG_n$ is the sum of the Information Gain IG for all nodes n where feature j was used for splitting,
  - $\sum_{m \in N} IG_m$ is the sum of the Information Gain across all nodes m in the entire tree.

  So, the feature importance is computed as the ratio of the total Information Gain for feature j to the total Information Gain for all features in the tree.

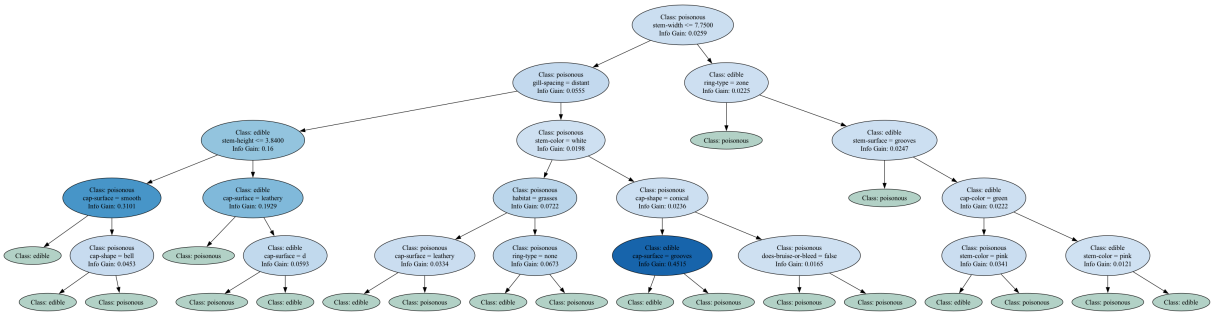

Figure 1: Decision Tree on Mushroom dataset with maximum depth of 5, minimum samples spit of 100 and Gini as criterion. The nodes are colored based on their information gain, with darker shades representing higher information gain. Leaf nodes are shaded differently to distinguish them from decision nodes.

5 Hyperparameter Tuning and Cross-Validation

5.1 Hyperparameter Tuning

The fine-tuning of `hyperparameters` refers to the process of optimising the hyperparameters of a machine learning model. Hyperparameters are settings that must be specified before training the model and can significantly influence the final performance of the model.

In our decision trees, the hyperparameters are the stopping and splitting criteria. The correct adjustment of these hyperparameters is essential because if the tree grows too large, it can capture noise in the data, resulting in poor generalisation (overfitting). On the other hand, if the tree is too small, it may fail to capture important patterns in the data (underfitting).

Hyperparameters should not be confused with model `parameters`. Hyperparameters are defined by the user before training, while parameters are values that the model learns automatically during training. An example of a parameter in the Decision Tree is the splitting value associated with each node.

The goal of hyperparameter tuning is to find the optimal balance that maximises the performance of the model on the unseen data.

5.2 Cross-Validation

Cross-validation is a technique used to more reliably assess the performance of a model, particularly when adjusting hyperparameters. It works by splitting the data into several subsets or `folds`. In the project,

k-fold cross-validation was implemented, the data is divided into k parts of equal size. The model is trained k times, each time using $k-1$ folds as the training set and the remaining folds as the test set. This process is repeated for each fold and the average loss of all iterations is taken as the final evaluation score. [2]

The pseudocodes of the two algorithms:

Algorithm 3 K-Fold

Input: Samples X, Number of fold n_splits
Inizialize empty list $folds$
Shuffle indices of samples
Determine the size of each fold
for each $fold$ **do**
    Assign train and test indices
    Append (train, test) pair to $folds$
end for
return folds

Algorithm 4 Cross-Validation

Input: Samples X, target values y, Number of fold n_splits
$folds \leftarrow$ K-Fold(X, y, n_splits)
Initialize empty list $losses$
for each $(train\_indices, test\_indices)$ in $folds$ **do**
    Fit model on training set
    $loss \leftarrow$ compute 0-1 loss on test set
    Append $loss$ to $losses$
end for
return The mean of the $losses$

This technique provides a better estimate of model performance on unseen data. In addition, it allows for a more robust comparison of models or hyperparameter configurations, as it decreases the potential risk of overfitting or underfitting that could result from relying on a single train-test split.

6 Random Forest

Random Forest is an ensemble learning method that combines multiple decision trees to improve predictive performance and reduce overfitting. It consists of building multiple decision trees during training and returning the value that has been predicted multiple times. [2]

6.1 Key Points of Random Forest

- **Bootstrap Sampling**: Each tree is trained on a different bootstrap sample, a random sample with replacement, of the training data, leading to diversity among the trees. [3]
- **Feature Randomness**: For each tree, only a random subset of features is considered, usually the $\sqrt{features}$, for splitting at each node, which reduces correlation among trees and improves model robustness.
- **Feature Importance**: Random Forest provides a reliable measure of feature importance.

6.2 Pseudocode for Random Forest

Algorithm 5 Random Forest

Input: Training data (X, y), number of trees n_trees, max features $max\_features$
Initialize empty list of trees $trees$
for $i = 1$ **to** n_trees **do**
    $X\_bootstrap, y\_bootstrap \leftarrow \text{BootstrapSample}(X, y)$
    $X\_subset \leftarrow \text{RandomSubset}(X\_bootstrap, max\_features)$
    $tree \leftarrow \text{DecisionTree}(X\_subset, y\_bootstrap)$
    Append $(tree, X\_subset.columns)$ to $trees$
end for
return $trees$

Algorithm 6 Predict Function for Random Forest

Input: Test data X
Initialize empty array $tree\_predictions$
for each $tree$, $features$ in $trees$ **do**
    $tree\_predictions[i] \leftarrow tree.predict(X[features])$
end for
Initialize empty array $majority\_votes$ of size $n\_samples$
for each sample j in range($n\_samples$) **do**
    $majority\_votes[j] \leftarrow$ most predicted value for sample j
end for
return $majority\_votes$

7 Results

After hyperparameter tuning, both with and without cross-validation, the following parameters were found as the best ones for the decision tree, using 0-1 loss:

```
Best hyperparameters: {'max_depth': 50, 'min_samples_split': 2, 'min_impurity_decrease': 0.0, 'criterion': 'squared'}, loss: 0.0013334700994973843
Best hyperparameters CV: {'max_depth': None, 'min_samples_split': 10, 'min_impurity_decrease': 0.0, 'criterion': 'gini'}, loss: 0.001148734781476601
```

Figure 2: Best Hyperparameters found with and without cross-validation and the 0-1 loss evaluated on the validation set

The loss concerns the validation set. From these results, we can observe that the dataset does not suffer of overfitting, as indicated by the similar performance across different validation strategies. The similarity in loss between cross-validation and the initial hyperparameter tuning suggests that the model generalizes well across unseen data, with no significant performance degradation across the different validation folds.

After re-fitting with the best parameters found on the entire train set, the three splitting critaria were compared. The results obtained are as follows on the test set:

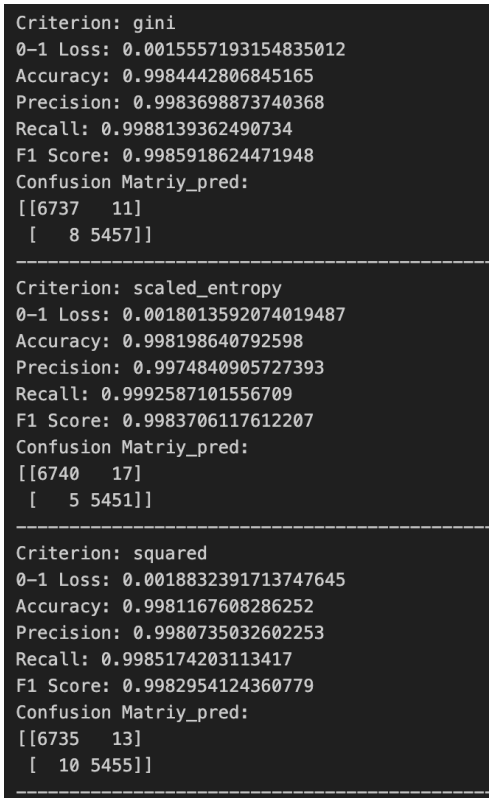

Figure 3: Comparison within the three splitting criteria with the best hyperparameters

This comparison further highlights the model's robustness, demonstrating that the choice of splitting criterion has a relatively minor impact on overall performance when using the optimized hyperparameters.

In addition, hyperparameter tuning for the random forest has been performed. Due to computational reasons a small parameter grid has been used.

Regarding the importance of the features the random forest identified the following in descending order:

Figure 4: Feature importance in descending order from a Random Forest model with 200 trees, max_depth of None, min_sample_split of 10 using the Gini criterion and 'sqrt' for max_features.

This feature importance values provides the factors that have the most significant influence on model predictions, which can be useful for interpretability and further refinement of the model.

References

[1] Dennis Wagner, D. Heider, and Georges Hattab. *Secondary Mushroom*. UCI Machine Learning Repository. 2021.

[2] Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge, UK: Cambridge University Press, 2014. ISBN: 9781107057135.

[3] Gareth James et al. *An Introduction to Statistical Learning: with Applications in R*. New York, NY: Springer, 2013. ISBN: 9781461471387.