Decision Tree Classifier for binary classification

Simone Barbalace
Data Science and Economics
Matricola: 923835
Statistical Methods for Machine Learning

October 8, 2024

Contents

1 Introduction

The goal of this project is to build binary decision tree classifiers from scratch that must be able to classify the mushrooms into two main categories: edible or poisonous. To accomplish this objective, we implemented two classes: Node class and Tree class. In these two classes are contained all the methods and all the attributes necessary to run our decision tree classifiers in a recursive manner. Our interest is in observing the differences between each classifier according to their split approach, we used 3 different split functions and several stopping criteria.

2 Dataset Description

The dataset used for this study consists of 61,069 mushroom samples, each represented by 21 features. These features describe various morphological characteristics of the mushrooms, which are essential for predicting whether they are poisonous or edible. The features are primarily categorical, although some numerical measurements are also included.

2.1 Features

- **class**: This is the target variable of the dataset, indicating whether the mushroom is poisonous (denoted by 'p') or edible (denoted by 'e').
- **cap_diameter**: A numerical feature representing the diameter of the mushroom cap.
- **cap_shape**: A categorical feature that describes the shape of the mushroom's cap. Different shapes are represented by unique symbols (e.g., 'x' for convex, 'f' for flat, etc.).
- **cap_surface**: Describes the texture of the mushroom's cap surface, such as whether it is smooth, fibrous, or scaly.
- **cap_color**: A categorical feature representing the color of the mushroom cap. This feature can take on values such as 'e' (buff), 'o' (orange), and others.
- **does_bruise_or_bleed**: Indicates whether the mushroom bruises or bleeds when damaged. 'f' means false (does not bruise/bleed) and 't' means true (does bruise/bleed).
- **gill_attachment**: Refers to how the gills (the spore-producing part of the mushroom) are attached to the stem. This feature contains categorical values.
- **gill_spacing**: A categorical feature indicating whether the gills are close or wide apart.
- **gill_color**: Describes the color of the mushroom's gills. The color categories include 'w' (white), 'p' (pink), and others.
- **stem_height**: A numerical feature representing the height of the mushroom stem in centimeters.
- **stem_width**: A numerical feature indicating the width of the mushroom stem at its widest point.

- **stem_root**: A categorical feature that refers to the type of root the mushroom has, such as 's' for a bulbous base or 'r' for a rooted base.
- **stem_surface**: Describes the surface texture of the mushroom's stem, such as smooth, fibrous, or scaly.
- **stem_color**: A categorical feature representing the color of the mushroom's stem, with values like 'w' (white) or 'y' (yellow).
- **veil_type**: Indicates the type of veil (a membrane covering the mushroom cap) the mushroom has.
- **veil_color**: The color of the veil, which can be 'w' (white) or other colors.
- **has_ring**: A binary feature that denotes whether the mushroom has a ring on its stem ('t' for true, 'f' for false).
- **ring_type**: Describes the type of ring the mushroom has, such as 'g' for a large ring or 'p' for a pendant.
- **spore_print_color**: Refers to the color of the mushroom's spore print. Spores, which are reproductive cells, leave a print when dropped onto a surface.
- **habitat**: A categorical feature indicating where the mushroom is found, such as 'g' for grassy areas or 'd' for wooded areas.
- **season**: Refers to the season in which the mushroom is found, with values like 'w' (winter) or 'u' (summer).

2.2 Preprocessing

For sake of simplicity all the None values were converted into 0 values and given the great amount of categorical values, we converted all categories into numerical values. This conversion from categories to numerical values was made because it was more intuitive to make internal tests with numerical values and numerical thresholds.

3 Model Structure

3.1 Splitting Criteria

In order to observe and analyze the performance of our algorithm with different splitting criteria, we implemented three distinct functions, each of which calculates a different measure to select the most suitable feature and threshold for creating meaningful and efficient splits.

The algorithm evaluates the potential splits for each feature and chooses the one that minimizes impurity or maximizes information gain.

3.1.1 Gini Index

Gini index is used to measure the "impurity" of a node by computing how often a randomly chosen element would be incorrectly classified. It is calculated as:

$$G = 1 - \sum_{i=1}^{k} p_i^2$$

where p_i is the probability of class i in the node.

This measure is highly efficient for classification problems, particularly for binary splits, which is why it was chosen as the default splitting criterion.

3.1.2 Entropy

Entropy quantifies the uncertainty of a random variable, and in the case of decision trees, it measures the unpredictability of the classification outcome. The formula for entropy is:

$$H = -\sum_{i=1}^{k} p_i \log_2(p_i)$$

For our classifier, entropy was used as a secondary splitting criterion to explore its effectiveness compared to Gini.

3.1.3 Misclassification Error

Misclassification error is a simpler measure, defined as the fraction of incorrect classifications at each node. It is calculated as:

$$E = 1 - \max(p_i)$$

where p_i is the proportion of samples of class i in the node.

3.2 Node Class

This is the class which describes the node structure. Each node can be an internal node with an internal test which splits data into left child or right child or in alternative a node can be a leaf with a prediction value correspondent to the dominant class occurring in that group of data points.

```
class Node:
    def __init__(self, feature_index=None, threshold=None,
                 left_child=None, right_child=None, prediction=None):
        self.feature_index = feature_index
        self.threshold = threshold
        self.left_child = left_child
        self.right_child = right_child
        self.prediction = prediction


```

```python
    def classify(self, x):
        if self.prediction is not None:
            return self.prediction

        if x[self.feature_index] < self.threshold:
            return self.left_child.classify(x)
        else:
            return self.right_child.classify(x)
```

Each instance of a node object contains these attributes and methods:

- **left_child**: The left child of the node.
- **right_child**: The right child of the node.
- **feature_index**: Index of the feature to split on.
- **threshold**: Threshold value for the split.
- **prediction**: The predicted label if the node is a leaf.
- **classify method**: This is the recursive function responsible for classifying each data point. If it reaches a leaf node, it returns the corresponding prediction. Otherwise, it proceeds by invoking the classify method to compare the feature value of the data point x against the threshold of the current node: if the feature value in x is less than the threshold, the function will recurse on the left child; otherwise, it will recurse on the right child. This function will be fundamental with the `predict` method of the `TreePredictor` class.

3.3 TreePredictor Class

`TreePredictor` is the core class which contains all the methods necessary to build the tree recursively, split nodes, and predict the label of a given input. In the code below we can see all the main attributes which are essentially the root, the stopping criteria we adopted during the project and the splitting criterion used in that tree.

3.3.1 TreePredictor attributes

```python
class TreePredictor:
    def __init__(self, max_depth=5, min_samples_split=2, split_criterion='gini'):
        self.root = None
        self.max_depth = max_depth
        self.min_samples_split = min_samples_split
        self.split_criterion = split_criterion
```

- **max_depth**: The maximum depth allowed for the tree.
- **min_samples_split**: Minimum number of samples required to split a node.
- **split_criterion**: The criterion for selecting splits (Gini, entropy, etc.).

3.3.2 Recursive Tree Building with the build_tree method

The build_tree method, shown below, is responsible for recursively constructing the decision tree starting from the root node. At each step, it selects the best feature and threshold to split the data, and then continues to recursively build the left and right child nodes until one of the stopping criteria is met (such as maximum depth or minimum number of samples).

```python
    def build_tree(self, X, y, depth):

        n_samples, n_features = np.shape(X)
        if depth >= self.max_depth or n_samples < self.min_samples_split or len(
    np.unique(y)) == 1:
            leaf_value = self.majority_class(y)
            return Node(prediction=leaf_value)

        best_feature, best_threshold = self.find_best_split(X, y)

        if best_feature is None:
            leaf_value = self.majority_class(y)
            return Node(prediction=leaf_value)

        node = Node(feature_index=best_feature, threshold=best_threshold)

        left_idx = X[:, best_feature] < best_threshold
        right_idx = ~left_idx


        node.left_child = self.build_tree(X[left_idx], y[left_idx], depth + 1)
        node.right_child = self.build_tree(X[right_idx], y[right_idx], depth + 1)

        return node
```

In the code above, the first if statement checks the stopping criteria, which includes reaching the maximum tree depth (max_depth), having fewer than the minimum number of samples required to split (min_samples_split), or when all the labels in the current node are the same. If any of these conditions are met, the method returns an instance of the Node class with the prediction attribute set to the majority class, which is computed using the corresponding method (`majority_class`). Next, the best feature and the best threshold are determined using the `find_best_split` method. In the second if statement, we check whether a valid best_feature was found. If not, the method returns an instance of the Node class with the majority class prediction, as no further split is possible. Once the best feature and threshold are found, the rows in X corresponding to this feature are compared to the threshold: rows with feature values less than the threshold will recursively form the left child node, and rows with feature values greater than or equal to the threshold will form the right child node. Finally, the method returns an instance of the Node class representing the current decision node, with the corresponding attributes feature_index, threshold, and child nodes.

3.3.3 Best feature and threshold identification with find_best_split method

In the code below, we can observe the method responsible for identifying the best feature and threshold for splitting the data. Initially, the variables best_feature, best_threshold, and best_impurity are initialized. The loop then iterates over each feature of the dataset X, identifying potential thresholds as the unique values in the current feature column. For each threshold, the data is split into two subsets: the values in the current feature that are less than the threshold are assigned to the left subset, while the remaining values are assigned to the right subset.

Next, the method checks if both the left and right subsets have at least one sample each (i.e., the sums of both subsets are greater than zero). If this condition is satisfied, the impurity of each subset is calculated using the calculate_impurity method, and the weighted average of the impurity for the split is computed.

Finally, the method checks if the total impurity for this split is smaller than the current best_impurity. If so, it updates the best_impurity, best_feature (the index of the feature being evaluated), and best_threshold (the current threshold being tested). Once all features and thresholds have been evaluated, the method returns the best feature and best threshold that minimize impurity.

```
    def find_best_split(self, X, y):
        best_feature, best_threshold, best_impurity = None, None, float('inf')
        current_impurity = self.calculate_impurity(y)

        for feature_idx in range(X.shape[1]):
            thresholds = np.unique(X[:, feature_idx])
            for threshold in thresholds:
                left = X[:, feature_idx] < threshold
                right = ~left

                if np.sum(left) > 0 and np.sum(right) > 0:
                    left_impurity = self.calculate_impurity(y[left])
                    right_impurity = self.calculate_impurity(y[right])
                    impurity = (np.sum(left) * left_impurity + np.sum(right) *
    right_impurity) / len(y)

                    if impurity < best_impurity:
                        best_impurity = impurity
                        best_feature = feature_idx
                        best_threshold = threshold

        return best_feature, best_threshold
```

3.3.4 The methods used in build_tree and find_best_split

```
    def calculate_impurity(self, y):
        if self.split_criterion == 'gini':
            return gini_impurity(y)
        elif self.split_criterion == 'entropy':
            return entropy(y)
        elif self.split_criterion == 'misclassification':
            return misclassification_error(y)
        else:
            raise ValueError(f"Invalid splitting criterion: {self.split_criterion
    }")

    def majority_class(self, y):
        return np.bincount(y).argmax()

    def predict(self, X):
        return np.array([self.root.classify(x) for x in X])

    def evaluate(self, X, y):
        predictions = self.predict(X)
        loss = np.mean(predictions != y)
        return loss
```

4 Training and Prediction

4.1 Training Process

Training is performed by passing the dataset through the decision tree, starting from the root. At each internal node, the data is split based on the best feature and threshold, and the subsets are passed recursively to the child nodes until a leaf node is reached or a stopping criterion is met. The `fit` method is responsible for invoking the `build_tree` method, which recursively constructs the tree as described previously.

```
def fit(self, X, y):
        self.root = self.build_tree(X, y, depth=0)
```

4.2 Prediction

The prediction for a new data point is made using the classify method, as previously explained. The input data is passed through the tree, and at each internal node, it is directed to either the left or right child based on whether the feature value is less than or greater than the node's threshold. This process continues recursively until a leaf node is reached, where the final prediction is made.

5 Results

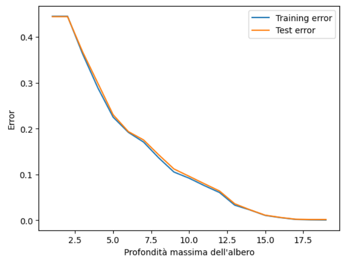
We evaluated the decision tree on both training and test sets. The training error was calculated using 0-1 loss, where incorrect predictions were penalized by 1 and correct predictions by 0. The tree was tuned for different depths and minimum sample splits to achieve a balance between training accuracy and generalization. Below you can see the code for the hyperparameters tuning.

```
   param_grid = {
    'max_depth': [5, 10, 15],
    'min_samples_split': [2, 5, 10],
    'split_criterion': ['gini', 'entropy', 'misclassification']
}

def hyperparameter_tuning(X_train, y_train, X_val, y_val, param_grid):

    best_params = None
    best_loss = float('inf')


    for max_depth in param_grid['max_depth']:
        for min_samples_split in param_grid['min_samples_split']:
            for split_criterion in param_grid['split_criterion']:
                print(f"Testando: max_depth={max_depth}, min_samples_split={
    min_samples_split}, split_criterion={split_criterion}")


                tree = TreePredictor(max_depth=max_depth, min_samples_split=
    min_samples_split, split_criterion=split_criterion)
                tree.fit(X_train, y_train)


                loss = tree.evaluate(X_val, y_val)
                print(f"Loss: {loss}")


                if loss < best_loss:
                    best_loss = loss
                    best_params = {
                        'max_depth': max_depth,
                        'min_samples_split': min_samples_split,
                        'split_criterion': split_criterion
                    }

    return best_params, best_loss


best_params, best_loss = hyperparameter_tuning(X_train, y_train, X_test, y_test,
    param_grid)
print(f"Migliori parametri trovati: {best_params} con loss={best_loss}")
```

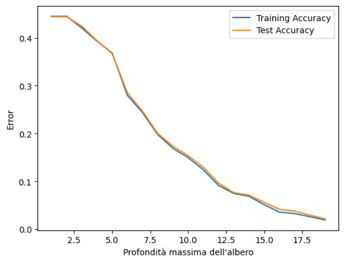
In the code above, we defined a grid of possible hyperparameters and a function that takes as input the training set X_train, training labels y_train, validation set X_test, validation labels y_test, and the parameter grid. We initialized the variables best_params and best_loss, which are updated at each iteration when a better combination of hyperparameters (yielding a lower loss) is found. The function contains three nested loops to try all possible combinations of hyperparameters for each splitting criterion, printing the results at each step. For each combination, we trained the model and evaluated it on the validation set. Finally, the function updates best_loss and best_params with the lowest loss and the best combination of hyperparameters.

5.1 Training and Test Errors

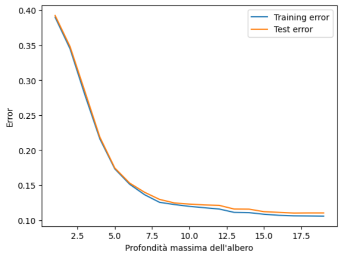
Below we reported some plot of the training and test errors for each split criteria based on the max_depth of each tree.



(a) Train and Test Error using Entropy Criterion



(b) Train and Test Error using Gini Criterion



(c) Train and Test Error using Misclassification Criterion

Figure 1: Comparison of Train and Test Error with Different Splitting Criteria

6 Conclusion

In this project, we successfully implemented a binary decision tree classifier from scratch and evaluated its performance using the mushroom dataset. The model achieved high accuracy on both the training and test sets, demonstrating its effectiveness. The results showed that the best criteria for mushroom classification were Gini and Entropy, with comparable performances, although Gini performed slightly better. The misclassification criterion produced the lowest performance overall. However, for simpler trees with lower depth, the misclassification criterion showed better error rates, while its performance diminished as the tree complexity increased compared to Gini and Entropy.