Tree Predictor for Binary Classification

Franco Reinaldo Bonifacini (41540A)

February 17, 2025

Abstract

This project aims to implement from scratch a **tree predictor for binary classification to determine whether mushrooms are poisonous or not**. Throughout this paper, the entire process of data preparation and model evaluation is explained. Ultimately, **the model delivers promising results** using the optimal parameter combination.

1 Introduction

This report presents the development of a decision tree classifier designed to predict whether a mushroom is poisonous or edible according to its attributes.

The report provides a comprehensive overview of the entire process, including data pre-processing, splitting the dataset into training and test sets, and model evaluation.

The **splitting criteria** considered include **gini impurity, entropy, and misclassification error**. Similarly, for the **stopping criteria**, we use **max_depth and min_samples**, which are crucial in preventing underfitting or overfitting of the model.

During model evaluation, the report details the **hyperparameter tuning process**, utilizing **GridSearchCV** to explore different parameter combinations.

2 Dataset Used and Pre-Processing

2.1 Dataset

The dataset used in this project was obtained from the **UC Irvine Machine learning Repository** [1]. This dataset consists of **61,069 hypothetical mushrooms**, with each cap modeled after one of 173 species—resulting in 353 mushrooms per species. Each mushroom is classified as **edible, poisonous, or of unknown edibility (not recommended for consumption)**.

2.2 Dataset's Attributes

This dataset contains 21 attributes which are:

- **Classification**: Edible or Poisonous.
- **Cap Diameter (Numerical)**: Float number in cm.
- **Cap Shape (Categorical)**: Bell, Conical, Convex, Flat, Sunken, Spherical, Others.
- **Cap Surface (Categorical)**: Fibrous, Grooves, Scaly, Smooth, Shiny, Leathery, Silky, Sticky, Wrinkled, Fleshy, d.
- **Cap Color (Categorical)**: Brown, Buff, Gray, Green, Pink, Purple, Red, White, Yellow, Blue, Orange, Black.
- **Does Bruise or Bleed (Categorical)**: True, False.
- **Gill Attachment (Categorical)**: Adnate, Adnexed, Decurrent, Free, Sinuate, Pores, None.
- **Gill Spacing (Categorical)**: Close, Distant, None.

- **Gill Color (Categorical)**: Brown, Buff, Gray, Green, Pink, Purple, Red, White, Yellow, Blue, Orange, Black, None.
- **Stem Height (Numerical)**: Float number in cm.
- **Stem Width (Numerical)**: Float number in mm.
- **Stem Root (Categorical)**: Bulbous, Swollen, Club, Cup, Equal, Rhizomorphs, Rooted.
- **Stem Surface (Categorical)**: Fibrous, Grooves, Scaly, Smooth, Shiny, Leathery, Silky, Sticky, Wrinkled, Fleshy, None.
- **Stem Color (Categorical)**: Brown, Buff, Gray, Green, Pink, Purple, Red, White, Yellow, Blue, Orange, Black, None.
- **Veil Type (Categorical)**: Partial, Universal.
- **Veil Color (Categorical)**: Brown, Buff, Gray, Green, Pink, Purple, Red, White, Yellow, Blue, Orange, Black, None.
- **Has Ring (Categorical)**: True, False.
- **Ring Type (Categorical)**: Cobwebby, Evanescent, Flaring, Grooved, Large, Pendant, Sheathing, Zone, Scaly, Movable, None.
- **Spore Print Color (Categorical)**: Brown, Buff, Gray, Green, Pink, Purple, Red, White, Yellow, Blue, Orange, Black.
- **Habitat (Categorical)**: Grasses, Leaves, Meadows, Paths, Heaths, Urban, Waste, Woods.
- **Season (Categorical)**: Spring, Summer, Autumn, Winter.

2.3 Pre-processing

During the exploratory data analysis, it was discovered that the **numerical attributes did not contain null values, but the categorical ones did**. To assess their impact, the percentage of null values for each attribute was calculated.

```
class                    0.00
cap-diameter             0.00
cap-shape                0.00
cap-surface             23.12
cap-color                0.00
does-bruise-or-bleed     0.00
gill-attachment         16.18
gill-spacing            41.04
gill-color               0.00
stem-height              0.00
stem-width               0.00
stem-root               84.39
stem-surface            62.43
stem-color               0.00
veil-type               94.80
veil-color              87.86
has-ring                 0.00
ring-type                4.95
spore-print-color       89.60
habitat                  0.00
season                   0.00
dtype: float64
```

After those results, it was fixed a **threshold of 80%**, where those attributes with number of nulls that **exceeded it were removed from the final dataset**. In this case, the columns removed were **stem-root, veil-type, veil-color, spore-print-color**.

Furthermore, a **check on duplicated rows** was done, to reduce the risk of having an over-fitted model. In this case, **146 rows were additionally removed** for this reason.

2.4 Splitting and Encoding

Finally, before entering into the construction of the tree, some additional manipulation was done. The dataset was **split into training (80%) and test (20%) sets**. This was done using the function **split**:

Algorithm 1 Split

Input: $X, y, test\_size = 0.3, seed = None$

1: **if** seed is not None **then**
2: Set random seed to *seed*
3: **end if**
4: Create an index array of size $N \leftarrow |X|$ ▷ Indices
5: Shuffle *indices*
6: $split\_idx \leftarrow n \times (1 - test\_size)$ ▷ Compute split
7: Partition indices:
8: $train\_indices \leftarrow$ indices[: split_idx]
9: $test\_indices \leftarrow$ indices[split_idx:]
10: Split X and y into train and test:
11: $X_{train}, X_{test} \leftarrow X[train\_indices], X[test\_indices]$
12: $y_{train}, y_{test} \leftarrow y[train\_indices], y[test\_indices]$

Output: $X_{train}, X_{test}, y_{train}, y_{test}$

On the other hand, after the split, **the remaining null values were replaced with the mode of the corresponding attribute**, as it was already mentioned that only categorical values had missing values.

Finally, with train and test sets already prepared, the **Encoding class** was used to replace the categorical values with 1 and 0 to get the final train and test sets for implementing the model.

3 Decision Tree

A decision tree is a widely used machine learning model for classification and regression tasks. It functions by **iteratively splitting the input data into subsets that become increasingly homogeneous** with respect to the target variable. The tree structure consists of **nodes, each representing a decision rule or an outcome**. This hierarchical division allows decision trees to efficiently model complex relationships between features and outputs. [2]

3.1 Structure

A decision tree is structured as a **hierarchy of nodes**, beginning with the **root node**, which contains the entire dataset. The tree expands downward as **internal nodes** applying the splitting criteria to separate data into **child nodes**. Each split is **based on a threshold** applied to one of the input features. The tree continues to branch out until it reaches the leaf nodes, which provide the final prediction or classification.

For binary classification tasks, each split divides the data into two subsets based on a decision threshold. The tree grows recursively until a stopping condition is met, such as reaching the maximum depth or the minimum sample requirement per node.

3.2 How does a Decision Tree Work

Decision trees operate by recursively partitioning data based on selected features to **minimize impurity within each subset**. In other words, it aims to increase the homogeneity of the data at a given tree node. The most commonly used splitting criteria for binary classification, also implemented in this project, are:

- **Gini impurity ('gini')**: $2p(1-p)$.
- **Scaled entropy ('entropy')**: $\psi(p) = -\frac{p}{2}\log_2(p) - \frac{1-p}{2}\log_2(1-p)$.
- **Classification error ('misclassification_error'**: $\psi(p) = 1 - \max(p, 1-p)$.

At each split, the algorithm evaluates all possible feature thresholds and selects the one that **maximizes homogeneity or minimizes impurity**. This process continues recursively until a predefined stopping criterion is reached, such as:

- Maximum tree depth (max_depth)
- Minimum number of samples in a node (min_sample)
- Minimum reduction in impurity.

4 Hyperparameter Tuning

4.1 Hyperparameter Search

A **randomized search strategy** was employed for hyperparameter tuning to optimize the decision tree classifier's performance. The **hyperparameter grid** used for tuning is as follows:

```
param_grid = {
    'max_depth': [10, 20, 30, 40, 50],
    'min_samples': [5, 10, 20, 30],
    'criterion': ['gini', 'entropy', 'misclassification_error']
}
```

This random search consists of 5 possible values for tree depth, 4 values for the minimum number of samples required to split a node, and 3 different splitting criteria. In total, this results in **60 possible hyperparameter combinations**.

4.2 RandomizedSearchCV

Considering the nature of the dataset and the hyperparameter grid, **RandomizedSearchCV was chosen as the preferred approach** due to its efficiency and effectiveness.

To understand this option, is important to **compare the randomized search and the grid search**. Both explore exactly the same space of parameters, the result in parameter settings is quite similar, but **the run time for randomized search is drastically lower** [3].

As grid search tries all possible combinations, this leads both to high running time and worst, it could result in overfitting. On the other hand, random search tries different random combinations with the grid of parameters given. For this project, the class constructor included **n_iter** which limits the number of random combinations to explore, which for this purpose it was define in 10. So after trying 10 different random combinations with the parameters given by the grid, the **RandomizedSearchCV class** will output the best parameters for the model.

Additionally, to accelerate the hyperparameter search, **parallel processing** was employed by setting n_jobs=-1. This setting enables the algorithm to **utilize all available CPU cores for processing**. This maximizes computational efficiency by distributing the workload across multiple cores instead of running sequentially on just one [4].

Algorithm 2 RandomizedSearchCV

Input: $param\_grid, x\_train, y\_train, n\_iter = 10, cv = 5, scoring =' accuracy', seed, n\_jobs = -1$

if seed is not None **then**
    Set random seed to *seed*
end if
$best\_params \leftarrow None$
$best\_score \leftarrow -1$
for $i \leftarrow 1$ **to** n_iter **do**
    $params \leftarrow$ Randomly sample from $param\_grid$
    $scores \leftarrow$ Parallel evaluation over cv folds:
    for $j \leftarrow 1$ **to** cv **do**
        $scores[j] \leftarrow$ **evaluate_fold**$(params, x\_train, y\_train, j)$
    end for
    $mean\_score \leftarrow$ Mean of *scores*
    if $mean\_score > best\_score$ **then**
        $best\_score \leftarrow mean\_score$
        $best\_params \leftarrow params$
    end if
end for

Output: $best\_params, best\_score$

4.3 Hyperparameter Tuning Results

After running the random search, the function RandomizedSearchCV.fit() outputed the best parameters for the model:

```
Best parameters: 'max_depth': 50, 'min_samples': 5, 'criterion': 'entropy'
```

The **max_depth** parameter being **set to 50** allows the decision tree to grow deeper and capture complex patterns. Nevertheless, as mentioned before, it is important to remember that deeper trees may run the risk of overfitting.

The **min_samples** parameter is **set to 5**, meaning that a node must have at least 5 samples before it can be split. This improves the tree to avoid overfitting as it sets a minimum number of samples to avoid having nodes with very few samples.

Last but not least, the **criterion is set to entropy**, which measures the impurity in the dataset by **quantifying uncertainty** [5]. The model evaluates splits based on information gain, selecting features that reduce disorder the most. This approach helps create a decision tree that **effectively minimizes uncertainty and improves class separation**, leading to more informed and balanced decisions.

Finally, the best average cross-validation score achieved during tuning was:

```
Best cv score: 0.9962
```

The **cross-validation score of 99.62%** indicates that the model consistently achieved high accuracy across multiple validation folds. This strong performance suggests that the model generalizes well to unseen data, demonstrating its ability to make reliable predictions during training.

5 Decision Tree Implementation

In this section, it is detailed the implementation of the decision tree.

5.1 Node Class

The Node class represents an individual node in the decision tree structure. Each node can either be a leaf node, containing a class prediction, or a decision node that splits the data based on a feature.

5.1.1 Attributes

The constructor initializes with the following attributes:

- **feature**: The index of the feature used to split the data.
- **threshold**: The threshold value at which the split occurs for the feature.
- **left_leaf**: The left child node (subtree) of the current node.
- **right_leaf**: The right child node (subtree) of the current node.
- **prediction**: The class prediction for leaf nodes. This is used when the node is a leaf.

5.1.2 Methods

The Node class includes the following methods:

- **predict()**: This method recursively traverses the tree to predict the label for a given sample. It checks whether the current node is a leaf and returns the class prediction. If not, it continues down the left or right subtree based on the feature value of the sample.
- **is_leaf()**: This method returns True if the node is a leaf (i.e., it has a prediction), and False otherwise.

5.2 Decision Tree Class

The **DecisionTree class** is the responsible for building and managing the decision tree. It handles **fitting the model to data, making predictions, and evaluating the model's performance**.

5.2.1 Attributes

The constructor initializes with the following attributes:

- **max_depth**: The maximum depth to which the tree can grow.
- **min_samples**: The minimum number of samples required to split a node.
- **criterion**: The criterion used to evaluate splits. It can be 'gini', 'entropy' or 'misclassification_error'.

5.2.2 Methods

The main methods of the Decision Tree are:

- **fit()**: This method builds the decision tree by calling the recursive **_build_tree()** method. The algorithm starts with the root node and recursively splits the data based on the best feature and threshold, until the stopping conditions are met (max_depth, min_samples, or pure leaves).
- **_best_split()**: This method finds the **best feature and threshold** to split the data. It evaluates potential splits by calculating the information gain for each feature and threshold, using the chosen criterion (gini, entropy or misclassification_error). It also uses parallel processing to speed up the evaluation for each feature.

Algorithm 3 Best Split Evaluation

Input: X (features), y (target values)

$best\_feature \leftarrow None$
$best\_threshold \leftarrow None$
$best\_score \leftarrow -\infty$
$m, n \leftarrow$ shape of X
for each feature from 1 to n **do**
    $thresholds \leftarrow$ unique values of $X[:, feature]$
    for each threshold in thresholds **do**
        $left\_mask \leftarrow X[:, feature] \leq threshold$
        $right\_mask \leftarrow X[:, feature] > threshold$
        if sum of $left\_mask < min\_samples$ or sum of $right\_mask < min\_samples$ **then**
            continue
        end if
        $score \leftarrow$ information gain using $criterion$
        if $score > best\_score$ **then**
            $best\_score \leftarrow score$
            $best\_feature \leftarrow feature$
            $best\_threshold \leftarrow threshold$
        end if
    end for
end for
    return $best\_feature$, $best\_threshold$

6 Final Model Evaluation

When assessing the performance of a classification model, several key evaluation metrics are commonly used to quantify its effectiveness.

6.1 Evaluation Metrics

- **0-1 Loss**: This metric measures the **proportion of incorrect predictions** among all predictions made by the model. It can be expressed as:

$$L_{0-1} = \frac{1}{n} \sum_{i=1}^{n} \ell_i$$

  where:

  $-n$ = Total number of predictions.

  $-\ell_i$ = Loss for the i-th sample.

- **Accuracy**: is one of the simplest and most widely used metrics, measuring the proportion of correct predictions out of the total predictions made. It is defined as:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

- **Precision**: evaluates how many of the instances predicted as positive are actually correct. A high precision score indicates that the model makes fewer false positive errors. It is given by:

$$Precision = \frac{TP}{TP + FP}$$

- **Recall**: measures the proportion of actual positive cases that the model correctly identifies. A high recall means the model successfully captures most of the actual positive cases, minimizing false negatives. It is calculated by:

$$Recall = \frac{TP}{TP + FP}$$

- **F1 Score**: is the harmonic mean of precision and recall, providing a balanced measure when there is an uneven class distribution. This metric is particularly useful when both precision and recall are important, ensuring that neither is disproportionately favored. It is formulated as:

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

- **Confusion Matrix**: is a tabular representation (2x2) of model performance that provides insight into classification errors and helps in evaluating model effectiveness beyond a single numerical metric. It summarizes the true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN):

	Predicted Positive	**Predicted Negative**
Actual Positive	TP (True Positive)	FN (False Negative)
Actual Negative	FP (False Positive)	TN (True Negative)

6.2 Model's Results

After getting the best parameters and running the tree with them, the metrics' results were as follow:

- **0-1 Loss for Training Set**: 0.14%
- **0-1 Loss for Test Set**: 0.31%
- **Accuracy**: 99.69%
- **Precision**: 99.62%
- **Recall**: 99.69%
- **F1 Score**: 99.66%

For the **confusion matrix**:

- **TP (True Positive**: 5,542
- **FP (False Positive**: 21
- **TN (True Negative)**: 6,605
- **FN (False Negative**: 17

7 Conclusion

The **decision tree classifier** implemented in this project **successfully predicts** whether a mushroom is poisonous or edible based on various attributes. The model underwent **data pre-processing, hyperparameter tuning, and evaluation** to ensure its reliability. By using **entropy** as the splitting criterion, a **max_depth** of 50, and a **min_samples** size of 5, the final model achieved exceptional performance, with a **cross-validation accuracy of 99.62% and a test set accuracy of 99.69%**.

The **low 0-1 loss values** indicate minimal misclassification, confirming that **the model generalizes well**. Additionally, metrics such as precision (99.62%) and recall (99.69%) demonstrate its robustness in distinguishing between edible and poisonous mushrooms.

Overall, this project effectively demonstrates the implementation of a decision tree from scratch and highlights its effectiveness in binary classification tasks.

Despite its strong results, **future work** could explore pruning techniques or trying different parameters to improve the model if possible, mainly because the max_depth is high and there could be risk of overfitting.

References

[1] Dennis Wagner, D. Heider, and Georges Hattab. *"Secondary Mushroom"*. UCI Machine Learning Repository, 2023, https://archive.ics.uci.edu/dataset/848/secondary+mushroom+dataset

[2] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification and Regression Trees*, CRC Press, 1984.

[3] Scikit-learn. *"Comparing randomized search and grid search for hyperparameter estimation"*, (n.d.), https://archive.ics.uci.edu/dataset/848/secondary+mushroom+dataset

[4] Joblib Navigation. *"joblib.Parallel"*, (n.d.), https://joblib.readthedocs.io/en/latest/generated/joblib.Parallel.html

[5] Niranjan Appaji. *"A Tale of Two Metrics: Gini Impurity and Entropy in Decision Tree Algorithms"*, 2024, https://niranjanappaji.medium.com/a-tale-of-two-metrics-gini-impurity-and-entropy-in-decision-tree-algorithms-7f61d46baacc

A Decision Tree Real Testing

After the project was finished, a random example was uploaded to test the tree and see the result:

```
new_mushroom = np.array([[
    5.2,  # cap-diameter
    0, 1, 0, 0, 0, 0, 0,  # cap-shape (x = 1, rest 0)
    0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,  # cap-surface (y = 1)
    0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  # cap-color (w = 1)
    0, 1,  # does-bruise-or-bleed (t = 1)
    1, 0, 0, 0, 0, 0, 0,  # gill-attachment (f = 1)
    0, 0, 1,  # gill-spacing (c = 1)
    0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,  # gill-color (k = 1)
    10.1,  # stem-height
    1.2,   # stem-width
    0, 1, 0, 0, 0, 0, 0, 0,  # stem-surface (s = 1)
    1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  # stem-color (g = 1)
    0, 1,  # has-ring (t = 1)
    0, 0, 0, 0, 0, 1, 0, 0,  # ring-type (p = 1)
    0, 0, 0, 0, 1, 0, 0, 0,  # habitat (w = 1)
    0, 0, 1, 0  # season (s = 1)
]])

prediction = final_tree.predict(new_mushroom)

if prediction[0] == 1:
    print("The mushroom is POISONOUS, DON'T eat it.")
else:
    print("The mushroom is EDIBLE, you can eat it.")

The mushroom is POISONOUS, DON'T eat it.
```