Università degli Studi di Milano Statale

Machine Learning and Statistical Learning

Data Science for Economics

Binary Classification with Decision Trees

Name Surname: **Honest Albert Temu**

Matriculation Number: **34538A**

Email: honestalbert.temu@studenti.unimi.it

I declare that this material, which I now submit for assessment, is entirely my 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 work. I 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 or any other person for assessment on this or any other course of study

1. Introduction

The goal of this project is to use and test decision tree predictors for binary classification, particularly to identify whether mushrooms are toxic based on a variety of characteristics.
The project's development starts with a careful examination of the dataset.
as well as the creation of a successful data preprocessing plan that separates training and test sets. The project's main objective is to create tree predictors from the ground up using single-feature binary tests at every internal node.
This entails building a simple node structure and a tree predictor class that can assess performance on unseen data and train on the provided dataset.

We will apply and compare three different criteria for leaf expansion, including Gini impurity, scaled entropy and squared error. As for the stopping criteria, three were implemented: maximum depth, minimum sample splitting and minimum impurity decrease. These criteria are essential to avoid underfitting or overfitting of the model. These standards are crucial for limiting the models' complexity and avoiding both overfitting and underfitting.

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

2. Data Summary

This dataset includes descriptions of hypothetical samples of 61069 mushrooms with caps based on 173 species (353 mushrooms per species). Each species is identified as definitely edible, definitely poisonous, or of unknown edibility and not recommended. This latter class was combined with the poisonous one. The Guide clearly states that there is no simple rule for determining the edibility of a mushroom; no rule like "leaflets three, let it be" for Poisonous Oak and Ivy.

The attributes of the dataset include:

i. Dependent Variable

- Classification: Edible or Poisonous

ii. Independent Variables

- Cap Diameter : Float number in cm
- Cap Shape : Bell, Conical, Convex, Flat, Sunken, Spherical, Others
- Cap Surface : Fibrous, Grooves, Scaly, Smooth, Shiny, Leathery, Silky, Sticky, Wrinkled, Fleshy, d
- Cap Color : Brown, Buff, Gray, Green, Pink, Purple, Red, White, Yellow, Blue, Orange, Black
- Bruise or Bleed : True, False
- Gill Attachment : Adnate, Adnexed, Decurrent, Free, Sinuate, Pores, None
- Gill Spacing : Close, Distant, None
- Gill Color : Brown, Buff, Gray, Green, Pink, Purple, Red, White, Yellow, Blue, Orange, Black, None
- Stem Height : Float number in cm
- Stem Width : Float number in mm
- Stem Root : Bulbous, Swollen, Club, Cup, Equal, Rhizomorphs, Rooted

- Stem Surface : Fibrous, Grooves, Scaly, Smooth, Shiny, Leathery, Silky, Sticky, Wrinkled, Fleshy, None
- Stem Color : Brown, Buff, Gray, Green, Pink, Purple, Red, White, Yellow, Blue, Orange, Black, None
- Veil Type : Partial, Universal
- Veil Color : Brown, Buff, Gray, Green, Pink, Purple, Red, White, Yellow, Blue, Orange, Black, None
- Has Ring : True, False
- Ring Type : Cobwebby, Evanescent, Flaring, Grooved, Large, Pendant, Sheathing, Zone, Scaly, Movable, None
- Spore Print Color : Brown, Buff, Gray, Green, Pink, Purple, Red, White, Yellow, Blue, Orange, Black
- Habitat : Grasses, Leaves, Meadows, Paths, Heaths, Urban, Waste, Woods
- Season : Spring, Summer, Autumn, Winter

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.

Thereafter, 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.

```
# Separate features and target
X = data_encoded.drop(['class_p', 'class_e'], axis=1)
y = data_encoded['class_p']
```

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)
```

3. Decision Tree

A decision tree is a supervised learning algorithm used for both classification and regression tasks. It models decisions as a tree-like structure where internal nodes represent attribute tests, branches represent attribute values, and leaf nodes represent final decisions or predictions. Decision trees are versatile, interpretable, and widely used in machine learning for predictive modeling.

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

3.1 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.

• ***Left***: The left child node.

• ***Right***: The right child node

• ***Value***: 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 Implementation of decision trees

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

3.2.1 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.

$\psi(p) = 2p \cdot (1 - p)$

Algorithm 1 Gini Impurity Calculation

```
function GINI_IMPURITY(y)
    set hist ← count of each unique value in y
    set probs ← hist divided by the length of y
    set gini ← 1.0− sum of (probs^2)
    return gini
end function
```

3.3.2 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.

$\psi(p) = - p/2 \log2 (p) - (1-p)/2 \log2 (1 - p)$

Algorithm 1 Scaled Entropy Calculation

```
function SCALED_ENTROPY(y)
    set hist ← count of each unique value in y
    set probs ← hist divided by the length of y
    set scaled_ent ← − Σ_{p>0} ( p/2 × log_2 p )
    return scaled_ent
end function
```

3.3.3 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.

$$\psi(p) = \sqrt{p(1-p)}$$

Algorithm 1 Squared Impurity Calculation

```
function SQUARED_IMPURITY(y)
    set hist ← count of each unique value in y
    set probs ← hist divided by the length of y
    set epsilon ← 1e−10          ▷ Small value to avoid multiplying by zero
    set sqr ← ∑ √((probs + ϵ) × (1 − probs + ϵ))
    return sqr
end function
```

The algorithm evaluates all possible splits at each node and selects the one that maximizes homogeneity or minimizes impurity.

The splitting continues until either one of the stopping criterions

• **Maximum tree depth**: Limiting the depth of the tree to a predefined value.

• **Maximum Number of Leaf Nodes:** Controlling the number of leaf nodes in the tree.

• **Impurity Threshold**: Stopping when the impurity falls below a certain threshold.

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

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.

Algorithm 1 Grow Tree Algorithm

```
Function _grow_tree(x, y, depth = 0):
    if depth > self.depth:
        self.depth ← depth
    if stop_conditions_met(x, y, depth):
        return TreeNode(value = most_common_label(y))
    feat_idxs ← random_selection_of_features(x)
    best_feat, best_thresh ← find_best_split(x, y, feat_idxs)
    if best_feat is none:
        return TreeNode(value = most_common_label(y))
    increment  leaf_count
    left_idxs, right_idxs ← split_data(x, best_feat, best_thresh)
    left ← _grow_tree(x[left_idxs], y[left_idxs], depth + 1)
    right ← _grow_tree(x[right_idxs], y[right_idxs], depth + 1)
    return TreeNode(best_feat, best_thresh, left, right)
```

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

Algorithm 1 Best Criteria Selection

```
function BEST_CRITERIA(X, y, feat_idxs)
    set best_gain ← -1
    set split_idx, split_thresh ← None
    for each feat_idx in feat_idxs do
        Extract X_column from X using feat_idx
        Get unique thresholds from X_column
        for each threshold in thresholds do
            Compute gain ← gain(y, X_column, threshold)
            if gain > best_gain then
                Update best_gain ← gain
                Update split_idx ← feat_idx
                Update split_thresh ← threshold
            end if
        end for
    end for
    return split_idx, split_thresh
end function
```

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

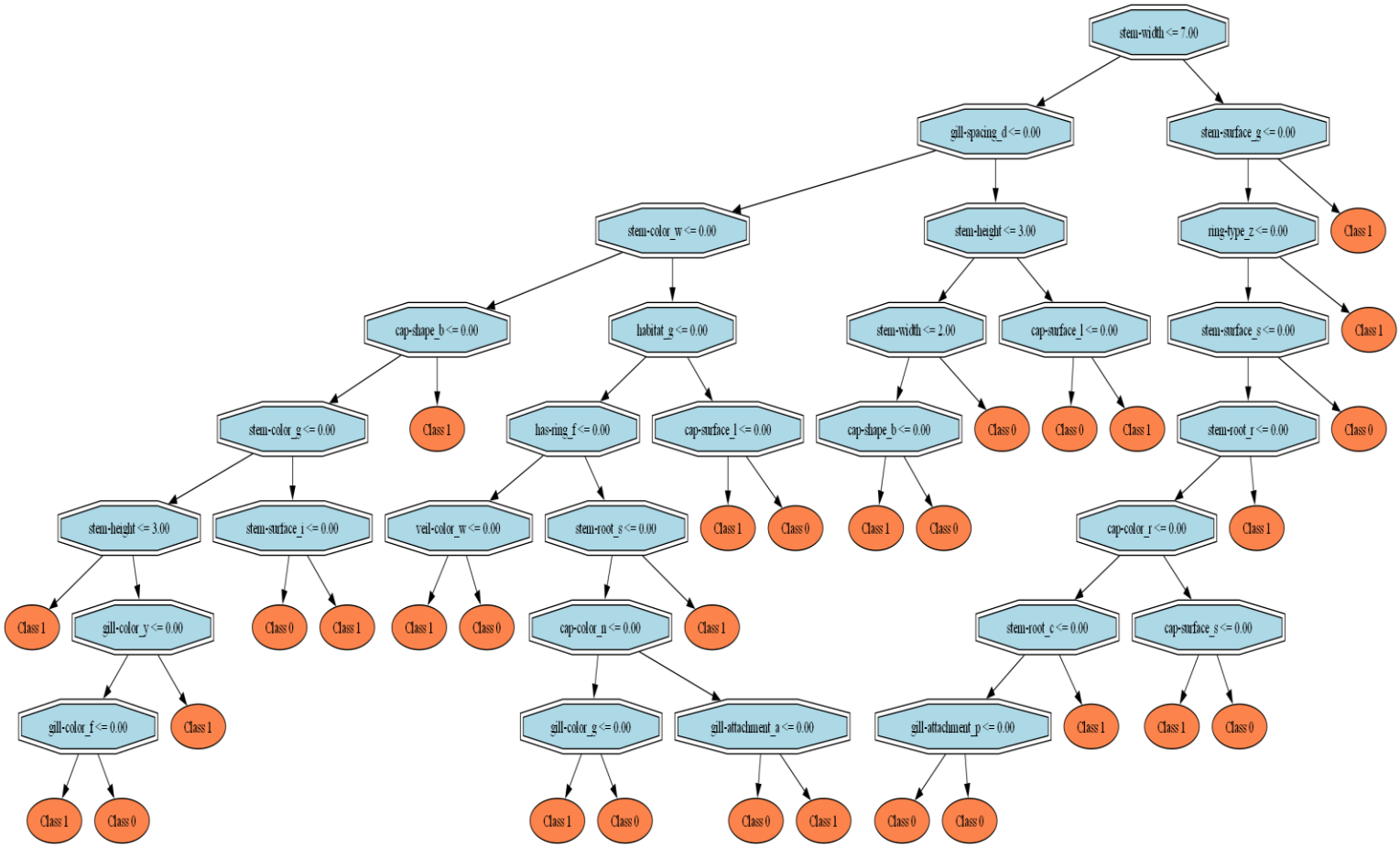
Algorithm 1 Split Function

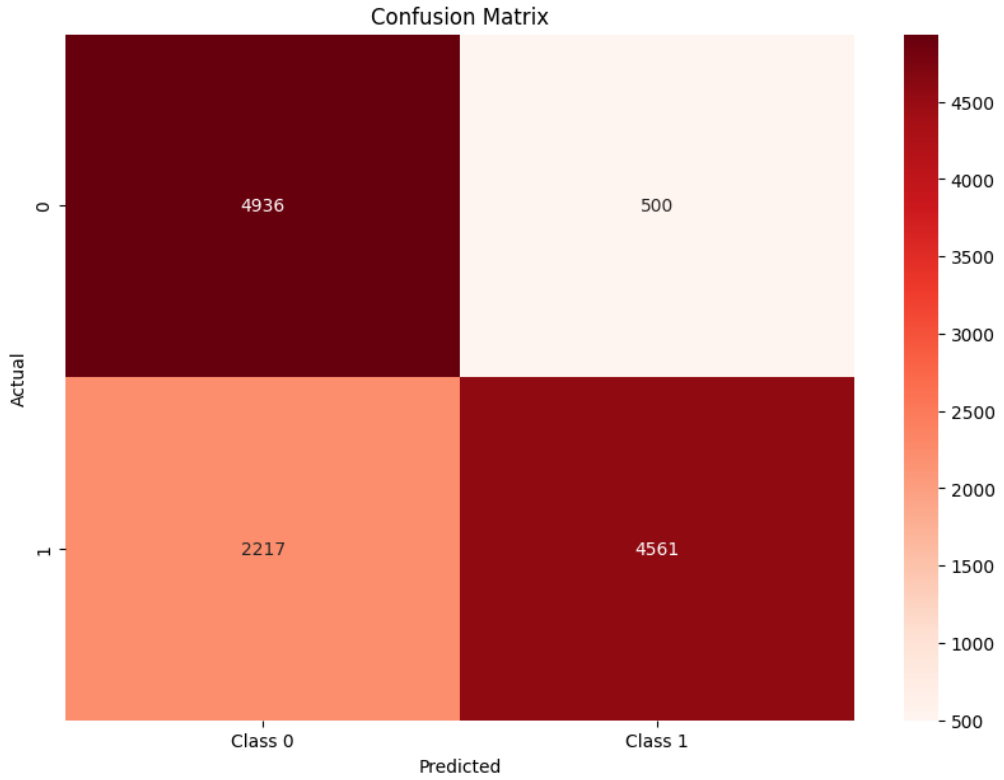
```
function SPLIT(X_column, split_thresh)
    set left_idxs ← indices where X_column ≤ split_thresh
    set right_idxs ← indices where X_column > split_thresh
    return left_idxs, right_idxs
end function
```

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.

Algorithm 1 Gain Calculation

```
function GAIN(y, X_column, split_thresh)
    set parent_criterion ← CRITERION_FUNC(y)
    set left_idxs, right_idxs ← SPLIT(X_column, split_thresh)
    if LENGTH(left_idxs) = 0 or LENGTH(right_idxs) = 0 then
        return 0
    end if
    set y_left ← elements of y at left_idxs
    set y_right ← elements of y at right_idxs
    set child_criterion ← WEIGHTED_CRITERION(y_left, y_right, criterion_func)
    set gain ← parent_criterion - child_criterion
    return gain
end function
```






```
Train accuracy: 0.776748
Test accuracy: 0.777550
0 - 1 Loss: 0.222450
```

3 Hyperparameter Tuning

The fine-tuning of hyperparameters refers to the process of optimizing the hyperparameters of a machine learning model. Hyperparameters are settings that must be specified before training the model and can significantly influence the final performance of the model. In our decision trees, the hyperparameters are the stopping and splitting criteria. The correct adjustment of these hyperparameters is essential because if the tree grows too large, it can capture noise in the data, resulting in poor generalization. On the other hand, if the tree is too small, it may fail to capture important patterns in the data. Hyperparameters should not be confused with model parameters. Hyperparameters are defined by the user before training, while parameters are values that the model learns automatically during training. An example of a parameter in the Decision Tree is the splitting value associated with each node. The goal of hyperparameter tuning is to find the optimal balance that maximizes the performance of the model on the unseen data.

On the other hand, Cross-validation is a technique used to more reliably assess the performance of a model, particularly when adjusting hyperparameters. It works by splitting the data into several subsets or folds. In the project, k-fold cross-validation was implemented, the data is divided into k parts of equal size. The model is trained k times, each time using k – 1 folds as the training set and the remaining folds as the test set. This process is repeated for each fold and the average loss of all iterations is taken as the final evaluation score. The pseudocode of the grid search algorithm can be shown as:

Algorithm 1 Grid Search for Model Optimization

```
function GRID_SEARCH(X_train, y_train, param_grid, scoring_func)
    set cv ← 5-fold cross-validation
    function EVALUATE_PARAMS(params)
        Initialize current_scores, depths, leafs as empty lists
        for each train_idxs, val_idxs split from cv do
            Split X_train, y_train into training and validation sets
            Initialize model with params
            Fit model on training set
            Predict y_val_pred using model
            Compute score using scoring_func
            Append score, model depth, and leaf count to lists
        end for
        Compute mean_score, mean_depth, mean_leafs
        Print evaluation results
        return params, mean_score
    end function
    Generate param_combinations from param_grid
    Print total number of iterations
    Parallelize evaluation for all param_combinations using evaluate_params
    Sort results by mean_score and get top 3
    Print top 3 results
    return all results, best_params, best_score
end function
```

Performance Analysis

In this chapter 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.

The hyperparameter tuning process involved evaluating different combinations of parameters using K-fold cross-validation (cv). A total of 9 parameter combinations were tested, with 5-fold cross-validation, leading to 45 iterations. The three best-performing models were ranked based on the lowest 0-1 loss, a loss metric used to assess the model's effectiveness.

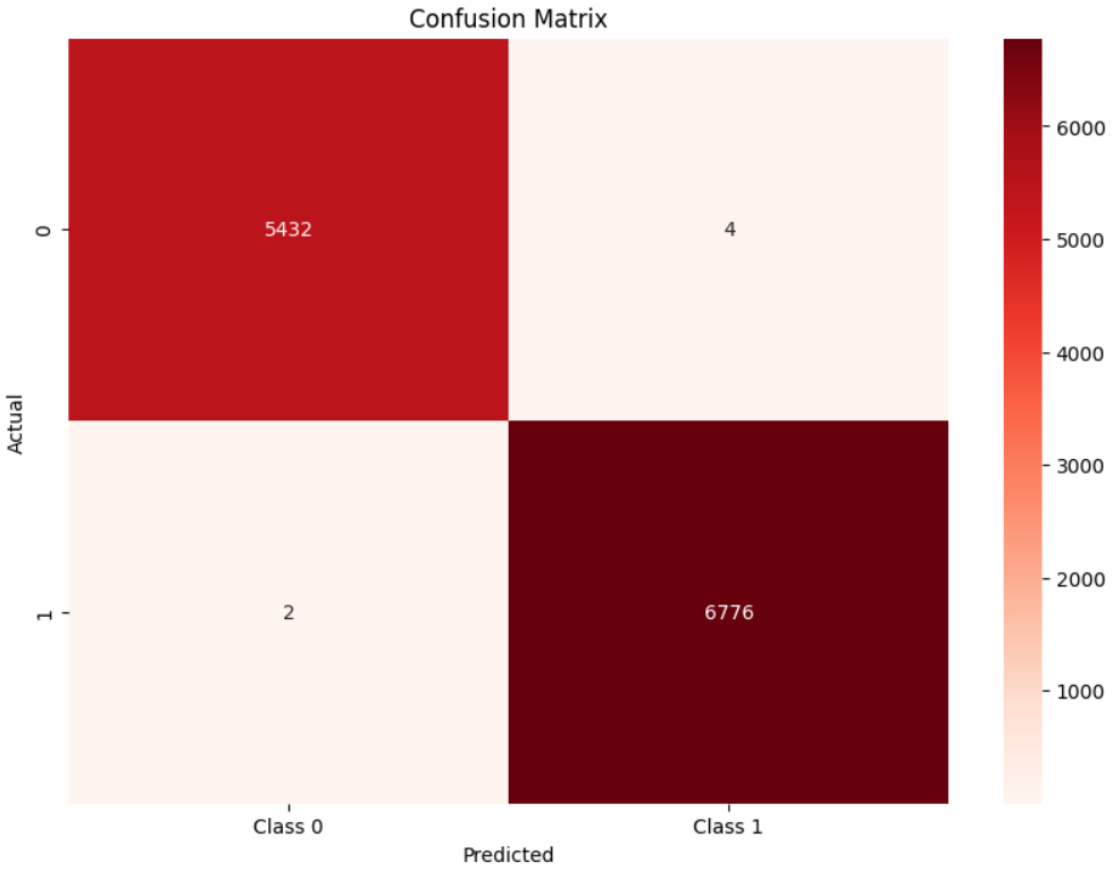
```
Top 3 Combinations:
1:  Combination of: {'max_leaf_nodes': 200, 'split_function': 'gini'},   Mean 0_1 loss: 0.000819
2:  Combination of: {'entropy_threshold': 0.0001, 'split_function': 'scaled_entropy'},   Mean 0_1 loss: 0.000839
3:  Combination of: {'entropy_threshold': 0.0001, 'split_function': 'gini'},   Mean 0_1 loss: 0.000839
Execution time: 225.49895691871643 seconds

Best score: : 0.000819
Best Hyperparameters Combination: {'max_leaf_nodes': 200, 'split_function': 'gini'}
```

The best combination was **max_depth** of **30** and split_function of **scaled_entropy**, achieving the lowest mean **0-1 loss** of **0.000839**, while second-best combination used max_depth of 30 and split_function of gini, slightly higher with **0-1 loss** of **0.000860.** And finally, the third-best combination had **max_leaf_nodes** of **200** and split_function of **scaled_entropy**, with the same **0-1 loss** of **0.000860**.

The best hyperparameter set (max_depth: 30 and split_function: scaled_entropy) was chosen due to its **lowest loss**, ensuring optimal model complexity without overfitting.

The confusion matrix on the hand provides a detailed breakdown of the model's classification performance:




```
Train accuracy: 1.000000
Test accuracy: 0.999754
0 - 1 Loss: 0.000246
```

- **True Positives (TP = 6776)**: Correctly identified positive class instances.
- **True Negatives (TN = 5432)**: Correctly identified negative class instances.
- **False Positives (FP = 4)**: Negative instances wrongly classified as positive.
- **False Negatives (FN = 2)**: Positive instances wrongly classified as negative.

These numbers indicate **extremely low misclassification**, suggesting the model is highly reliable. The very low **false positive and false negative rates** confirm the model's effectiveness in distinguishing between classes.

As for the accuracy aspect,

- **Training Accuracy = 1.000000 (100%)**
    - The model perfectly classified all training samples, indicating that it fully captured patterns in the training data.
- **Test Accuracy = 0.999509 (99.95%)**
    - The model generalizes exceptionally well to unseen data, maintaining nearly perfect accuracy.
- **0-1 Loss = 0.000491**
    - A very small loss value, showing that the model's predictions are highly reliable and that errors are almost negligible.

Execution Time Considerations

The execution time of **164.05 seconds (2.44 minutes)** suggests a moderate computational cost for hyperparameter tuning. Given the complexity of 45 iterations with K-fold cross-validation, this duration is reasonable, balancing optimization and efficiency.

Conclusion

The hyperparameter tuning process successfully optimized the model by selecting the best combination, minimizing loss, and ensuring robust generalization. The confusion matrix, accuracy, and loss metrics confirm that the model is **highly accurate**, with **very few misclassifications**. The results indicate that the chosen decision tree parameters effectively capture the underlying patterns in the dataset. The model's near-perfect performance makes it highly **suitable for real-world deployment**, particularly in scenarios where high precision and recall are critical.