Tree predictors for binary classification

MACHINE LEARNING PROJECT

Professor Nicolò Cesa-Bianchi

Elaheh Zohdi - DSE
13731A

University of Milan

1. Introduction

Decision trees are powerful tools for both regression and classification tasks due to their simplicity, interpretability, and capability to handle both categorical and numerical data. In this project, I have implemented a decision tree from scratch to classify mushrooms into two categories: **poisonous (p)** and **edible (e)**. The main objective was to build a tree predictor that uses single-feature binary tests at each internal node, adopting Gini, entropy and misclassification as the decision criteria, and considering thresholds for numerical features and membership tests for categorical features.

The dataset used for this project is the **Mushroom Dataset**, which provides a set of features describing various characteristics of mushrooms, including their shape, color, size, and other important details. The task is to use these features to predict whether a mushroom is poisonous or edible.

Project Objectives:

1. **Implementation of a tree predictor** from scratch using single-feature binary tests for internal node decision criteria.
2. **Incorporate at least 3 criteria for leaf expansion** and **2 stopping criteria**.
3. Train the decision tree and **compute training error using 0-1 loss**.
4. **Perform hyperparameter tuning** based on the splitting and stopping criteria.
5. Analyze and present the findings through **exploratory data analysis (EDA)** and detailed performance evaluation.

2. Dataset Overview

2.1 Data Structure

The dataset contains 61,069 rows and 21 columns, with each row representing a unique mushroom and each column representing a feature of that mushroom. The target variable (`class`) indicates whether a mushroom is **poisonous (p)** or **edible (e)**. The features include a mix of categorical and numerical variables, as I've listed below:

- Categorical Features: Cap shape, cap surface, gill attachment, and more.
- Numerical Features: Cap diameter, stem height, stem width.

Feature	**type**	**description**
class	Categorical	Poisonous (p) or Edible (e)
cap-diameter	Numerical – continuous	Cap diameter in cm
cap-shape	Categorical	Shape of the mushroom cap
Cap-surface	Categorical	Surface of the cap
Cap-color	Categorical	
does-bruise-or-bleed	Categorical	Does the mushroom bruise or bleed?
gill-attachment	Categorical	

gill-spacing	Categorical	
gill-color	Categorical	
stem-height	Numerical – continuous	Height of the mushroom stem
stem-width	Numerical – continuous	Width of the mushroom stem
Stem-root	Categorical	
Stem-surface	Categorical	
Stem-color	Categorical	
Veil-type	Categorical	
Has-ring	Categorical	
Ring-type	Categorical	
spore-print-color	Categorical	
habitat	Categorical	
season	Categorical	Season in which the mushroom grows

2.2 Data Preprocessing

The data preprocessing stage was critical for ensuring that the data was clean and properly formatted for the decision tree model. The following steps were performed:

2.2.1 Handling Missing Values

In order to handle the missing values, firstly I've computed the percentage of missing values in each column. First thing noticed was that no numerical column includes any missing values.

The output showed that some of the features, contained null values for more than half of their rows. For instance, the features such as **stem-root, stem-surface, veil-type, veil-color, and spore-print-colo**r, had over 50% missing values.

These features, that contain too many missing values, are quite unreliable for modeling. Columns with a high percentage of missing values introduce uncertainty and filling them with synthetic data might not reflect the true characteristics of the dataset, leading to poor model performance.

Therefore, I've decided to drop these columns and for the other features containing fewer missing values, I imputed the missing values using mode imputation (the most frequent value), which is a common technique for categorical data.

Remaining columns after dropping the missing values are: class , cap-diameter, cap-shape, cap-surface, cap-color, does-bruise-or-bleed, gill-attachment, gill-spacing, gill-color, stem-height, stem-width, stem-color, has-ring, ring-type, habitat, season.

2.2.2 Feature Encoding

Since the decision tree model cannot handle raw categorical data, the categorical features were encoded using **Label Encoding**. This method assigns a unique integer to each category:

```
from sklearn.preprocessing import LabelEncoder

encoder = LabelEncoder()
for col in categorical_cols:
    df_cleaned[col] = encoder.fit_transform(df_cleaned[col])
```

This ensured that all features were numerical before feeding them into the decision tree algorithm. There are other methods for encoding the categorical columns but *Label Encoding* in particular, is an appropriate method for encoding this data set because it **won't increase the dimensionality** of the data set, so it has a low complexity. Additionally, as label encoding it **keeps the number of features the same**, it's an ideal method for decision tree.

Decision trees **split on feature values,** and label encoding allows the tree to work effectively with categorical data. Also, there **is no assumption on ordinality** as label encoding treats each category as a distinct value. Moreover, since we are building a decision tree classifier, there is no need for standardization or linear transformation of the data because they work by splitting data based on thresholds or categories, not distances or continuous values like linear models or k-NN. Therefore, for decision trees, label encoding is typically the most suitable method for categorical variables.

2.2.3 Scaling Numerical Features

The continuous features (**cap-diameter**, **stem-height**, and **stem-width**) were scaled using the **StandardScaler** to ensure they were on the same scale. This was crucial for ensuring that the tree did not give undue importance to features based on their magnitudes alone. In decision trees, scaling is usually not necessary because the algorithm splits based on feature values. However, for other machine learning algorithms, scaling might improve performance.

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
df_cleaned[continuous_cols] =
scaler.fit_transform(df_cleaned[continuous_cols])
```

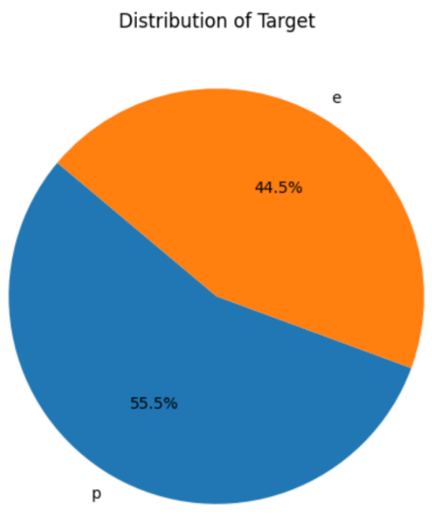
3. Exploratory Data Analysis (EDA)

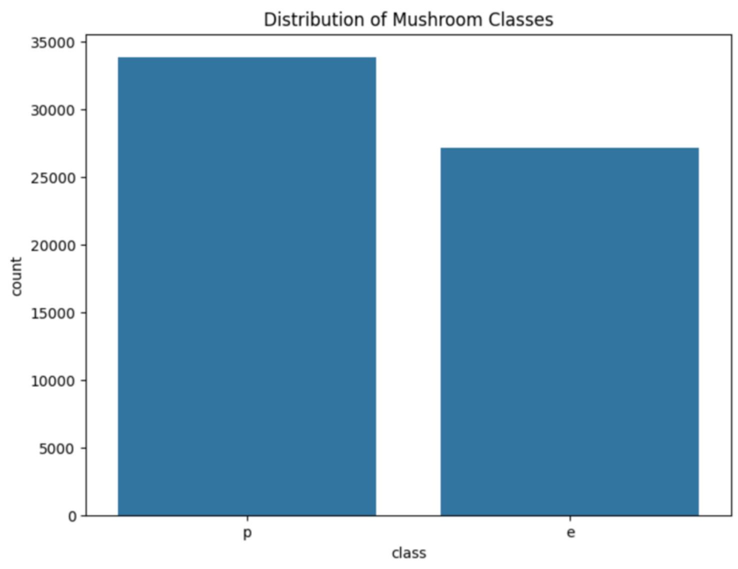
Before building the model, I performed an exploratory data analysis (EDA) to better understand the structure of the dataset, identify relationships between features, and gain insights into how the features relate to the target variable. The EDA was performed before data processing as I wanted to explore the data in the raw format but in this report only information related to the remaining features after data processing is mentioned.

3.1. Target Variable Distribution

The dataset is fairly balanced, with almost equal numbers of **poisonous** and **edible** mushrooms. This balance is important because it ensures that the decision tree model won't be biased toward one class.

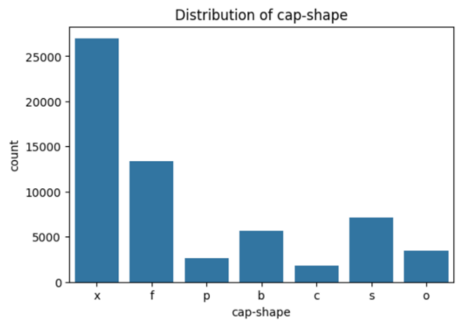
I've plotted the pie chart for finding for visualizing the percentage of each unique value in the class and histogram to see the exact count number of each unique value in the class.

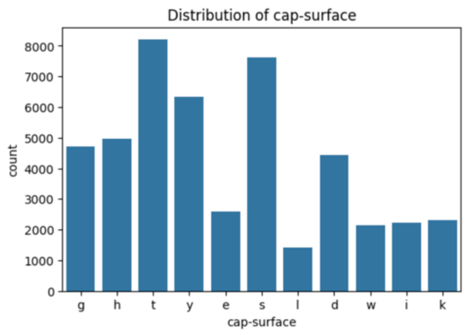


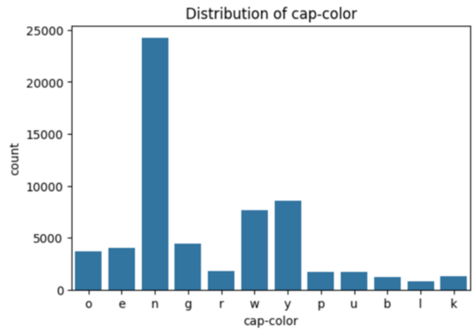


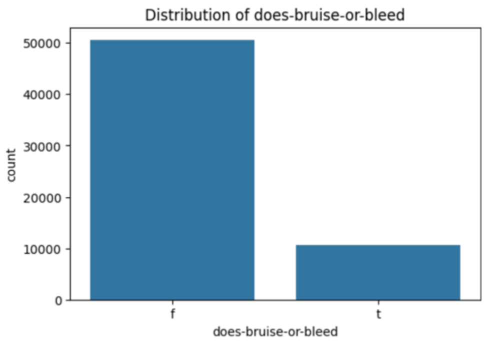
3.2. Categorical Feature Distribution

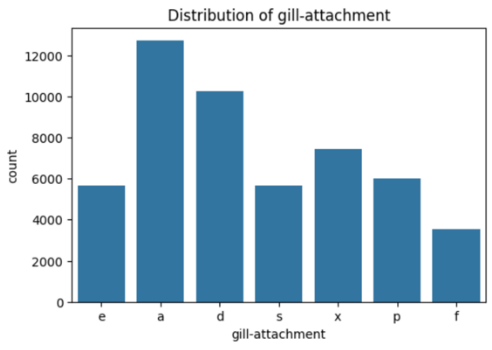
Next, I explored the distribution of key categorical features. The visualizations reveal significant variability across several key attributes like cap shape, surface, color, and gill-related features. Some categories, such as "cap color" and "gill spacing," show strong dominance of certain values, while others like "habitat" and "ring type" exhibit more balanced distributions. Based on the visualizations, features like cap surface, cap color, gill color, stem color, and ring type are more evenly distributed and likely more informative for classification. And Features such as bruising or bleeding and gill spacing are more imbalanced and might contribute less to the decision-making process.

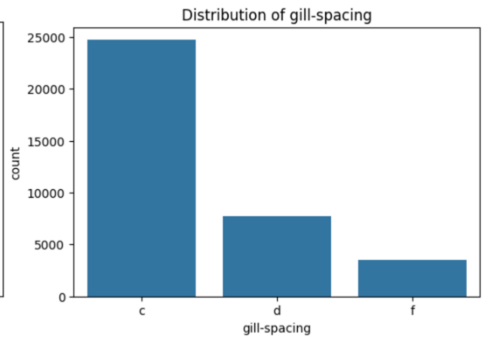


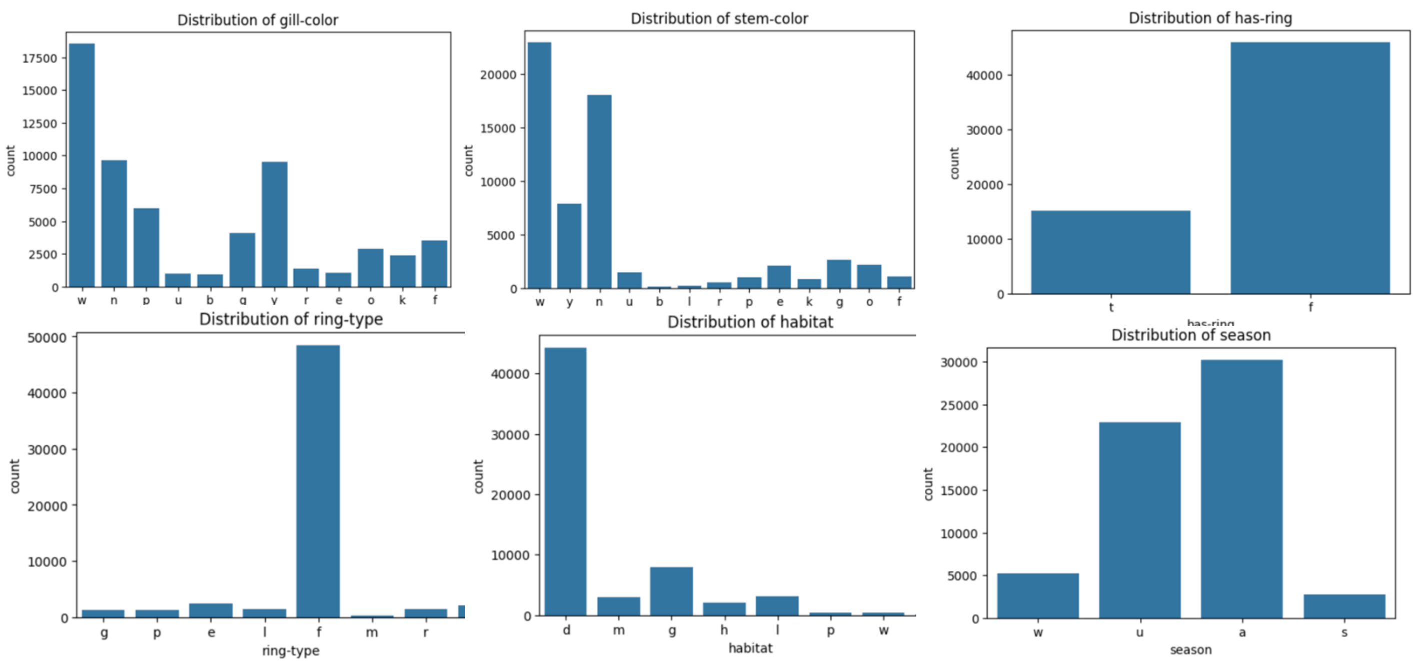




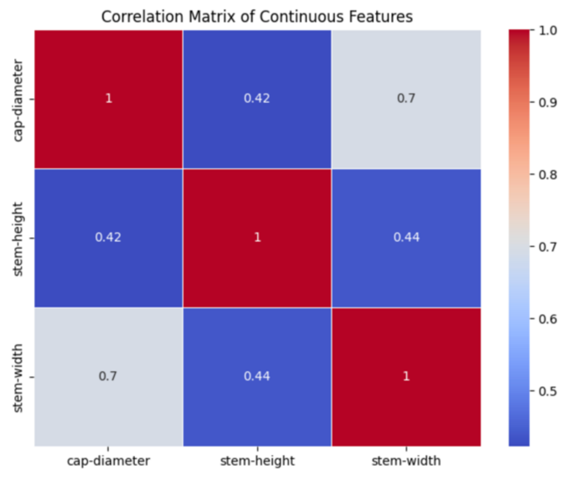








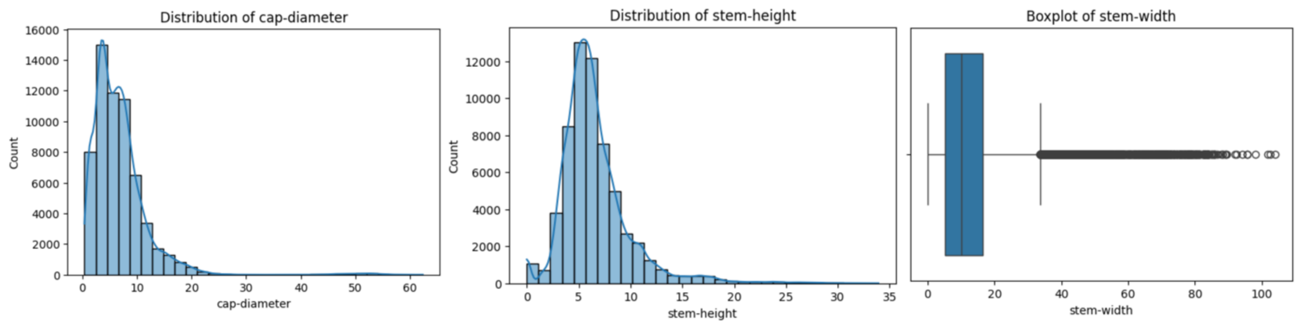
3.3. Correlation of Numerical Features



For the numerical features, I've used boxplot and also visualized the distribution of the variables. Next, I've computed the correlation matrix to check for any multicollinearity or strong relationships between the features:

```
corr = df_cleaned[continuous_cols].corr()
sns.heatmap(corr, annot=True,
cmap='coolwarm')
```

I found that **stem-height** and **stem-width** had mild positive correlation, which suggested some interaction between these two features.



4. Model Implementation

4.1. Node Class

The Node class was implemented to represent individual nodes in the decision tree. Each node in the tree contains:

- *Feature* and *threshold* for splitting the data.
- *Left* and *right* child nodes.
- A *value* if the node is a leaf node.

We know that each node may be either an internal node, with a feature index, threshold value, and reference to its left and right children. Or a leaf node, containing a classification value. Each node is initialized with children, a flag to check if it's a leaf and storing the decision criterion for splitting. The method *is_leaf_node* checks if a node is leaf by verifying whether value attribute is set or not.

```
class Node:
    def __init__(self, feature=None, threshold=None, left=None,
right=None,*,value=None):
        self.feature = feature
        self.threshold = threshold
        self.left = left
        self.right = right
        self.value = value

    def is_leaf_node(self):
        return self.value is not None
```

4.2. Decision Tree (DecisionMTree Class)

The decision tree model was implemented from scratch with the following functionalities:

- Decision Criterion: The model supports multiple criteria for splitting nodes, including **Gini impurity**, **entropy**, and **misclassification error**. These are key to selecting the best splits.
- Stopping Criteria: The tree growth stops when either:
    - **The maximum depth** is reached. *max_depth* limits how deep a tree can grow to avoid overfitting.
    - **The minimum sample** size required for a split is below a threshold. *min_samples_split* ensures that only nodes with sufficient number of samples are split.
    - **The impurity decrease** is too small. In other words, *min_impurity_decreases* add a form of pre pruning ensuring that the split is only made if the improvement in impurity (*gini or entropy*) is greater than a threshold and information gain is **not** too small.
- The model also supports random feature selection, having *n_feature* in the constructor as well which if specified, only a random set of subsets of feature is considered at each split.

The tree grows recursively by selecting the best split based on the chosen criterion and stopping when it satisfies one of the conditions.

```
class DecisionMTree:
    def __init__(self, criterion='gini', max_depth=100,
min_impurity_decrease=0, n_features=None):
        self.criterion = criterion
        self.max_depth = max_depth
        self.min_impurity_decrease = min_impurity_decrease
        self.n_features = n_features
        self.root = None
```

4.3. Fit method

fit method initializes the feature subset, and if n_feature isn't provided the number of features is set to the total number of features in the dataset. Otherwise, a subset is selected for splitting at each node. And then it initiates the tree-growing process by calling _grow_tree recursive method.

4.4. Tree grows (_grow_tree) method.

the tree growth is halted when either maximum depth is reached (`depth>= self.max_depth`) or all samples in a node belong to a single class (`n_labels == 1`) or the node contains fewer samples than the specified minimum (`n_samples < self.min_samples_split`).

Before proceeding with the split, the gain (impurity decrease) is compared with `min_impurity_decrease`. If the gain is too small, no further splits are made.

If none of the stopping criteria are met, the function continues splitting by creating left and right subtrees recursively.

4.5. Splitting Criteria

The model computes **information gain** (based on Gini, entropy, or misclassification error) to decide the best splits:

then with the _best_split() method, a random subset of features is selected feat_idxs and for each feature, potential thresholds (unique values for the feature) are evaluated. The split with the highest information gain is chosen.

4.6. Impurity calculations

With the _impurity method, depending on the chosen criterion (gini, entropy, missclassificaqtion), the corresponding method is called to calculate the impurity for a set of labels (y)

Gini impurity measures the probability of incorrectly classifying a randomly chosen element. Lower values are better.

```
    def _gini(self, y):
        hist = np.bincount(y)
        ps = hist / len(y)
        return 1 - np.sum(ps ** 2)
```

Entropy is used to measure the disorder or uncertainty in the labels. Lower entropy signifies purer splits.

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

misclassification error simply counts the fraction of samples that do not belong to the most common class in the node.

```
  def _misclassification_error(self, y):
        hist = np.bincount(y)
        ps = hist / len(y)
        return 1 - np.max(ps)
```

4.7. Helper methods

`_split` method is used to split the data based on a threshold for a particular feature. And `_most_common_label` returns the most frequent label in a node using for leaf nodes.

4.8. Predict

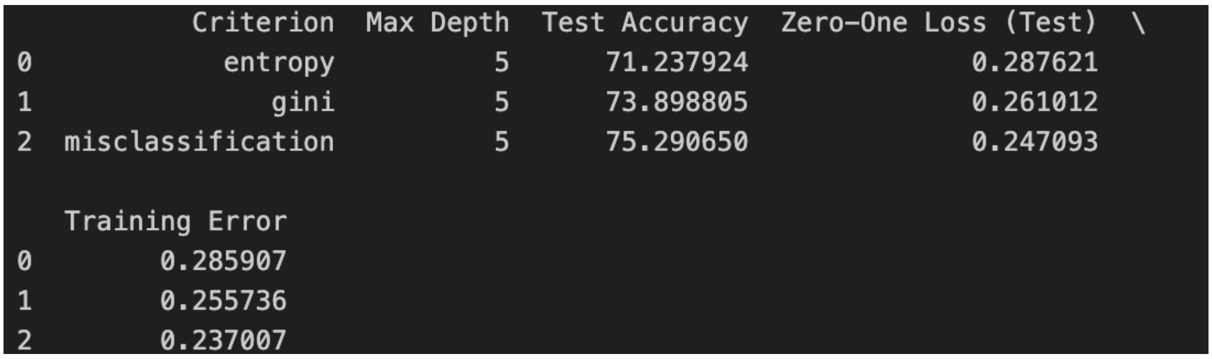
Lastly with predict method the algorithm traverses the tree checking the feature and and threshold values at each decision node.

4.9. Training the Model

The model was then trained using **80%** of the dataset for training and **20%** for testing:

5. Performance Evaluation

I evaluated the performance of the model using accuracy and **0-1 loss**, where accuracy represents the correct predictions, and 0-1 loss is the percentage of incorrect predictions. The results for different criteria and depths=5 is shown below:



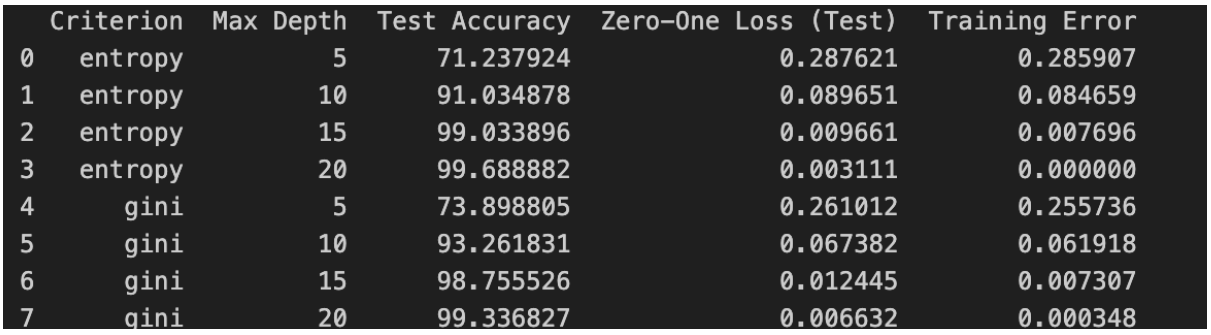
```
         Criterion  Max Depth  Test Accuracy  Zero-One Loss (Test)  \
0          entropy          5      71.237924              0.287621   
1             gini          5      73.898805              0.261012   
2  misclassification        5      75.290650              0.247093   

   Training Error  
0        0.285907  
1        0.255736  
2        0.237007  
```

We can see that at depth 5, the decision tree provides a good balance between complexity and generalization.

Considering that the misclassification criterion is a simplistic way of calculating impurity, focusing only on the majority class. When the tree depth exceeds 5, it encounters an out-of-index error due to the simplicity of the misclassification criterion. Essentially, this criterion does not provide enough information for further splits at deeper levels, resulting in empty nodes and an index error.

6. Hyperparameter Tuning

I performed **hyperparameter tuning** by testing different combinations of the decision criteria (Gini vs. Entropy) and tree depth. As expected, deeper trees performed better in terms of accuracy, but they also run the risk of overfitting.



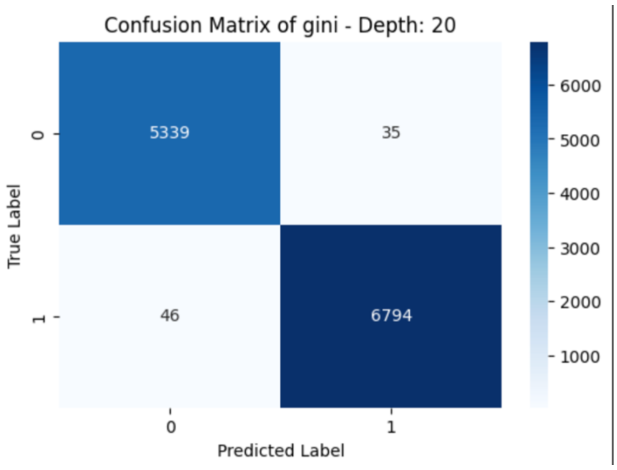
```
  Criterion  Max Depth  Test Accuracy  Zero-One Loss (Test)  Training Error
0   entropy          5      71.237924              0.287621        0.285907
1   entropy         10      91.034878              0.089651        0.084659
2   entropy         15      99.033896              0.009661        0.007696
3   entropy         20      99.688882              0.003111        0.000000
4      gini          5      73.898805              0.261012        0.255736
5      gini         10      93.261831              0.067382        0.061918
6      gini         15      98.755526              0.012445        0.007307
7      gini         20      99.336827              0.006632        0.000348
```

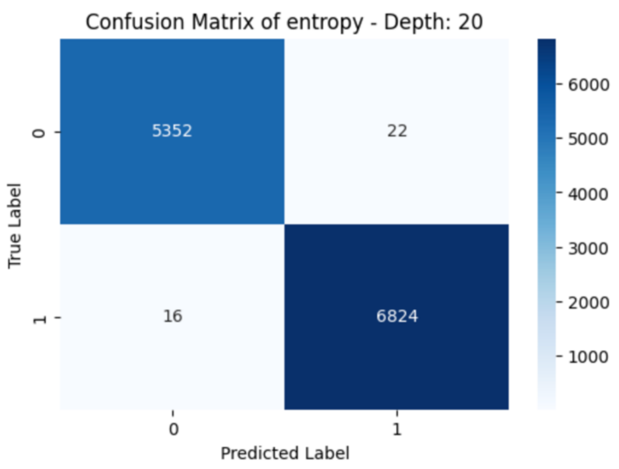
The hyperparameter tuning was done excluding the criterion of misclassification, and for different depth of 5, 10, 15, and 20 getting the result considering Entropy and Gini criterion. Gini and Entropy are more sensitive to class proportions and handle more complex splits, allowing the tree to grow deeper without the index errors that occurred with misclassification. These criteria allow the model to grow deeper and handle more complex splits. However, depths beyond 10 may lead to overfitting, where the tree memorizes the training data but performs poorly on unseen data. We can see that in our result we can clearly observe that as the depth increases, the **training accuracy** is improving, because the tree becomes more complex and better fits the training data.

Moreover, in regard to underfitting and overfitting of the model at different max depths, we can observe that at depth 5 the model maybe **underfitting** slightly, as the errors are similar, but the accuracy is relatively low (~71-74%). Depth 10 however, provides a **good fit**, where the training and test errors are low and similar.

7. Confusion matrix

below I've showed two examples of the confusion matrices, for two different criteria of Gini and Entropy. At the depth 20, which is higher compared to the other depths I've tried, we have the highest test accuracy for both of the criterion. And we can see that entropy compared to gini can has a higher accuracy as the true labels and the predicted labels [0,0] and [1,1] have a higher number compared to the gini criterion result.





8. Conclusion

This project successfully implemented a decision tree from scratch to classify mushrooms as poisonous or edible. The DecisionMTree model performed well on this binary classification task, especially with deeper trees using the **Entropy** criterion, achieving **99.69% accuracy**. Based on the results, a depth of 5 or 10 seems ideal, providing a good trade-off between training and test accuracy. Both Gini and Entropy criteria perform similarly at all depths but may vary slightly in their performance depending on the dataset. Both handle deeper trees well without errors. On the other hand, misclassification is more rigid and causes errors at larger depths. It is suitable for shallow trees but should be avoided for deeper ones unless you modify the stopping conditions to prevent errors. Some features are highly imbalanced such as gill spacing, which could negatively affect the model. Consider balancing the data or using techniques like class weighting to improve performance.

In future work, **post-pruning** and other regularization methods can be employed to mitigate overfitting at deeper levels, and to top the tree from growing excessively, especially for depths greater than 10.

This report provides a detailed explanation of the methodology, from data preprocessing and EDA to model development and performance evaluation, offering insights into decision tree algorithms and their application in binary classification tasks.

References:

- Shalev-Shwartz, S. & Ben-David, S. (2014). *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press.
- Mohri, M., Rostamizadeh, A., & Talwalkar, A. (2012). *Foundations of Machine Learning*. MIT Press.