Tree predictors for binary classification

Ikram Ait Taleb Naser

Declaration of Authorship

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.

Abstract

This project focuses on the implementation of binary tree predictors from scratch for the task of determining whether mushrooms in the provided dataset are poisonous or edible. The Mushroom dataset contains categorical and nominal features and serves as a binary classification problem with the target labels indicating poisonous or edible classes.The tree predictors use single-feature binary tests as decision criteria at internal nodes, based on thresholds for numerical/ordinal features or membership tests for categorical features.

The objectives of the project include:

- Designing classes for tree predictors and their nodes
- Implementing different splitting and stopping criteria
- Training and evaluating the tree predictors
- Performing hyperparameter tuning and analyzing overfitting or underfitting in the models

My report includes:

A description of the methodology, including the splitting and stopping criteria used.
A discussion of the models' performance, including training and validation errors.
An analysis of overfitting and how I addressed it through hyperparameter tuning and pruning.
A comparison of the different splitting criteria and their impact on performance.
A discussion of the hyperparameter tuning process and its results.

1 Introduction

A tree predictor has the structure of an ordered and rooted tree where each node is either a leaf (if it has zero children) or an internal node (if it has at least two children). An ordered tree is one where the children of any internal node are numbered consecutively. Hence, if the internal node v has $k \geq 2$ children, we can access the first child, the second child, and so on until the k-th child. Fix $X = X_1 \times \cdots \times X_d$, where X_i is the range of the i-th attribute x_i. A tree predictor $h_T : X \to Y$ is a predictor defined by a tree T whose internal nodes are tagged with tests and whose leaves are tagged with labels in Y. A test on attribute i for an internal node with k children is a function $f : X_i \to \{1, \ldots, k\}$. The function f maps each element of X_i to the node's children.

The prediction $h_T(x)$ is computed as follows. Start by assigning $v \leftarrow r$, where r is the root of T;

1. If v is a leaf ℓ, then stop and let $h_T(x)$ be the label $y \in Y$ associated with ℓ;
2. Otherwise, if $f : X_i \to \{1, \ldots, k\}$ is the test associated with v, then assign $v \leftarrow v_j$ where $j = f(x_i)$ and v_j denotes the j-th child of v;
3. Go to step 1.

If the computation of $h_T(x)$ terminates in leaf ℓ, we say that the example x is routed to ℓ. Hence $h_T(x)$ is always the label of the leaf to which x is routed.

Our is the case of a binary classification $Y = \{-1, 1\}$ and we only consider complete binary trees, i.e., all internal nodes have exactly two children. The idea is to grow the tree classifier starting from a single-node tree (which must be a leaf) that corresponds to the classifier assigning to any data point the label that occurs most frequently in the training set. The tree is grown by picking a leaf (at the beginning there is only a leaf to pick) and replacing it with an internal node and two new leaves.

Suppose we have grown a tree T up to a certain point, and the resulting classifier is h_T. We start by computing the contributions of each leaf to the training error $\ell_S(h_T)$ (each x is classified by some leaf, the leaf which x is routed to). For each leaf ℓ, define

$$S_\ell \equiv \{(x_t, y_t) \in S : x_t \text{ is routed to } \ell\}.$$

That is, S_ℓ is the subset of training examples that are routed to ℓ. Define further two subsets of S_ℓ, namely

$$S_\ell^+ \equiv \{(x_t, y_t) \in S_\ell : y_t = +1\} \quad \text{and} \quad S_\ell^- \equiv \{(x_t, y_t) \in S_\ell : y_t = -1\}.$$

For each leaf ℓ, let

$$N_\ell^+ = |S_\ell^+|, \quad N_\ell^- = |S_\ell^-|, \quad \text{and} \quad N_\ell = |S_\ell| = N_\ell^+ + N_\ell^-.$$

In order to minimize the training error $\ell_S(h_T)$, the label associated with ℓ must be

$$y_\ell = \begin{cases} +1 & \text{if } N_\ell^+ \geq N_\ell^-, \\ -1 & \text{otherwise.} \end{cases}$$

The recursive splitting of data is contingent upon specific criteria, including entropy, Gini impurity, and other measures of information gain. These criteria evaluate the efficacy of a given feature in differentiating between target classes. By traversing the tree, predictions are made based on the path a given instance follows until it reaches a leaf node, which will represent the predicted class. In this project, the implementation of binary decision trees is tailored to predict the edibility of mushrooms.

Decision nodes within the tree apply single-feature binary tests, which either compare numeric or ordinal attributes against a threshold, or evaluate membership for categorical attributes. Several splitting and stopping criteria are explored to improve the tree's effectiveness and address overfitting issues. Additionally, hyperparameter tuning is conducted to optimize parameters such as maximum tree depth and minimum impurity decrease. Post-training pruning techniques are also employed to further control overfitting.

2 The Dataset

The dataset used for this project is the **Mushroom dataset**, sourced from the UCI Machine Learning Repository. This dataset contains **8,124 samples**, each representing a mushroom specimen described by **22 categorical features**, including characteristics such as cap shape, gill color, and odor. The target label is binary, indicating whether a given mushroom is `edible` or `poisonous`. The primary-data dataset includes 173 species of mushrooms with caps from various families and one entry for each species. Each species is identified as definitely edible, definitely poisonous, or of unknown edibility and not recommended (the latter class was combined with the poisonous class). Of the 20 variables, 17 are nominal and 3 are metrical. The values of each nominal variable are a set of possible values and for the metrical variables a range of possible values.

The secondary-data dataset includes 61069 hypothetical mushrooms with caps based on 173 species (353 mushrooms per species). Each mushroom is identified as definitely edible, definitely poisonous, or of unknown edibility and not recommended (the latter class was combined with the poisonous class). Of the 20 variables, 17 are nominal and 3 are metrical. Table 1 summarize all the features.

2.1 Preprocessing

Since the dataset consists of categorical features, a custom label encoding function is applied to convert categorical values into numerical representations, in which the function first identifies all unique values in the given categorical column, then a dictionary is created where each unique value is assigned a unique integer index then categorical values in the column are replaced with their corresponding numerical indices.

When the secondary dataset (secondary_data) is provided, it applies systematic sampling to reduce computational overhead, using a configurable sampling ratio (default is 10%) before merging.

The dataset is finally partitioned into training and testing subsets using an 80-20 split.

3 Methodology

The following pseudo-code outlines the process of building a binary decision tree classifier based on a training set S.

1: **Input:** Training set S, initialized at the root node l.
2: **Initialization:**

- Set $S_l = S$.
- Assign the label at node l as:

$$y_l = \begin{cases} +1, & \text{if } N_l^+ \geq N_l^- \\ -1, & \text{otherwise} \end{cases}$$

Table 1: Variable Descriptions in the Mushroom Dataset

#	Variable Name	Type	Values/Details
1	cap-diameter	Metrical (m)	Float values (cm): range (min, max) or mean.
2	cap-shape	Nominal (n)	bell (b), conical (c), convex (x), flat (f), sunken (s), spherical (p), others (o).
3	cap-surface	Nominal (n)	fibrous (i), grooves (g), scaly (y), smooth (s), shiny (h), leathery (l), silky (k), sticky (t), wrinkled (w), fleshy (e).
4	cap-color	Nominal (n)	brown (n), buff (b), gray (g), green (r), pink (p), purple (u), red (e), white (w), yellow (y), blue (l), orange (o), black (k).
5	does-bruise-bleed	Nominal (n)	bruises-or-bleeding (t), no (f).
6	gill-attachment	Nominal (n)	adnate (a), adnexed (x), decurrent (d), free (e), sinuate (s), pores (p), none (f), unknown (?).
7	gill-spacing	Nominal (n)	close (c), distant (d), none (f).
8	gill-color	Nominal (n)	Same as cap-color + none (f).
9	stem-height	Metrical (m)	Float values (cm): range (min, max) or mean.
10	stem-width	Metrical (m)	Float values (mm): range (min, max) or mean.
11	stem-root	Nominal (n)	bulbous (b), swollen (s), club (c), cup (u), equal (e), rhizomorphs (z), rooted (r).
12	stem-surface	Nominal (n)	Same as cap-surface + none (f).
13	stem-color	Nominal (n)	Same as cap-color + none (f).
14	veil-type	Nominal (n)	partial (p), universal (u).
15	veil-color	Nominal (n)	Same as cap-color + none (f).
16	has-ring	Nominal (n)	ring (t), none (f).
17	ring-type	Nominal (n)	cobwebby (c), evanescent (e), flaring (r), grooved (g), large (l), pendant (p), sheathing (s), zone (z), scaly (y), movable (m), none (f), unknown (?).
18	spore-print-color	Nominal (n)	Same as cap-color.
19	habitat	Nominal (n)	grasses (g), leaves (l), meadows (m), paths (p), heaths (h), urban (u), waste (w), woods (d).
20	season	Nominal (n)	spring (s), summer (u), autumn (a), winter (w).

While stopping criterion is not met

3: Select a node l to split, creating an internal node v with new leaves l' and l''.

4: Choose an attribute i.

5: Define a test function $f : X_i \to \{1, 2\}$.

6: Assign f to node v and partition S_l as follows:

$$S_{l'} = \{(x_t, y_t) \in S_l \mid f(x_{t,i}) = 1\}, \quad \text{set } y_{l'}$$
$$S_{l''} = \{(x_t, y_t) \in S_l \mid f(x_{t,i}) = 2\}, \quad \text{set } y_{l''}$$

3.1 Node class

The core structure of the decision tree relies on its fundamental building block, the `Node` class. Each node in the tree represents a decision point or a final classification outcome. The implementation defines a binary structure where nodes can have at most two children: a left child and a right child.

3.1.1 Node Attributes

The `Node` class contains the following attributes:

- `feature`: This attribute stores the index of the feature used to split the data at the current node. If the node is a leaf, this value remains `None`.

- `threshold`: For numerical or ordinal features, this value represents the threshold used in the binary decision. Instances with values lower than or equal to the threshold are sent to the left child, while instances with higher values go to the right child.
- `left and right`: These attributes represent the child nodes. The left child contains samples that satisfy the decision rule (values less than or equal to the threshold or belonging to a specific category), while the right child holds samples that do not.
- `value`: If the node is a leaf, this attribute stores the class label assigned to it based on the most frequent label in the subset of data reaching that node.
- `is_leaf`: A Boolean flag that indicates whether the node is a leaf (i.e., a terminal node with a classification decision).
- `is_categorical`: a Boolean flag that indicates whether the feature used for splitting is categorical or numerical. If True, the node splits data based on exact category matches instead of numerical thresholds.

3.1.2 Node Functionality

Nodes serve as the essential components in constructing a decision tree, forming its hierarchical structure. The tree is constructed recursively, beginning from a root node and expanding downward by repeatedly partitioning the dataset.

Internal nodes contain a feature and a threshold (for numerical features) or a category check (for categorical features). These nodes act as decision points, directing samples toward the appropriate subtree. In contrast, leaf nodes are responsible for outputting the final classification label for the given input instance.

During training, nodes are split iteratively based on the selected splitting criterion (e.g., Gini impurity, entropy reduction or misclassification error), ensuring that each decision maximizes the homogeneity of the resulting subsets. If a stopping criterion is met—such as reaching a predefined depth or observing minimal impurity decrease—the growth of the tree halts, and a leaf node is assigned a class value.

The structured definition of nodes allows for efficient traversal during inference, where a new instance is passed from the root node down through the tree, following the appropriate conditions at each step until reaching a leaf node that provides the final classification.

3.2 Binary Decision Tree class

The `BinaryDecisionTree` class implements a binary classification decision tree that partitions data based on a single-feature criterion. It recursively splits the dataset into increasingly homogeneous subsets until a stopping condition is met. This class serves as the main structure for constructing tree predictors, encapsulating the logic for training, predicting, and handling tree traversal.

3.2.1 Class Attributes and Initialization

The tree is initialized with key hyperparameters that control its growth and prevent overfitting:

- `criterion`: A function defining the splitting strategy, such as the minimization of entropy, Gini impurity or misclassification error.
- `max_depth`: The maximum depth allowed for the tree, ensuring controlled growth.
- `min_samples_leaf`: This parameter sets the minimum number of samples required in a leaf node. If a split results in a child node containing fewer samples than this value, the split is discarded. It helps prevent overfitting by ensuring nodes are not too small.
- `min_impurity_decrease`: The minimum reduction in impurity required to justify a split.
- `min_weight_fraction_leaf`: Instead of specifying an absolute number of samples like `min_sample_leaf`, this parameter defines the minimum fraction of total sample weight that a leaf node must contain. It is useful when dealing with weighted datasets.

- `cpp_alpha`: penalizes overly complex trees by adding a regularization term to the impurity measure. Increasing it leads to more pruning, reducing overfitting.
- `root`: The root node of the decision tree, initialized as `None` and assigned after training.
- `feature_types_`: A NumPy array tracking the importance of each feature based on how often and significantly it contributes to the splits.
- `n_classes`: The number of unique class labels present in the dataset, determined from the training labels. This helps in handling multi-class classification problems.

These parameters enable customization of the learning process to optimize performance while mitigating issues such as overfitting.

3.2.2 Tree Construction and Training Procedure

The training process is initiated by calling the `fit` method, which constructs the tree recursively using the `_build_tree` function. The process follows these steps:

1. If all instances in the node belong to the same class, or the maximum depth is reached, a leaf node is created, storing the majority class as the predicted value.
2. The `_find_best_split` method is invoked to identify the optimal feature and threshold for partitioning the data. This selection is made based on minimizing the impurity score calculated through the chosen splitting criterion.
3. If the impurity decrease from the selected split does not meet the `min_impurity_decrease` threshold, the node is assigned a class label without further division.
4. Otherwise, the dataset is split into left and right child subsets based on the best-found threshold, and the recursive `_build_tree` function is called to continue constructing the subtrees.

The tree-growing process terminates upon reaching a stopping criterion.

3.2.3 Finding the Optimal Split

The function `_find_best_split` scans all features to determine the threshold that minimizes impurity. It follows these steps:

- Extract all unique values for each feature as potential thresholds.
- Iterate over thresholds and compute the impurity score when data is divided at that point.
- Select the feature-threshold combination that achieves the lowest impurity score.
- Store the indices of data points belonging to the left and right splits.

This selection process directly influences the quality of the constructed tree, dictating how well-separated the data becomes with each split.

3.2.4 Making Predictions

The `predict` function classifies new data points by passing them through the decision tree from the root to a leaf node. The traversal process in `_predict_single` follows these steps:

1. Start at the root node.
2. If the node is a leaf, return its stored class label.
3. Otherwise, evaluate the decision function associated with the node.
4. If the function evaluates to `True`, move to the left subtree; otherwise, traverse the right subtree.
5. Repeat this process recursively until a leaf node is reached, and return its stored class label as the final prediction.

3.3 Splitting and Stopping criteria

3.3.1 Entropy

In the context of decision trees, entropy measures the impurity of a dataset, helping to determine the best feature to split on. A lower entropy value signifies higher homogeneity (i.e., samples in the dataset mostly belong to the same class), whereas a higher entropy value indicates a more diverse dataset with mixed class labels.

In the general case, when p_i is the fraction of examples labeled i:

$$\text{Entropy}(\{p_1, p_2, \ldots, p_k\}) = -\sum_{i=1}^{k} p_i \log(p_i)$$

If all samples belong to the same label, entropy is 0. If all samples are equally mixed, entropy is 1. In general, entropy represents a level of uncertainty. A decision tree constructs its structure by recursively partitioning the dataset based on feature values. The effectiveness of a split is evaluated using Information Gain (IG), which quantifies the reduction in entropy achieved after splitting.

Given a dataset S that is split into two subsets, S_{left} and S_{right}, using a feature X_j at threshold t, the entropy of the split is calculated as:

$$H(S_{\text{split}}) = \frac{|S_{\text{left}}|}{|S|} H(S_{\text{left}}) + \frac{|S_{\text{right}}|}{|S|} H(S_{\text{right}}) \tag{1}$$

where $|S|$ represents the number of samples in the dataset before splitting, and $|S_{\text{left}}|, |S_{\text{right}}|$ denote the sizes of the subsets.

The Information Gain is computed as:

$$IG = H(S) - H(S_{\text{split}}) \tag{2}$$

where:

- IG represents the expected reduction in entropy after performing the split,
- A higher information gain indicates a more meaningful split.

Entropy and its derived metric, information gain, are widely used for splitting decision trees because:

- Information gain measures how much a split reduces uncertainty about class labels. High information gain means better separation between different classes.
- Entropy is mostly suitable when feature values have diverse distributions and when dealing with categorical attributes.

3.3.2 Gini Index

The Gini Index is a measure of impurity used in decision tree learning to evaluate the effectiveness of a split. It quantifies the probability of incorrectly classifying a randomly chosen sample if it were labeled according to the class distribution within a dataset. A lower Gini Index indicates a purer dataset, which makes it a useful metric for decision tree splitting.

The Gini Index for a dataset S is defined as:

$$Gini(S) = 1 - \sum_{i=1}^{C} p_i^2 \tag{3}$$

where:

- $Gini(S)$ is the impurity of the dataset S,
- C is the number of possible classes,
- p_i is the probability of class i in dataset S.

During the construction of a decision tree, the dataset is recursively partitioned using feature values to form internal nodes. The best split is determined by minimizing the weighted sum of the Gini Index values of the resulting subsets. Given a dataset S, which is split into two subsets S_{left} and S_{right} using a feature X_j at threshold t, the Gini impurity of the split is:

$$Gini(S_{\text{split}}) = \frac{|S_{\text{left}}|}{|S|} Gini(S_{\text{left}}) + \frac{|S_{\text{right}}|}{|S|} Gini(S_{\text{right}}) \tag{4}$$

where $|S|$ represents the total number of samples before splitting, and $|S_{\text{left}}|, |S_{\text{right}}|$ represent the sizes of the resulting subsets.

The reduction in impurity, known as **Gini Gain**, is computed as:

$$Gini\ Gain = Gini(S) - Gini(S_{\text{split}}). \tag{5}$$

A higher Gini Gain suggests a better split.

The Gini Index is chosen for decision tree learning due to the following advantages:

- Since the Gini Index avoids logarithmic computations, it is faster to evaluate than entropy.
- The Gini Index directly measures the probability of misclassification, making it an intuitive choice for binary classifiers.
- Unlike entropy, Gini is less sensitive to rare class occurrences.

3.3.3 Misclassification Error

Misclassification Error is a measure of impurity that evaluates the quality of a split by quantifying the proportion of samples in a node that would be misclassified if the node were labeled with its majority class.

The misclassification error for a dataset S is defined as:

$$\text{Error}(S) = 1 - \max_{i=1}^{C} p_i,$$

where:

- p_i is the proportion of samples belonging to class i in the dataset S,
- C is the total number of classes.

During the decision tree construction, the dataset S is split into two subsets, S_{left} and S_{right}, based on a feature X_j and a threshold t. The overall misclassification error of the split is then computed as the weighted average of the misclassification errors of the resulting subsets:

$$\text{Error}_{\text{split}} = \frac{|S_{\text{left}}|}{|S|} \text{Error}(S_{\text{left}}) + \frac{|S_{\text{right}}|}{|S|} \text{Error}(S_{\text{right}}),$$

where $|S|$ represents the total number of samples, and $|S_{\text{left}}|$ and $|S_{\text{right}}|$ denote the sizes of the left and right subsets, respectively. A split that results in a lower misclassification error is considered better, as it implies a higher purity in the child nodes.

While misclassification error is simple and computationally efficient, its straightforward approach comes with some limitations compared to other criteria such as the Gini Index and entropy:

- Misclassification error is easy to understand and implement because it only focuses on the majority class in a node. This simplicity makes it an attractive choice for a quick assessment of node purity.
- Its computation avoids the more complex operations (such as squaring probabilities or calculating logarithms) required by the Gini Index and entropy, which can be beneficial when processing large datasets.

- Unlike the Gini Index, which considers the squared probabilities of all classes, and entropy, which accounts for the overall uncertainty in class distribution, misclassification error only reflects the error rate of the most frequent class. This can result in a less sensitive measure of impurity, particularly when the class distributions in a node are nuanced or nearly balanced.

while the misclassification error provided fast and easily interpretable results, it sometimes led to suboptimal splits compared to the Gini Index. The Gini Index and entropy, by incorporating information about the entire class distribution, were often better at detecting subtle improvements in node purity, resulting in decision trees with improved predictive performance.

Overall, misclassification error serves as a complementary tool in decision tree learning.

3.4 Stopping Criteria

Stopping criteria play a crucial role in the construction of decision trees by determining when to halt further node splitting. Without such criteria, a tree may grow excessively, leading to overfitting—where it memorizes the training data rather than learning generalizable patterns. In this section, two commonly used stopping criteria are discussed: **Maximum Depth** and **Minimum Impurity Decrease**.

3.4.1 Maximum Depth

The maximum depth stopping criterion limits the number of recursive splits applied during tree construction. Given a decision tree T, its depth $d(T)$ is defined as the length of the longest path from the root node to a leaf node:

$$d(T) = \max_{n \in \mathcal{L}} \text{depth}(n) \tag{6}$$

where $\mathcal{L}$ represents the set of leaf nodes in the tree. When $d(T)$ reaches a predefined limit $d_{\max}$, the tree stops growing, and further splits are disallowed.

Limiting the depth of the tree serves as an effective regularization technique; by preventing excessive depth ensures the tree does not perfectly fit the training set while failing on unseen data. The time complexity of constructing a decision tree is approximately $O(Nd)$, where N is the number of samples, and d is the depth. Controlling d significantly improves efficiency. However, choosing an overly small $d_{\max}$ may lead to underfitting, where the tree is too simple to capture the underlying structure of the data.

3.4.2 Minimum Impurity Decrease

Minimum impurity decrease ensures that a node is split only if it results in a sufficient reduction in impurity. Given an impurity function $I(S)$, a split of dataset S into subsets S_{left} and S_{right} results in an impurity reduction computed as:

$$\Delta I = I(S) - \left(\frac{|S_{\text{left}}|}{|S|} I(S_{\text{left}}) + \frac{|S_{\text{right}}|}{|S|} I(S_{\text{right}}) \right) \tag{7}$$

where:

- $I(S)$ is the impurity of the current node,
- $I(S_{\text{left}})$ and $I(S_{\text{right}})$ are the impurities of the two resulting child nodes,
- $|S_{\text{left}}|$ and $|S_{\text{right}}|$ are the sizes of the respective subsets.

A split is accepted only if:

$$\Delta I \geq \gamma \tag{8}$$

where γ is the minimum impurity decrease threshold.

Unlike maximum depth, which sets a hard constraint on growth, minimum impurity decrease dynamically prevents unnecessary expansion based on data characteristics. Without this constraint, the tree may create nodes that do not meaningfully contribute to classification. Therefore, by stopping weak splits, the tree remains more generalized rather than adapting to noise in the training set.

4 Results

4.1 Hyperparameter Tuning

Grid search was conducted to identify optimal parameters among the combination of the following hyperparameters:

Splitting criteria: Gini index, entropy, misclassification error.
Maximum depth: Values of 3, 5, 7, 10.
Minimum impurity decrease: 0.001, 0.01, 0.1.
Minimum sample leaf: 1, 5, 10
Cost-complexity pruning :0.0, 0.01, 0.1
Pruning: True, False
Each combination of parameters was evaluated on a validation split using the 0-1 loss metric

Validation 0-1 Loss	Training 0-1 Loss	Criterion	Max Depth	Min Samples Leaf	CCP Alpha	Prun
0.0946	0.0784	Gini	10	1	0.1	True
0.2181	0.2159	Entropy	7	1	0.0	False
0.2371	0.2495	Misclassification	5	5	0.0	False
0.2380	0.2527	Gini	5	1	0.0	False
0.2380	0.2527	Gini	5	1	0.01	True
0.3645	0.3602	Entropy	3	5	0.0	False

Table 2: Results for All Configurations (Sorted by 0-1 Loss)

4.2 Hyperparameter results

The results, summarized in Table 2, demonstrate the impact of different hyperparameters on classification performance. The best-performing configuration, using the Gini criterion with `max_depth` = 10 and `cpp_alpha` = 0.1, achieved the lowest validation 0-1 loss (0.0946) and a comparable training loss (0.0784), indicating good generalization. Conversely, configurations with shallower trees `max_depth` = 3 or `max_depth`=5 exhibited higher validation losses, suggesting underfitting. For instance, Configuration 6, using `max_depth` = 3 and `_min_samples_leaf` = 5, resulted in a validation 0-1 loss of 0.3645, the highest among tested configurations. Additionally, the test set evaluation (Table 5) confirmed that pruning did not significantly alter performance, as both pruned and unpruned models achieved an accuracy of 90.28 with a 0-1 loss of 0.0972. The best combination of parameters is found when **{'criterion': 'gini', 'max_depth': 10, 'min_samples_leaf': 1, 'min_impurity_decrease': 0.0, 'ccp_alpha': 0.1, 'prune': True}** as summarized in Table 3

Parameter	Value
Criterion	Gini
Max Depth	10
Min Samples Leaf	1
Min Impurity Decrease	0.0

Table 3: Best parameters for the model

4.3 Overfitting Analysis

A hypothesis h is said to overfit the training data if there is another hypothesis h, such that h has a smaller error than h on the training data but h has larger error on the test data than h. In other words, a decision tree overfits the training data when its accuracy on the training data goes up but its accuracy on unseen data goes down. Sevaral may be the reason for overfitting: Too much variance in the training data such that data is not a representative sample of the instance space and we split on irrelevant features. Too much noise in the training data: incorrect featured or class labels In both cases, we want to minimize the empirical error when we learn, we can do so with decision trees.

In order to avoid overfitting with decision trees, we want to prune the decision tree: remove leaves and assign the majority label of the parent to all items. We can do this in two ways. Pre-pruning refers to when we stop growing the tree at some point during construction when it is determined that there is insufficient data to make reliable decisions. Post-pruning refers to the process of growing the full decision tree and removing nodes for which we have insufficient evidence One mechanism for doing so is to prune the children of S if all children are leaves and the accuracy on the validation set does not decrease if we assign the most frequent class label to all items at S. There are various methods for evaluating which subtrees to prune.

Metric	**Without Pruning**	**With Pruning**
Accuracy	0.9028	0.9028
Zero-One Loss	0.0972	0.0972
Precision	0.9413	0.9413
Recall	0.8828	0.8828
F1 Score	0.9111	0.9111

Table 4: Evaluation Results: Test Set Metrics (With and Without Pruning)

4.4 Overcoming Overfitting

To mitigate overfitting, several strategies were implemented, including restricting tree depth, introducing pruning, and setting constraints on node splits. Notably, pruning was employed with the cost-complexity pruning parameter (ccp-alpha). As observed in Table 2, the configuration with ccp-alpha = 0.1 and pruning enabled yielded the lowest validation 0-1 loss (**0.0946**), indicating effective regularization. Additionally, increasing the minimum number of samples required for a split (min-samples-leaf) was explored, though models with min-samples-leaf=1 tended to overfit, as seen in the higher discrepancy between training and validation loss in some configurations in Table 2.

Metric	Value
Accuracy	0.9028
Zero-One Loss	0.0972
Precision	0.9413
Recall	0.8828
F1 Score	0.9111

Table 5: Test set metrics

4.4.1 Cost-Complexity Pruning

To mitigate overfitting, I implemented in the `_prune_tree` function a cost-complexity pruning. The pruning process aims to remove branches that do not significantly contribute to reducing classification error while balancing model complexity.

The `_prune_tree` function operates recursively these steps:

1. If the current node is a leaf, the function terminates at this branch. Otherwise, the function recursively calls itself to prune both the left and right subtrees.

2. The function first evaluates the tree's predictive performance before pruning. Using the `predict` function, it classifies the dataset X, then computes the *misclassification error* (0-1 loss) as:

$$\text{error\_before} = \frac{1}{n} \sum_{i=1}^{n} 1(\hat{y}_i \neq y_i)$$

where $\hat{y}_i$ is the predicted class for sample i, and y_i is the actual class.

3. The current internal node is *converted into a leaf*, storing the most frequent class in its subset. This is achieved by setting both child nodes (`left` and `right`) to `None` and updating the node's value to:

$$\arg\max \text{bincount}(y)$$

which selects the most common class in the subset routed to this node.

4. The newly pruned tree makes predictions on X, and the misclassification error is recomputed.

5. The function computes the *cost-complexity gain* as:

$$\text{gain} = \text{error\_after} - \text{error\_before} + \alpha$$

where α (stored as `self.ccp_alpha`) is the complexity parameter controlling the trade-off between model size and accuracy. If the gain is positive (indicating pruning worsens accuracy), the pruning is *reverted*, restoring the original node structure.

The `calculate_zero_one_loss` function computes the *misclassification error*, which serves as a performance metric for pruning:

$$\text{Loss} = \frac{1}{n} \sum_{i=1}^{n} 1(\hat{y}_i \neq y_i)$$

where n is the number of samples.

4.5 Feature importance

Feature importance was assessed to determine which attributes contributed most significantly to the classification decision. The five most influential features are reported in Table 6, with Feature 12 having the highest importance (0.1784)

Feature	Importance
Feature 12	0.1784
Feature 7	0.1487
Feature 14	0.1162
Feature 9	0.1144
Feature 22	0.0972

Table 6: Top 5 most important features