UNIVERSITÀ DEGLI STUDI DI MILANO
FACULTY OF SCIENCE AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

Master in
Computer Science

STATISTICAL METHODS FOR MACHINE LEARNING
TREE PREDICTORS FOR BINARY CLASSIFICATION

Professor: Nicolò Cesa-Bianchi

Student: Matteo Rigat
Matr. Nr. 13888A

Academic year 2023-2024

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

Chapter 1

Introduction

This project focuses on implementing and experimenting with decision tree predictors for binary classification, specifically to determine whether mushrooms are poisonous based on various features.

The development of the project begins with a thorough exploration of the dataset and the establishment of an effective data preprocessing strategy, including the division into training and test sets. The core of the project involves building tree predictors from scratch, utilizing single-feature binary tests at each internal node. This involves creating a basic node structure and a tree predictor class capable of training on the given dataset and evaluating performance on unseen data.

Three different criteria for leaf expansion, including the Gini index and scaled entropy, will be implemented and compared. Additionally, two stopping criteria, such as maximum tree depth and minimum impurity decrease, will be utilized to halt the construction of the decision trees. These criteria are essential for controlling the complexity of the models and preventing underfitting and overfitting.

Hyperparameter tuning will be performed to optimize the tree predictors, focusing on the splitting and stopping criteria. This step is critical for enhancing the performance and generalization capabilities of the models. The final evaluations will include the training error for each tree predictor based on the 0-1 loss, providing insights into their effectiveness.

Chapter 2

Dataset

The dataset used in this project is the Secondary Mushroom Dataset. This 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).

The dataset includes 21 features that describe various physical attributes of the mushrooms, such as cap shape, cap surface, cap color, and others. These features are encoded as categorical or numerical values. The target variable, 'class', indicates whether the mushroom is edible or poisonous.

Table 1: Sample of Mushroom Dataset

class	cap-diameter	cap-shape	...	spore-print-color	habitat	season
p	7.27	b	...	NaN	d	a
e	7.80	x	...	NaN	d	a
e	4.21	s	...	g	l	u
e	4.32	x	...	NaN	d	w
p	4.05	f	...	NaN	m	a

Nan values are exclusively found among categorical features

2.1 Data Preprocessing

To prepare the dataset for modeling, several preprocessing steps were performed. This includes encoding categorical features and splitting the dataset into training and testing sets. The following subsections detail the preprocessing steps.

Categorical Encoding

Since the dataset consists also of categorical features, one-hot encoding was applied to transform these features into a format suitable for machine learning algorithms. This encoding converts each categorical feature into a set of binary columns.

```
data_encoded = pd.get_dummies(data)
```

This transformation results in a dataset with 121 binary and numerical features. Note that all NaN values have been encoded as False.

```
data_encoded = data_encoded.astype(int)
```

Subsequently, the data were converted to 'int' type, where binary values were encoded as 0 and 1, and numerical features were rounded to the nearest integer.

This transformation significantly reduced the complexity of the decision tree model, resulting in substantial improvements in both performance and execution time.

Table 2: Sample of Mushroom Dataset

cap-diameter	stem-height	stem-width	...	season$_s$	season$_u$	season$_w$
7	13	18	...	0	0	0
7	5	12	...	0	0	0
4	6	47	...	0	1	0
4	5	7	...	0	0	1
4	4	3	...	0	0	0

Train-Test Split

The dataset was split into training and testing sets using an 80-20 split. This ensures that the model is evaluated on unseen data, providing a more accurate measure of its performance.

```
X_train, X_test, y_train, y_test =  \
train_test_split(X, y, test_size=0.2, stratify=y, random_state=42)
```

2.2 Data Summary

The preprocessing steps resulted in a clean dataset that is ready for model training and evaluation. The one-hot encoding process expanded the number of features from 21 to 121, making the data suitable for machine learning algorithms. The balanced nature of the dataset (almost equal distribution of edible and poisonous instances) ensures that the model will not be biased towards any particular class.

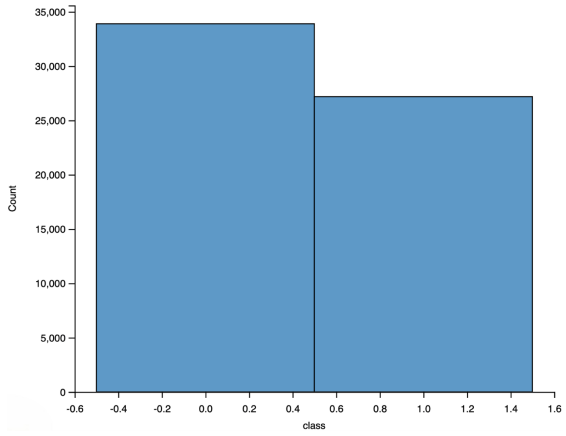

Figure 1: class_p *and* $class_e$ *distribution*

Chapter 3

Tree architecture

3.1 Node Structure

In a decision tree, each node represents a decision point based on the value of a feature. The tree is built from the root node down to the leaves.

Each node in the decision tree has the following structure:

- **Feature**: The feature based on which the split is made.
- **Threshold**: The threshold value of the feature to determine the split.
- **Left Child**: The left subtree that contains instances where the feature value is less than or equal to the threshold.
- **Right Child**: The right subtree that contains instances where the feature value is greater than the threshold.
- **Value (optional)**: If the node is a leaf, this represents the class label. If the node is not a leaf, this is set to None.

3.2 Node Creation

Nodes are created during the tree-building process. For each node, the best feature and threshold are selected based on a criterion (e.g., Gini impurity, scaled entropy) that measures the quality of the split. The dataset is then split into left and right subsets, and the process is recursively repeated for each child node.

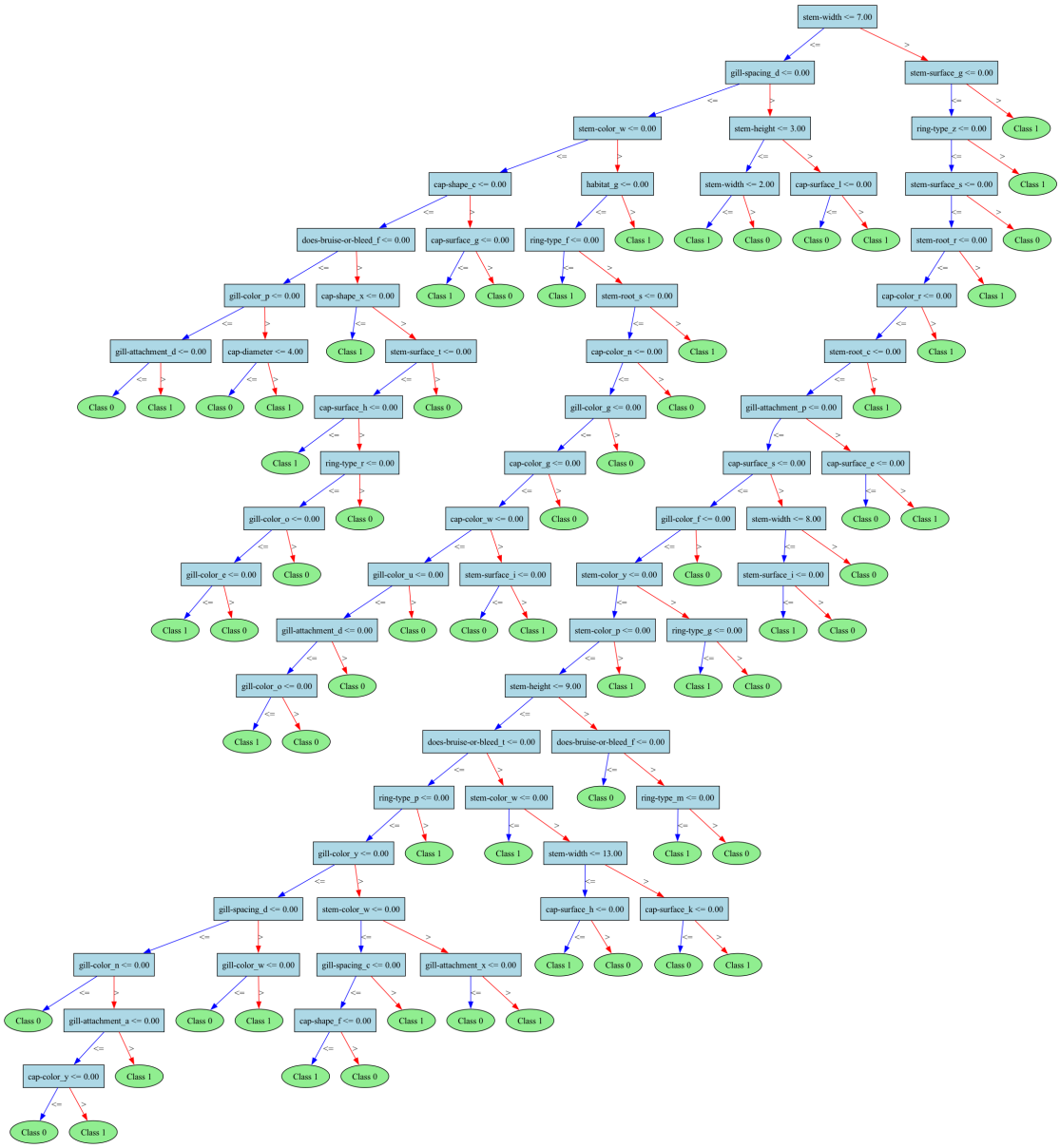

Figure 2: Example of binary tree

3.3 Tree implementation

The 'DecisionTree' procedure initializes parameters and defines the 'fit' method to start growing the tree using the grow-tree function.

```
Procedure DecisionTree:
    Initialize max_depth, max_leaf_nodes, split_function,
   min_samples_split, root, feature_names, leaf_count, depth

    Procedure fit(X, y):
        root = grow_tree(X, y)
```

The 'grow-tree' function recursively constructs the decision tree nodes based on specified criteria until termination conditions are met or the tree is fully grown.

Here is where the stopping criteria are implemented: specifically, the max_depth, max_leaf_nodes, and entropy_threshold conditions halt the tree's growth if any one of these conditions is met.

```
Procedure grow_tree(X, y, depth):
    If     max_depth is reached
        OR max_leaf_nodes is reached
        OR entropy_threshold is reached
        OR num_samples < min_samples_split
        OR all labels are the same
    Then
        Return a new leaf node with the most common label in y
    End If

    Select the best feature and threshold to split on using
        _best_criteria method

    If no best feature (gain is 0 for all features) Then
        Return a new leaf node with the most common label in y
    End If

    Split the dataset into left and right subsets using _split
        method
    Create a new internal node with the best feature and threshold

    Recursively grow the left and right child of the internal node

    Return the internal node
```

The ‘best-criteria‘ function identifies the optimal feature and threshold to split the dataset based on the gain criterion, iterating through potential splits.

```
Procedure _best_criteria(X, y, feat_idxs):
    Initialize best_gain to -1
    Initialize split_idx and split_thresh to None

    For each feat_idx in feat_idxs:
        Get the column of X corresponding to feat_idx
        Get the unique values in this column as thresholds
        For each threshold in thresholds:
            Compute the gain using _gain method with y, the column
                of X, threshold, and the split_function
            If the computed gain is greater than best_gain:
                Update best_gain to the computed gain
                Update split_idx to feat_idx
                Update split_thresh to threshold

    Return split_idx, split_thresh
```

The ‘split‘ function divides a column of data into two subsets based on a specified threshold, returning indices for the left and right subsets.

```
Procedure _split(X_column, split_thresh):
    Get the indices of X_column where the value is less than or
        equal to split_thresh as left_idxs

    Get the indices of X_column where the value is greater than
        split_thresh as right_idxs

    Return left_idxs, right_idxs
```

The ‘gain‘ function calculates the impurity or other specified criterion when splitting data into left and right subsets, evaluating the quality of a potential split.

```
Procedure _gain(y, X_column, split_thresh, criterion):
    Get the function corresponding to the criterion from a
        dictionary of functions

    Compute the parent_criterion using the function obtained above
        with y

    Get left_idxs and right_idxs by calling _split method with
        X_column and split_thresh

    If either left_idxs or right_idxs is empty:
        Return 0

    Get y_left and y_right using left_idxs and right_idxs from y

    Compute child_criterion using _weighted_criterion method with
        y_left, y_right, and the function obtained above

    Compute gain as the difference between parent_criterion and
        child_criterion

    Return gain
```

3.4 Split criteria

Scaled Entropy

The Scaled Entropy criterion is a measure of impurity that calculates the entropy of the class distribution in a node and scales it.

$$scaled\_ent = -\frac{p}{2}\log_2 p \quad - \quad \frac{1-p}{2}\log_2(1-p)$$

```
def _scaled_entropy(self, y):
    hist = np.bincount(y)
    probs = hist / len(y)
    scaled_ent = -np.sum([(p / 2) * np.log2(p) for p in probs if
        p > 0])
    return scaled_ent
```

Gini Impurity

Gini Impurity measures the impurity of a node by quantifying the likelihood of incorrect classification based on the class distribution within the node. It represents the probability that any given element would be misclassified if it were assigned a class label according to the observed frequencies of the classes in the node.

$$gini\_funct = 2p \cdot (1 - p)$$

```
def _gini_impurity(self, y):
    hist = np.bincount(y)
    probs = hist / len(y)
    gini = 1.0 - np.sum(probs ** 2)
    return gini
```

Squared Impurity

The Squared Impurity criterion is a variation of the Gini impurity specifically designed for binary classification tasks. It calculates the square root of the product of probabilities of the class and its complement for each class label.

$$squared = \sqrt{p \cdot (1 - p)}$$

```
def _squared_impurity(self, y):
    hist = np.bincount(y)
    probs = hist / len(y)
    epsilon = 1e-10  # Small constant to avoid multiplying by zero
    sqr = np.sum(np.sqrt((probs + epsilon) * (1 - probs + epsilon)
    ))
    return sqr
```

Chapter 4

Hyperparameter Tuning

Hyperparameters refer to parameters that are set before training a model and cannot be directly learned from the data during training. Tuning these hyperparameters is crucial to improving the performance of decision tree models. This process involves evaluating models with different hyperparameter settings and selecting the configuration that yields the best results. To conserve computational resources and expedite computations, it's common practice to conduct hyperparameter tuning on a substantial subset of the dataset. In this study, the entire dataset was used for these operations because the marginal increase in training time did not justify the loss of accuracy observed.

K-fold cross validation is a validation technique that consists in dividing the dataset into K subsets, also called folds. Iteratively the model is trained K times, using a different training set and test set in each iteration. In particular each time a fold is chosen, which will be used as a test set, while the remaining part of the dataset will be used for training. Using K-fold cross-validation ensures that the evaluation of the decision tree model's performance is robust and less sensitive to the specific partitioning of the dataset. It provides a more accurate estimation of how well the model will generalize to new, unseen data. Specifically, a 5-fold cross validation was used in this project.

In summary, hyperparameter tuning and K-fold cross-validation are essential techniques in optimizing and evaluating decision tree models for binary classification tasks, ensuring both effectiveness in model selection and reliability in performance assessment.

To perform the hyperparameter tuning the following function was implemented

```
Procedure grid_search(X_train, y_train, param_grid, scoring_func):
    Initialize best_score to infinity
    Initialize best_params to None
    Initialize results to an empty list

    Create a KFold object for cross-validation

    Procedure evaluate_params(params):
        Initialize current_scores as empty list

        For each train_idxs, val_idxs in the KFold splits of
            X_train and y_train:

            Split the training data into training and validation
                sets using train_idxs and val_idxs

            Create a DecisionTree model with the given params

            Fit the model to the training data

            Predict on the validation data

            Compute the score and append it to current_scores

        Compute the mean of current_scores

        Append the params, mean_score to results

        Return params and mean_score

    Sort the results based on the score

    Return results, best_params, and best_score
End Procedure
```

Combining hyperparameter tuning with K-fold cross-validation is crucial for optimizing model performance and ensuring robustness.

Hyperparameter tuning allows us to systematically explore different configurations to identify the set of parameters that best suit our specific dataset and problem domain. This process helps in enhancing the model's predictive accuracy and generalizability.

On the other hand, K-fold cross-validation provides a rigorous method to assess the model's performance by repeatedly splitting the data into training and validation sets. By averaging the results over multiple folds, K-fold cross-validation offers a more reliable estimate of the model's performance compared to a single train-test split.

4.1 Zero-one loss

The metric used in this project to evaluate the training process is the zero-one loss. This is one of the simplest metrics: it gives 0 if the classification is correct and 1 otherwise.

$$\text{zero\_one\_loss} = \frac{1}{n}\sum_{i=1}^{n} \mathbb{I}(y_i \neq y_i^{\text{pred}})$$

```
def zero_one_loss(y_true, y_pred):
    return np.mean(y_pred != y_true)
```

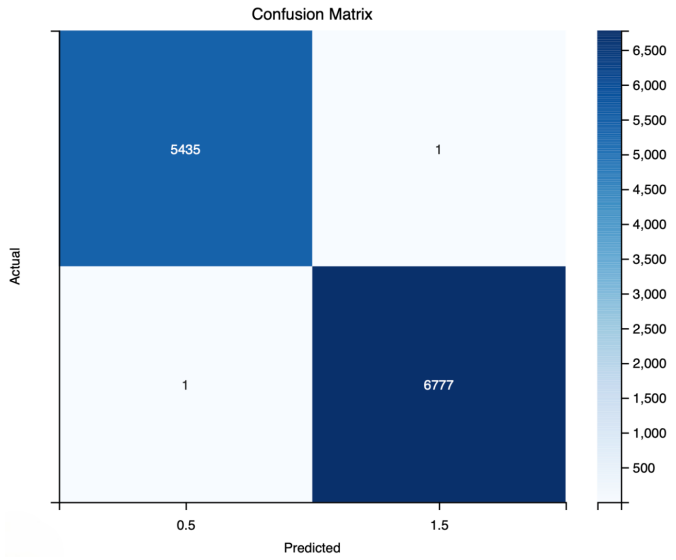

Figure 3: Example of confusion Matrix on the test set

Chapter 5

Experiments

In this chapter several experiments will be reported based on the techniques presented in previous chapters. Initially, the capabilities of the models was evaluated using a simple training technique, while subsequently, 5-fold cross validation is used to verify the robustness of the best models.

5.1 Stop criteria

max_depth

The depth of the tree grows until it reaches an average depth that depends on the splitting criteria, typically around 28 to 35. Even if we set a maximum depth that is potentially very large, the tree usually stops growing much earlier.

```
Total number of combinations: 20  x  5 cv = 100 iterations

zero one loss: 0.16414 with params: {'max_depth': 10, 'split_function': 'gini'}      mean depth: 10.0 and mean leafs: 65.4
zero one loss: 0.14447 with params: {'max_depth': 11, 'split_function': 'gini'}      mean depth: 11.0 and mean leafs: 75.2
zero one loss: 0.12777 with params: {'max_depth': 12, 'split_function': 'gini'}      mean depth: 12.0 and mean leafs: 82.0
zero one loss: 0.10857 with params: {'max_depth': 13, 'split_function': 'gini'}      mean depth: 13.0 and mean leafs: 88.4
zero one loss: 0.08163 with params: {'max_depth': 14, 'split_function': 'gini'}      mean depth: 14.0 and mean leafs: 96.0
zero one loss: 0.00143 with params: {'max_depth': 25, 'split_function': 'gini'}      mean depth: 25.0 and mean leafs: 165.0
zero one loss: 0.00115 with params: {'max_depth': 26, 'split_function': 'gini'}      mean depth: 26.0 and mean leafs: 166.4
zero one loss: 0.00098 with params: {'max_depth': 27, 'split_function': 'gini'}      mean depth: 26.6 and mean leafs: 167.6
zero one loss: 0.00096 with params: {'max_depth': 28, 'split_function': 'gini'}      mean depth: 27.2 and mean leafs: 168.6
zero one loss: 0.00082 with params: {'max_depth': 29, 'split_function': 'gini'}      mean depth: 27.6 and mean leafs: 168.8
zero one loss: 0.00094 with params: {'max_depth': 30, 'split_function': 'gini'}      mean depth: 27.8 and mean leafs: 169.4
zero one loss: 0.00088 with params: {'max_depth': 31, 'split_function': 'gini'}      mean depth: 28.0 and mean leafs: 169.4
zero one loss: 0.00090 with params: {'max_depth': 32, 'split_function': 'gini'}      mean depth: 28.0 and mean leafs: 170.0
zero one loss: 0.00086 with params: {'max_depth': 33, 'split_function': 'gini'}      mean depth: 28.0 and mean leafs: 169.2
zero one loss: 0.00090 with params: {'max_depth': 34, 'split_function': 'gini'}      mean depth: 28.0 and mean leafs: 169.8
zero one loss: 0.00086 with params: {'max_depth': 95, 'split_function': 'gini'}      mean depth: 28.0 and mean leafs: 169.4
zero one loss: 0.00088 with params: {'max_depth': 96, 'split_function': 'gini'}      mean depth: 28.0 and mean leafs: 169.6
zero one loss: 0.00086 with params: {'max_depth': 97, 'split_function': 'gini'}      mean depth: 28.0 and mean leafs: 169.6
zero one loss: 0.00092 with params: {'max_depth': 98, 'split_function': 'gini'}      mean depth: 28.0 and mean leafs: 169.6
zero one loss: 0.00082 with params: {'max_depth': 99, 'split_function': 'gini'}      mean depth: 28.0 and mean leafs: 169.8
```

max_leaf_nodes

Using the max_leaf_nodes stopping criterion, a very similar phenomenon occurs. Depending on the split criteria, the tree stops growing when it reaches between 160 and 180 leaves. It is interesting to note that this often happens when the tree reaches the depth of approximately 30, as previously mentioned.

```
Total number of combinations: 25  x  5 cv = 125 iterations

zero one loss: 0.31209 with params: {'max_leaf_nodes': 45, 'split_function': 'gini'}     mean depth: 17.2 and mean leafs: 45.0
zero one loss: 0.31155 with params: {'max_leaf_nodes': 46, 'split_function': 'gini'}     mean depth: 17.2 and mean leafs: 46.0
zero one loss: 0.30564 with params: {'max_leaf_nodes': 47, 'split_function': 'gini'}     mean depth: 17.2 and mean leafs: 47.0
zero one loss: 0.30525 with params: {'max_leaf_nodes': 48, 'split_function': 'gini'}     mean depth: 17.2 and mean leafs: 48.0
zero one loss: 0.30187 with params: {'max_leaf_nodes': 49, 'split_function': 'gini'}     mean depth: 17.2 and mean leafs: 49.0
zero one loss: 0.00317 with params: {'max_leaf_nodes': 165, 'split_function': 'gini'}    mean depth: 28.0 and mean leafs: 163.6
zero one loss: 0.00327 with params: {'max_leaf_nodes': 166, 'split_function': 'gini'}    mean depth: 28.0 and mean leafs: 164.0
zero one loss: 0.00301 with params: {'max_leaf_nodes': 167, 'split_function': 'gini'}    mean depth: 28.0 and mean leafs: 164.6
zero one loss: 0.00303 with params: {'max_leaf_nodes': 168, 'split_function': 'gini'}    mean depth: 28.0 and mean leafs: 165.2
zero one loss: 0.00287 with params: {'max_leaf_nodes': 169, 'split_function': 'gini'}    mean depth: 28.0 and mean leafs: 165.8
zero one loss: 0.00231 with params: {'max_leaf_nodes': 170, 'split_function': 'gini'}    mean depth: 28.0 and mean leafs: 166.6
zero one loss: 0.00229 with params: {'max_leaf_nodes': 171, 'split_function': 'gini'}    mean depth: 28.0 and mean leafs: 167.0
zero one loss: 0.00229 with params: {'max_leaf_nodes': 172, 'split_function': 'gini'}    mean depth: 28.0 and mean leafs: 167.2
zero one loss: 0.00223 with params: {'max_leaf_nodes': 173, 'split_function': 'gini'}    mean depth: 28.0 and mean leafs: 168.0
zero one loss: 0.00172 with params: {'max_leaf_nodes': 174, 'split_function': 'gini'}    mean depth: 28.0 and mean leafs: 168.0
zero one loss: 0.00133 with params: {'max_leaf_nodes': 175, 'split_function': 'gini'}    mean depth: 28.0 and mean leafs: 168.6
zero one loss: 0.00094 with params: {'max_leaf_nodes': 176, 'split_function': 'gini'}    mean depth: 28.0 and mean leafs: 168.6
zero one loss: 0.00104 with params: {'max_leaf_nodes': 177, 'split_function': 'gini'}    mean depth: 28.0 and mean leafs: 169.0
zero one loss: 0.00084 with params: {'max_leaf_nodes': 178, 'split_function': 'gini'}    mean depth: 28.0 and mean leafs: 169.6
zero one loss: 0.00086 with params: {'max_leaf_nodes': 179, 'split_function': 'gini'}    mean depth: 28.0 and mean leafs: 169.8
zero one loss: 0.00086 with params: {'max_leaf_nodes': 595, 'split_function': 'gini'}    mean depth: 28.0 and mean leafs: 169.6
zero one loss: 0.00094 with params: {'max_leaf_nodes': 596, 'split_function': 'gini'}    mean depth: 28.0 and mean leafs: 169.6
zero one loss: 0.00084 with params: {'max_leaf_nodes': 597, 'split_function': 'gini'}    mean depth: 28.0 and mean leafs: 170.0
zero one loss: 0.00096 with params: {'max_leaf_nodes': 598, 'split_function': 'gini'}    mean depth: 28.0 and mean leafs: 169.4
zero one loss: 0.00096 with params: {'max_leaf_nodes': 599, 'split_function': 'gini'}    mean depth: 28.0 and mean leafs: 169.8
```

entropy/impurity threshold

Using the entropy/impurity threshold, we observed consistent results across all splitting criteria, notably with identical values for tree depth and maximum leaf nodes as previously discussed.

```
Total number of combinations: 18  x  5 cv = 90 iterations

zero one loss: 0.00084 with params: {'entropy_threshold': 0.0001, 'split_function': 'scaled_entropy'}    mean depth: 27.2 and mean leafs: 166.2
zero one loss: 0.00080 with params: {'entropy_threshold': 0.0001, 'split_function': 'gini'}      mean depth: 27.0 and mean leafs: 169.8
zero one loss: 0.00094 with params: {'entropy_threshold': 0.0001, 'split_function': 'squared'}   mean depth: 33.0 and mean leafs: 156.0
zero one loss: 0.00090 with params: {'entropy_threshold': 0.001, 'split_function': 'scaled_entropy'}     mean depth: 27.0 and mean leafs: 165.8
zero one loss: 0.00090 with params: {'entropy_threshold': 0.001, 'split_function': 'gini'}   mean depth: 26.4 and mean leafs: 165.8
zero one loss: 0.00096 with params: {'entropy_threshold': 0.001, 'split_function': 'squared'}    mean depth: 33.0 and mean leafs: 155.8
zero one loss: 0.00098 with params: {'entropy_threshold': 0.01, 'split_function': 'scaled_entropy'}      mean depth: 27.0 and mean leafs: 153.6
zero one loss: 0.00186 with params: {'entropy_threshold': 0.01, 'split_function': 'gini'}    mean depth: 25.8 and mean leafs: 141.6
zero one loss: 0.00088 with params: {'entropy_threshold': 0.01, 'split_function': 'squared'}     mean depth: 33.0 and mean leafs: 155.4
zero one loss: 0.00831 with params: {'entropy_threshold': 0.1, 'split_function': 'scaled_entropy'}   mean depth: 25.0 and mean leafs: 109.0
zero one loss: 0.02092 with params: {'entropy_threshold': 0.1, 'split_function': 'gini'}     mean depth: 23.0 and mean leafs: 87.2
zero one loss: 0.00096 with params: {'entropy_threshold': 0.1, 'split_function': 'squared'}      mean depth: 33.0 and mean leafs: 152.0
zero one loss: 0.44509 with params: {'entropy_threshold': 0.5, 'split_function': 'scaled_entropy'}   mean depth: 0.0 and mean leafs: 0.0
zero one loss: 0.44509 with params: {'entropy_threshold': 0.5, 'split_function': 'gini'}     mean depth: 0.0 and mean leafs: 0.0
zero one loss: 0.01099 with params: {'entropy_threshold': 0.5, 'split_function': 'squared'}      mean depth: 32.2 and mean leafs: 108.6
zero one loss: 0.44509 with params: {'entropy_threshold': 1.0, 'split_function': 'scaled_entropy'}   mean depth: 0.0 and mean leafs: 0.0
zero one loss: 0.44509 with params: {'entropy_threshold': 1.0, 'split_function': 'gini'}     mean depth: 0.0 and mean leafs: 0.0
zero one loss: 0.44509 with params: {'entropy_threshold': 1.0, 'split_function': 'squared'}      mean depth: 0.0 and mean leafs: 0.0
```

5.2 Split criteria

A comparison across all splitting criteria reveals that, regardless of the stopping criteria used, we consistently achieve similar performance scores among the different splitting criteria

```
Total number of combinations: 9  x  5 cv = 45 iterations

zero one loss: 0.00082 with params: {'max_depth': 50, 'split_function': 'scaled_entropy'}    mean depth: 28.2 and mean leafs: 165.8
zero one loss: 0.00088 with params: {'max_depth': 50, 'split_function': 'gini'}      mean depth: 28.0 and mean leafs: 169.4
zero one loss: 0.00094 with params: {'max_depth': 50, 'split_function': 'squared'}   mean depth: 34.0 and mean leafs: 155.8
zero one loss: 0.00088 with params: {'max_leaf_nodes': 200, 'split_function': 'scaled_entropy'}      mean depth: 28.2 and mean leafs: 165.8
zero one loss: 0.00094 with params: {'max_leaf_nodes': 200, 'split_function': 'gini'}    mean depth: 28.0 and mean leafs: 169.6
zero one loss: 0.00094 with params: {'max_leaf_nodes': 200, 'split_function': 'squared'}     mean depth: 34.0 and mean leafs: 156.0
zero one loss: 0.00092 with params: {'entropy_threshold': 0.0001, 'split_function': 'scaled_entropy'}    mean depth: 28.2 and mean leafs: 166.0
zero one loss: 0.00086 with params: {'entropy_threshold': 0.0001, 'split_function': 'gini'}      mean depth: 28.0 and mean leafs: 169.6
zero one loss: 0.00102 with params: {'entropy_threshold': 0.0001, 'split_function': 'squared'}   mean depth: 34.0 and mean leafs: 156.4

Top 10 Results:
Rank 1: Mean zero one loss: 0.000819 with params: {'max_depth': 50, 'split_function': 'scaled_entropy'}
Rank 2: Mean zero one loss: 0.000860 with params: {'entropy_threshold': 0.0001, 'split_function': 'gini'}
Rank 3: Mean zero one loss: 0.000880 with params: {'max_depth': 50, 'split_function': 'gini'}
Rank 4: Mean zero one loss: 0.000880 with params: {'max_leaf_nodes': 200, 'split_function': 'scaled_entropy'}
Rank 5: Mean zero one loss: 0.000921 with params: {'entropy_threshold': 0.0001, 'split_function': 'scaled_entropy'}
Rank 6: Mean zero one loss: 0.000942 with params: {'max_depth': 50, 'split_function': 'squared'}
Rank 7: Mean zero one loss: 0.000942 with params: {'max_leaf_nodes': 200, 'split_function': 'gini'}
Rank 8: Mean zero one loss: 0.000942 with params: {'max_leaf_nodes': 200, 'split_function': 'squared'}
Rank 9: Mean zero one loss: 0.001023 with params: {'entropy_threshold': 0.0001, 'split_function': 'squared'}

Best score: : 0.000819
Best Hyperparameters: {'max_depth': 50, 'split_function': 'scaled_entropy'}

Train accuracy: 1.000000
Test accuracy: 0.999509
zero one loss on test set with best params: 0.000491
```

5.3 'int' dataset performance improvements

```
[5 rows x 21 columns]
   cap-diameter  stem-height  stem-width  ...  season_s  season_u  season_w
0         15.26        16.95       17.09  ...     False     False      True
1         16.60        17.99       18.19  ...     False      True     False
2         14.07        17.80       17.74  ...     False     False      True
3         14.17        15.77       15.98  ...     False     False      True
4         14.64        16.53       17.20  ...     False     False      True

[5 rows x 121 columns]

Execution time: 108.7676899433136 seconds

Train accuracy: 1.000000
Test accuracy: 0.998936
zero one loss on test set with best params: 0.001064
```

```
[5 rows x 21 columns]
   cap-diameter  stem-height  stem-width  ...  season_s  season_u  season_w
0            15           16          17  ...         0         0         1
1            16           17          18  ...         0         1         0
2            14           17          17  ...         0         0         1
3            14           15          15  ...         0         0         1
4            14           16          17  ...         0         0         1

[5 rows x 121 columns]

Execution time: 2.756330966949463 seconds

Train accuracy: 1.000000
Test accuracy: 0.999836
zero one loss on test set with best params: 0.000164
```

As previously discussed, transforming the entire dataset into integers has led to approximately a 10x performance improvement accompanied by a remarkable 40x reduction in execution time.

Chapter 6

Conclusion

As observed, the tree does not accumulate enough samples to attain high values of the stopping criteria. Therefore, techniques such as tree pruning are not applicable here, as the tree does not grow excessively to require pruning.

Moreover, the model consistently avoids overfitting regardless of the stopping or splitting criteria employed. This is evident from the minimal difference between training and test errors, indicating good generalization.

The primary factor influencing the tree's predictive performance is the dataset size. Larger datasets tend to yield higher scores, suggesting that our current dataset is not large enough to maximize the model's potential. Consequently, achieving better scores might necessitate a larger dataset.

The current model's complexity appears to be well-balanced, avoiding both underfitting and overfitting. Future models should maintain this balance while exploring additional features and larger datasets.